Security Project Report

ECE 422 Winter 2019

By: Nathan Klapstein (1449872) and Thomas Lorincz (1461567)

Table of Contents

Abstract

A Secure File System (SFS) is a filesystem-managing technology or application that provides filesystem-level encryption, user authentication, and checksum defense against file corruption. SFS systems are designed to work in concert with an existing filesystem so that multiple users can access a central file server while maintaining mutual distrust among all parties. Like many modern filesystems, an SFS should support user-level, group-level, and admin-level permissions for accessing, modifying, and executing files.

Introduction

The purpose of this project was to create a SFS that allows users to store data on an untrusted file server. The SFS is intended to run on a user's machine so that they can connect with their encrypted files that are hosted on the Cybera cloud platform.

Our solution to these project requirements was DeadSFS, a SFS written in Python that runs on Linux, Windows, and OSX environments through a shell interface. DeadSFS communicates encrypted commands to a Linux-based FTP server that can relay its secured filesystem information to the user when requested.

Design Factors

Programming Language

DeadSFS was written in Python because we have more experience in Python than any other language. While we would have liked to take the opportunity to learn a new language (like Go), we concluded that our unfamiliarity with a new language may lead to an insecure system due to implementation errors.

Python is a highly-used scripting language that focuses on simplifying its APIs so that programmers can be more productive. On top of the increased productivity, we were also able to leverage the extensible Cmd2 and ArgumentParser libraries to provide a cohesive and familiar shell interface to users.

Another Python library that we were able to leverage was PyNaCl (SecretBox), a well-tested, highly-used binding to the libsodium C++ library. libsodium is an all-in-one cryptography package that allows users to create digital signatures, secret-key encryption, public-key encryption, hashing, and message authentication.

Technologies

TLS

TLS is a cryptographic protocol that is used to provide end-to-end encryption of data that is communicated across devices. In TLS, data is communicated privately through symmetric-key cryptography with a shared secret key that is negotiated at the start of a TLS session (part of the TLS handshake).

In DeadSFS, TLS is used to secure the communication between the client running the DeadSFS client shell and the DeadSFS server that is running on a machine in the Cybera cloud.

FTP/VSFTPD

File Transfer Protocol (FTP) is a standard network protocol for transferring files between computers. FTP is often combined with TLS in order to secure the files being transferred.

VSFTPD is a high-speed, secure FTP server for Unix-like systems. DeadSFS server extends the functionality of VSFTPD to provide a high quality filesystem experience to users with stable core functionality (VSFTPD is used in many high-traffic production servers).

File Checksums (Salsa20)

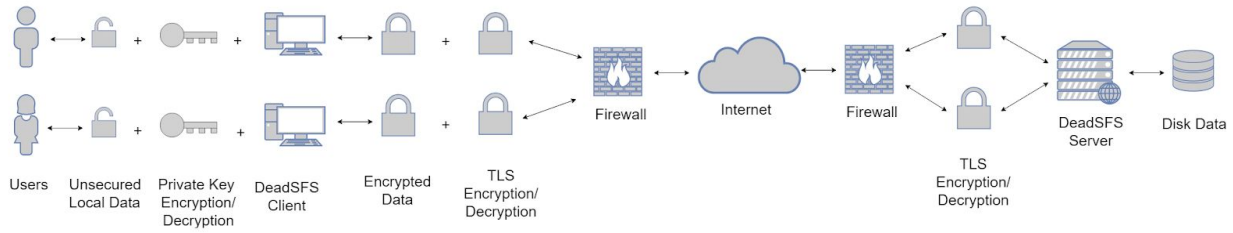
To ensure that files have not been compromised, regular checksums are computed for files to ensure that they have not been modified without permission. The algorithm behind this checksum hash is Salsa20. Usage of Salsa20 for message authentication is supported in the PyNaCl library.

Salsa20 is a stream cipher that supports 128-bit or 256-bit keys. It is typically performed in 20 rounds of add-rotate-xor operations, though as few as 15 rounds provides significant defence against cryptanalysis.

Design Artifacts

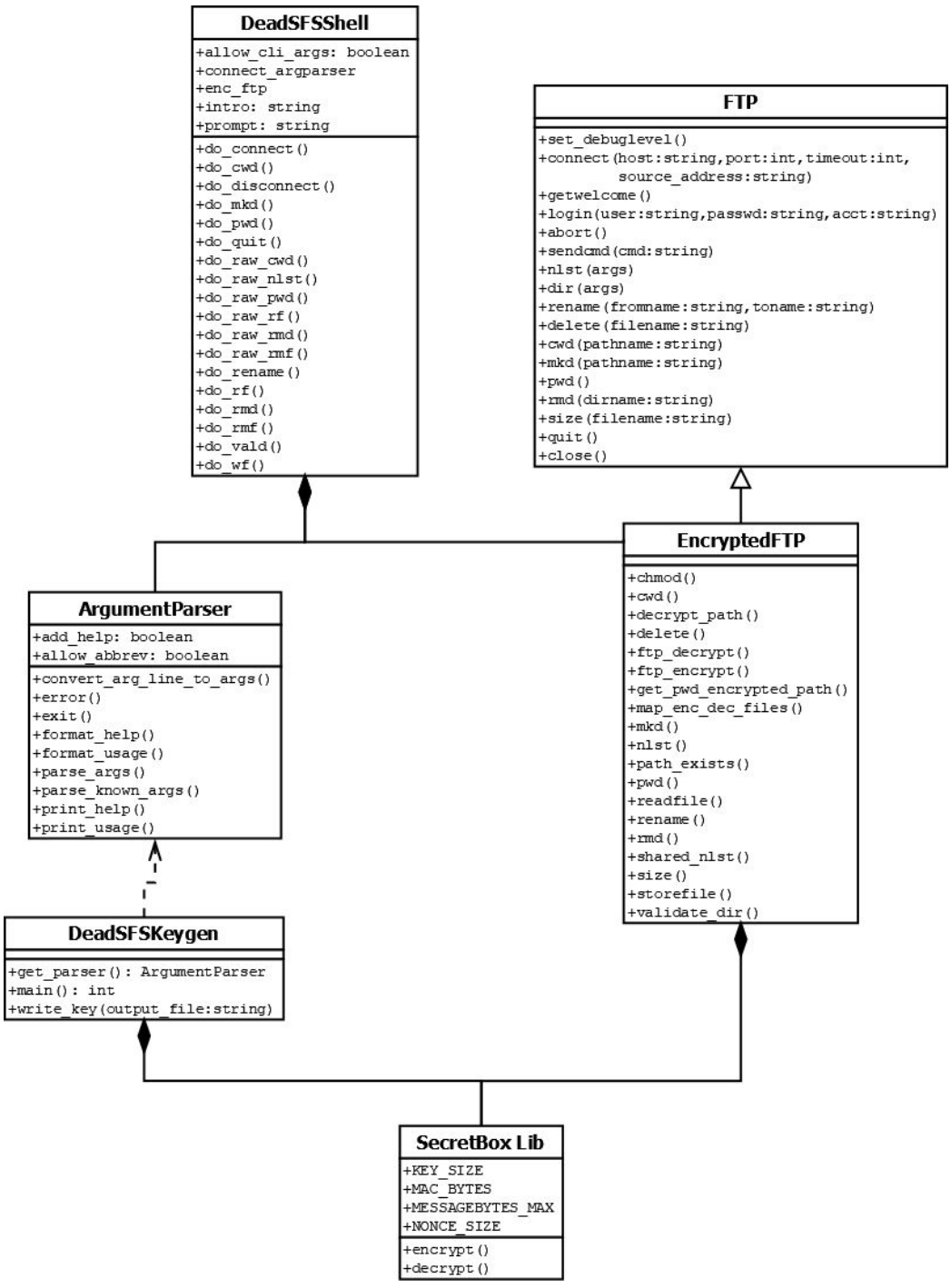
High-Level Architecture Diagram

Below is the high-level architecture for DeadSFS. It shows how two users may be interacting with the system. A highlight of our design is the combination of private key encryption and TLS encryption in the client application before the information/requests are sent to the DeadSFS server.



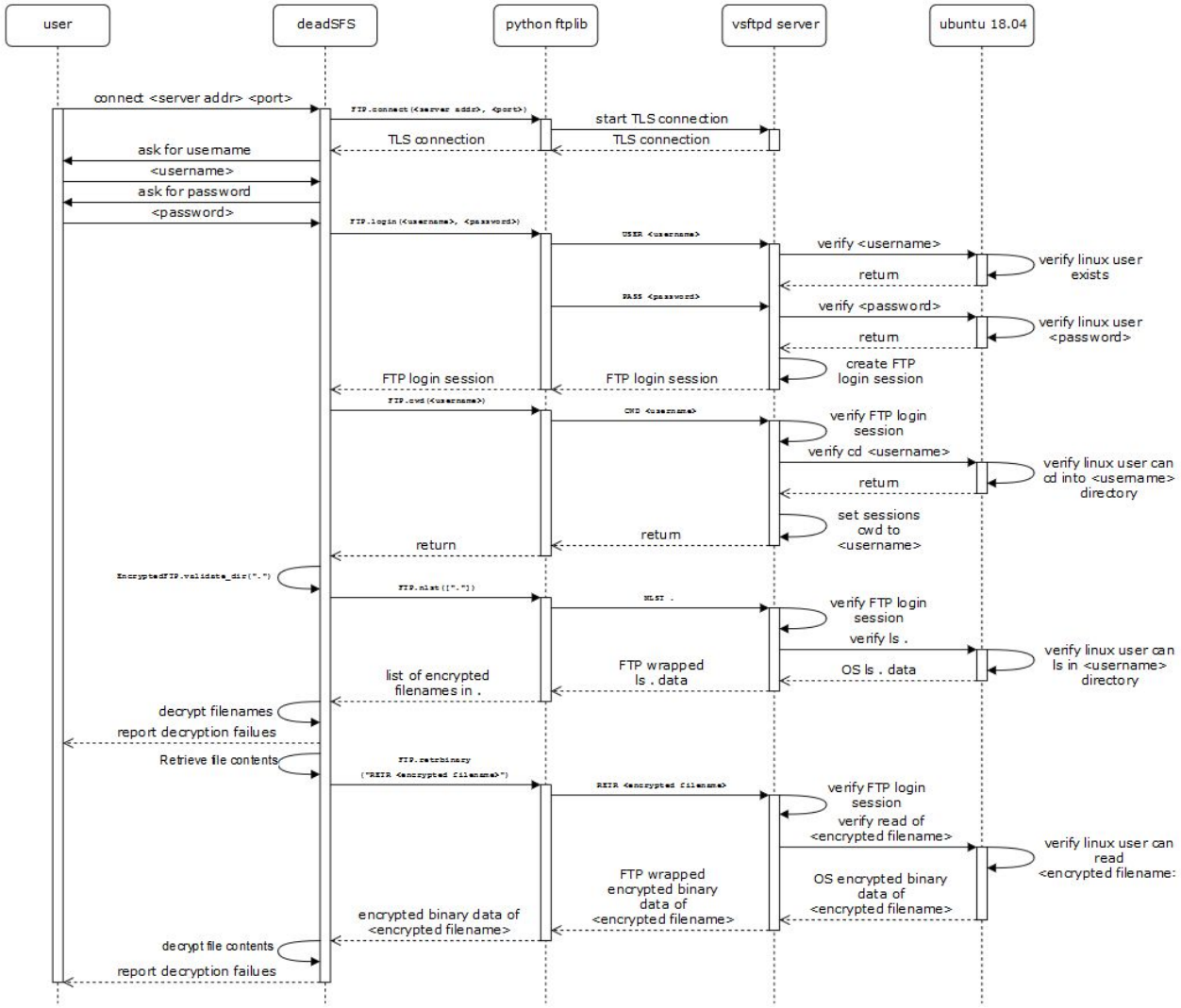
UML Class Diagram

The following UML class diagram outlines the structure of the DeadSFS client shell.



UML Sequence Diagram (Login + nlst command)

The following sequence diagram walks through the use case of logging in and then running the “nlst” command (similar in function to “ls”). After login is completed, the user has access to all commands that a common FTP server for a Linux filesystem supports (chmod, cwd, pwd, mkd, etc.).



Deployment

Server

For the project's server side deployment the vsftpd FTP server package was used within a Ubuntu 18.04 LTS operating system. vsftpd was chosen as it is a proven FTP server that has history of secure usage and limited exploit vectors. Additionally, it is easy to configure into a relatively secure, fast, and manageable FTP server. The Ubuntu 18.04 LTS operating system was chosen as it is easy to set up, and supported vsftpd without any odd configurations. Furthermore, securing users and groups within Linux / Ubuntu was relatively easy and connected well with the creation of users and groups (i.e creating a user/group in the Ubuntu host created a FTP user that could connect to the hosted vsftpd FTP server).

A simple outline of the commands and scripts used to setup the FTP server for the project are as follows:

A Ubuntu 18.04 LTS cloud machine was setup on Cybera Rapid Access Cloud by following standard practices documented within the Rapid Access Cloud Guide: Part 1 page. Additionally, ports 21 and 10000 to 10100 were opened for incoming UDP/TCP connections. This was done for supporting later deployment of a PASV mode enabled FTP server.

After the Ubuntu machine was setup and operating, vsftpd was installed on it via the apt-get command `sudo apt-get install vsftpd`.

Next, the vsftpd configuration file (located at `/etc/vsftpd.conf`) was modified to follow an outline similar to that provided in deadSFS's repo in `/server/vsftpd.conf`.

Next, the FTP filesystem and group creation was executed by running the bash script provided in `/server/setup_server.sh`. This created the ftpuser Linux user group, and created the default FTP home directory located at `/var/ftp/ece_422_security_project/home/` with proper permissions.

The FTP users and access groups were created by running the bash script provided in `/server/add_user.sh`. It would be invoke similar to `./add_user.sh <username> <group>` and this would create a FTP user with username `<username>` and group `<group>`. Additionally, a password would be set using the stdin console input. The `/server/add_user.sh` bash script creates and sets the FTP users home directory to `/var/ftp/ece_422_security_project/home/<username>` with proper group and user permissions while still retaining access to the base FTP home directory `/var/ftp/ece_422_security_project/home`.

Next, the vsftpd service was enabled and started in `systemctl` to ensure that the FTP server was consistently up, operating nominally. This was done via the following commands:

```
sudo systemctl stop vsftpd.service
sudo systemctl start vsftpd.service
sudo systemctl enable vsftpd.service
```

Thus, after creating a valid Linux based FTP server and setting up proper FTP users via the above steps, we were then able to use the deadSFS client to connect to the FTP server and encrypt and upload / download data in a secure fashion.

Client

Deployment of the deadSFS client is easy with a proper Python 3.6 environment supporting the Python package manager pip. deadSFS can be installed from source by running `pip install .` within the same directory as deadSFS's `setup.py` file.

User Guide

Server

After deploying the remote FTP server and configuring the complimenting Linux users and filesystem for deadSFS, server side usage is mostly hands free.

The only server side usage case after initial setup of the deadSFS server side components is the addition of new FTP users. Using the bash script `/server/add_user.sh` ensures that proper user and file system modifications are done to correctly add new FTP users to the deadSFS. See the above statements on server deployment for a explanation on using the `/server/add_user.sh` script.

Client

After installing the deadSFS client, a private encryption key for end to end encryption with deadSFS must be generated before running the deadSFS shell. This can be done by running `deadSFS-keygen <output_path>` command. This generates a private key at the path specified by `<output_path>` for later use in the `deadSFS-shell` command.

Next, the deadSFS shell can be started by the following console command `deadSFS-shell <keyfile>`. To get additional usage help on starting the deadSFS shell run `deadSFS-shell --help`. After starting the `deadSFS-shell` and obtaining the `deadSFS>` command prompt, the `help` command can be used to obtain a list of available commands within the `deadSFS-shell`. Additionally, the `help <command_name>` command can be used to obtain help on a specific command available to the deadSFS shell.

After starting the deadSFS shell, one then can connect to a hosted FTP server via the `connect <server_addr> <server_port>` command and specifying their username and password when prompted. If the login is successful they will be transmitted into their FTP home directory and its contents will be validated against unwarranted modification. Afterwards, the deadSFS shell should be analogous to other simple FTP command line clients, allowing users to upload/download files and modify their remotely hosted filesystem.

Once the user is set up and connected to the server, they have access to commands similar to those supported in many FTP servers:

`cwd <new_directory>` - Change working directory
`disconnect` - Disconnect from the FTP server
`mkd <directory_name>` - Make a new directory
`nlst` - List the contents of the current working directory
`pwd` - Print working directory
`rename <old_name> <new_name>` - Rename a file or directory
`rf <file_name>` - Decrypt, read, and save a file
`rmd <directory_name>` - Remove a directory
`rmf <file_name>` - Remove a file
`vald <directory_name>` - Validate that all files within a specified directory are properly encrypted and not tampered with
`wf <file_name>` - Encrypt and write a new file

Conclusion

Through completing this project we obtained valuable experience in developing an application to be secure. The development of DeadSFS made us consider many challenges associated with keeping data secure. Framing this project as a user application challenged us to consider not just confidentiality and integrity, but also availability. In each decision that we made, we had to consider the quality of the user experience.

Thus, we are proud of DeadSFS and will most likely use components or methods learned in the development of it in our future projects.

References

pynacl secret key usage:
https://pynacl.readthedocs.io/en/stable/secret/

Salsa20:
https://cr.yp.to/salsa20.html

Cybera Rapid Access Cloud Guide: Part 1:
Rapid Access Cloud Guide: Part 1

vsftpd Information
https://security.appspot.com/vsftpd.html

Creating new vsftpd users:
https://serverfault.com/questions/544850/create-new-vsftpd-user-and-lock-to-specify-home-login-directory

Configuring vsftpd with SSL/TLS:
https://www.liquidweb.com/kb/configure-vsftpd-ssl/