Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU33032 Advanced Computer Networks
Group Cryptography System

Ted Johnson, TCD 19335618
17th of April, 2022

Overview of Solution

The primary focus of this project was to develop a method of securing public messages between members of a group who share no secure channels of communication. The application of such a system was so that members could communicate privately via public social media posts. Additionally, I wanted to use Python as it seemed like an appropriate language and I was unfamiliar with its networking and multiprocessing capabilities.

I chose to use symmetric cryptography to encrypt messages sent between members. Each member has access to a shared session key which is generated by the authoritative group server. This session key is transmitted to each member through asymmetric cryptography - Clients send their public key to the server and, if recognised as a member of the group, receive the session key encrypted using that public key. As such, only clients who possess both the identifying public key and the corresponding private key can access the session key. Members can be added by simply storing their public key while members can be removed by regenerating and redistributing the session key to the remaining members.

Initially, I had considered using a specialised form of attribute-based encryption to prevent the need of regenerating and distributing a fresh session key after removing a member - Instead, members could just exclude the ex-members corresponding attribute from their local attribute list. However, such a system would require some form of coordination between all members to remove the ex-member from their local attribute list. As such, I deemed it justifiable to maintain a fresh session key between all members of the group instead.

A separate .mp4 video demonstration has been provided with this document. In this video, I set up a `gryptserver` to generate session keys and one `gryptd` daemon for Alice and Bob each. I use the `grypt` command-line client to issue requests to Alices and Bobs daemons who communicate with the `gryptserver`. Grypt is a portmanteau of group encrypt.

Implementation

I used Fernet (AES CBC, PKCS7, HMAC) for member message encryption and file encryption (using password-derived keys and salting). I used RSA for server-member session key encryption. Networking was not the focus of this project but was accomplished using sockets and named pipes.

Limitations

It is possible to send a user's public key to the server to find out if the user belongs to the group. This defeats the purpose of encrypting the member list on the server. Additionally, the system does not make much of an attempt to prevent man-in-the-middle attacks where a third party discreetly intercepts and replaces the returned encrypted session key. This could be addressed with private key signatures to verify origins.

Project Dependencies

This is a Python 3 project developed for a Linux-based operating system. The program uses the non-standard but widely used `rsa` Python library to perform asymmetric encryption. If not already, it can be easily installed with `pip install rsa`.

Code Listing And Explanation

The remaining pages in this document provide a listing of the code in this project and a short explanation of some design decisions. A separate .zip file containing the project has also been provided for your convenience.

./grypt

This simple script serves as the client for the whole project. Users can use this program within Bash scripts as it integrates very well with standard I/O - Clear text can be piped in using stdin and the resulting ciphertext can be piped out from stdout.

There is nothing particularly note-worthy within this script - It mainly provides an easy interface for users, formats input/output and communicates with the local `gryptd` daemon.

```python
#!/bin/python

import sys
import argparse
from multiprocessing.connection import Client

DEFAULT_IPC_PORT = 19119

if __name__ == "__main__":

    # Parse arguments
    argparser = argparse.ArgumentParser(
        description="Encrypt and decrypt group chat messages with grypt",
        epilog="Ted Johnson <edjohnso@tcd.ie>")
    argparser.add_argument("-d", "--decrypt", action="store_true", help="decrypt input
text instead")
    argparser.add_argument("-p", "--port", type=int, help="port connect to gryptd with")
    argparser.add_argument("server", help="IP address of group server")
    argparser.add_argument("text", nargs="?", help="input text or none for stdin")

    args = argparser.parse_args()
    port = args.port or DEFAULT_IPC_PORT
    text = args.text or sys.stdin.read()

    # Get gryptd to encrypt or decrypt
    try:
        with Client(("localhost", port)) as conn:
            conn.send((args.server, args.decrypt, text))
            success, message = conn.recv()
    except ConnectionRefusedError:
        print("Unable to connect to gryptd. Is it running?", file=sys.stderr)
        exit(1)

    if not success:
        print(message, file=sys.stderr)
        exit(1)

    # Write response to stdout
    print(message)
```

./gryptd

I would say this daemon script is the core of the project. The client script issues encryption/decryption requests which it must try to fulfil by using the requested server's session key to perform symmetric encryption. I used the cryptography library to accomplish this. Additionally, persistent connections are kept to previously contacted servers to receive updated session keys using public-key cryptography. Finally, the daemon manages generation and password-protected storage of the user's public and private keys on disk.

A daemon must first generate the user's keypair if the encrypted file does not yet exist. This file is then encrypted using a password derived key and will be loaded on subsequent starts. If the daemon receives an encryption request with a session key it does not have, it must contact the group server. The daemon sends the user's public key and, if the server accepts the public key, receives the public key encrypted session key. The daemon decrypts the session key with the user's private key and persists the server's connection on a new thread in anticipation of session key updates due to member removal. Finally, the daemon responds to the client with the requested encryption/decryption using the fresh session key.

```python
#!/bin/python

import os
import sys
import pathlib
import json
import base64
import argparse
import threading
import getpass
import rsa
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from multiprocessing.connection import Client, Listener

USER_KEY_NBITS = 1024
PASSWORD_SALT_LENGTH = 16
DEFAULT_IPC_PORT = 19119
DEFAULT_SRV_PORT = 19118
DEFAULT_KEYFILE = os.getenv("XDG_DATA_HOME", os.getenv("HOME", "") + "/.local/share") +
"/grypt/user"

def derive_key(password, salt):
    """Derive Fernet key from password and salt"""
    kdf = PBKDF2HMAC(algorithm=hashes.SHA256(), length=32, salt=salt, iterations=390000)
    return base64.urlsafe_b64encode(kdf.derive(password.encode()))

def subscribe_to_group_session_key(sessions: dict, addr: str, user: tuple) -> None:
    session = sessions[addr]
    server = session["connection"]

    try:
        # Update session key when group server pushes new key out
        while True:
            cipher = server.recv()

            # An empty response means user is not a group member
            if not cipher:
                print("Group %s rejected user." % addr)
                break

            # Decrypt session key with public-key cryptography
            session_key = rsa.decrypt(cipher, user[1])

            # Update session key
            with session["lock"]:
                print("New session key received from group %s." % addr)
                session["key"] = session_key
                session["lock"].notify_all()

    except EOFError:
        print("Group %s has disconnected. The session will have to be re-established
when used again." % addr)

    # The thread removes the session itself
    server.close()
    with session["lock"]:
        session["key"] = ""
        session["lock"].notify_all()
        del sessions[addr]


if __name__ == "__main__":

    # Parse arguments
    argparser = argparse.ArgumentParser(
        description="Daemon for handling persistant connections to gryptservers",
        epilog="Ted Johnson <edjohnso@tcd.ie>")
    argparser.add_argument("-k", "--keyfile", help="file containing user key, defaults
to $XDG_DATA_HOME/grypt/user")
    argparser.add_argument("-p", "--port", type=int, help="port to use to listen for
grypt client connections")
    args = argparser.parse_args()
    ipc_port = args.port or DEFAULT_IPC_PORT
    keyfile = args.keyfile or DEFAULT_KEYFILE

    # Map groups to session keys
    sessions = dict()

    try:

        # Keep the user data in an password-encrypted file
        try:
            # Read keyfile to get user keypair
            with open(keyfile, "rb") as f:
                data = f.read()
```

```python
            password = getpass.getpass("Password: ")
            secret = data[:PASSWORD_SALT_LENGTH]
            fernet = Fernet(derive_key(password, secret))
            # Decrypt
            data = fernet.decrypt(data[PASSWORD_SALT_LENGTH:]).decode()
            # Deserialise
            user_pkcs = json.loads(data)
            user = (
                rsa.PublicKey.load_pkcs1(user_pkcs[0].encode()),
                rsa.PrivateKey.load_pkcs1(user_pkcs[1].encode())
            )
        except FileNotFoundError:
            # Generate a new user keypair and write to file
            print("User file not found. Creating new user...")
            password = getpass.getpass("Password for %s: " % keyfile)
            secret = os.urandom(PASSWORD_SALT_LENGTH)
            fernet = Fernet(derive_key(password, secret))
            # Generate new keypair
            print("Generating new user keypair...")
            user = rsa.newkeys(USER_KEY_NBITS)
            # Serialise
            user_pkcs = (
                user[0].save_pkcs1().decode(),
                user[1].save_pkcs1().decode()
            )
            user_pkcs_str = json.dumps(user_pkcs)
            # Encrypt
            data = fernet.encrypt(user_pkcs_str.encode())
            # Write to file
            pathlib.Path(keyfile).parent.mkdir(parents=True, exist_ok=True)
            with open(keyfile, "wb") as f:
                f.write(secret + data)

        # Accept client requests
        with Listener(("localhost", ipc_port)) as listener:
            print("Daemon listening on port", ipc_port)
            while True:
                with listener.accept() as client:

                    print("\nHandling new client: ", end="")

                    # Parse client requests and get corresponding group session
                    addr, decrypt, text = client.recv()

                    print(
                        "Decrypt" if decrypt else "Encrypt",
                        "'" + text + "'",
                        "for group", addr)

                    # If no session is found, attempt to join session
                    session = sessions.get(addr)
                    if not session:

                        print("Session not found, contacting %s..." % addr)
                        try:

                            # Send public key to group server
                            server = Client((addr, DEFAULT_SRV_PORT))
                            print("Sending user public key...")
                            server.send(user[0])

                            session = { "connection": server, "lock":
threading.Condition() }
                            sessions[addr] = session

                            with session["lock"]:

                                # Spawn a thread to receive responses from server
containing new session keys
                                threading.Thread(
                                    target=subscribe_to_group_session_key,
                                    args=(sessions, addr, user)
                                ).start()

                                # Wait for the first response
                                session["lock"].wait()

                                # If response is empty, the use has been rejected
                                if not session["key"]:
                                    err = "User is not a member of %s." % addr
                                    print(err, file=sys.stderr)
                                    client.send((False, err))
                                    continue # The thread will clean the session up
itself

                        except ConnectionRefusedError:
                            err = "Failed to connect to %s." % addr
                            print(err, file=sys.stderr)
                            client.send((False, err))
                            continue

                    # Encrypt/decrypt text with session key
                    with session["lock"]:
                        print("Decrypting" if decrypt else "Encrypting", "input with
session key...")
                        byte_text = text.encode()
                        fernet = Fernet(session["key"])
                        result = fernet.decrypt(byte_text) if decrypt else
fernet.encrypt(byte_text)
                        client.send((True, result.decode()))
                        print("Client request complete.")

    except KeyboardInterrupt:
        pass

    except OSError:
        print("Unable to bind to port %s. Is another gryptd instance already running?" %
ipc_port, file=sys.stderr)
```

./gryptserver

Finally, the server script is run on a remote machine and handles generating session keys and authenticating users by their public keys. Similarly to the daemon, this program also uses password-derived keys to encrypt the current session key and group member list on disk. This again allows restarts and prevents leaking information. Additionally, this script uses a separate thread to listen to admin command input to manage the member list.

Fundamentally, the server securely sends the session key to recognised users. This is achieved by keeping a list of public keys as the member list. If the server receives a public key on its member list, it sends a session key encrypted with the public key. As such, only an authenticated user with the corresponding private key can access the session key used to encrypt and decrypt messages within the group.

To remove a user from the group, their public key is removed from the member list, a new session key is generated and finally every member with a persistent connection is securely sent the fresh session key. Notice this system is set up in such a way to eliminate the chance of use of an old session key - Active members immediately receive the fresh session key through the persistent connection while inactive members must connect to the server before being sure they have a fresh session key.

```python
#!/bin/python

import os
import sys
import pathlib
import json
import base64
import argparse
import threading
import getpass
import rsa
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from multiprocessing.connection import Listener

PASSWORD_SALT_LENGTH = 16
DEFAULT_SRV_PORT = 19118
DEFAULT_GROUP_FILE = os.getenv("XDG_DATA_HOME", os.getenv("HOME", "") + "/.local/share")
+ "/grypt/group"

def derive_key(password, salt):
    """Derive Fernet key from password and salt"""
    kdf = PBKDF2HMAC(algorithm=hashes.SHA256(), length=32, salt=salt, iterations=390000)
    return base64.urlsafe_b64encode(kdf.derive(password.encode()))

def save_group_data(group, active):
    """Write encrypted group data to file"""
    # Serialise
    serialised_group = { "session_key": group["session_key"], "members": [] }
    for member in group["members"]:
        serialised_group["members"].append(member.save_pkcs1().decode())
    # Encrypt
    group_str = active["fernet"].encrypt(json.dumps(serialised_group).encode())
    # Write to file
    pathlib.Path(groupfile).parent.mkdir(parents=True, exist_ok=True)
    with open(groupfile, "wb") as f:
        f.write(active["secret"] + group_str)

def send_session_key(active, public_key):
    """Encrypts and pushes the session key to a client"""
    data = rsa.encrypt(active["session_key"], public_key)
    active["members"][public_key].send(data)

def admin_input_handler(active, group):
    while True:
        command = input()
        if command == "regen":
            regenerate_session_key(active, group)
        elif command == "add":
            add_member(active, group)
        elif command == "kick":
            remove_member(active, group)
        else:
            print("Unknown command", file=sys.stderr)

def regenerate_session_key(active, group):
    print("Generating new session key...")
    active["session_key"] = Fernet.generate_key()
    group["session_key"] = active["session_key"].decode()
    save_group_data(group, active)
    # Send new session key to other members
    print("Pushing public-key encrypted session key to active members...")
    for member in active["members"]:
        send_session_key(active, member)
    print("Done.")

def add_member(active, group):
    """Add user public key to group members list"""
    if "latest_public_key" not in active:
        print("No recent rejected users.", file=sys.stderr)
        return
    group["members"].append(active["latest_public_key"])
    save_group_data(group, active)
    print("Added client to group members.")

def remove_member(active, group):
    """Remove user public key from group members list"""

    print("\nCurrent members:\n")
    for i, member in enumerate(group["members"]):
        print(str(i + 1) + ".\n" + member.save_pkcs1().decode())

    try:
        sel = int(input("Select user to remove: "))
        public_key = group["members"][sel - 1]
    except:
        print("Cancelled.")
        return

    group["members"].remove(public_key)
    if public_key in active["members"]:
        conn = active["members"][public_key]
        del active["members"][public_key]
        conn.send("")
        conn.close()
    print("Removed user from group members.")
    regenerate_session_key(active, group)


if __name__ == "__main__":

    # Parse arguments
    argparser = argparse.ArgumentParser(
        description="Authoritative server for generating and distributing grypt session
keys.",
        epilog="Ted Johnson <edjohnso@tcd.ie>")
    argparser.add_argument("-g", "--groupfile", help="file containing group members,
defaults to $XDG_DATA_HOME/grypt/group")
    args = argparser.parse_args()
    groupfile = args.groupfile or DEFAULT_GROUP_FILE
    srv_port = DEFAULT_SRV_PORT

    # Map active members to active connections
    active = dict()
    active["members"] = dict()

    try:

        # Keep the group data in an password-encrypted file
        try:
            # Read groupfile to get group data
            with open(groupfile, "rb") as f:
                data = f.read()
            password = getpass.getpass("Password: ")
            active["secret"] = data[:PASSWORD_SALT_LENGTH]
            active["fernet"] = Fernet(derive_key(password, active["secret"]))
            # Decrypt
            group_str = active["fernet"].decrypt(data[PASSWORD_SALT_LENGTH:])
            # Deserialise
            group = json.loads(group_str)
            for i, member in enumerate(group["members"]):
                group["members"][i] = rsa.PublicKey.load_pkcs1(member.encode())
        except FileNotFoundError:
            # Generate a new group and write to file
            print("Generating new group...")
            password = getpass.getpass("Create password for %s: " % groupfile)
            active["secret"] = os.urandom(PASSWORD_SALT_LENGTH)
            active["fernet"] = Fernet(derive_key(password, active["secret"]))
            group = { "session_key": Fernet.generate_key().decode(), "members": [] }
            save_group_data(group, active)
        # The key is stored as a string but is used as bytes
        active["session_key"] = group["session_key"].encode()

        # Create a thread to handle admin user input
        admin_input_thread = threading.Thread(
            target=admin_input_handler,
            args=(active, group))
        admin_input_thread.start()

        # Accept client requests
        with Listener(("localhost", srv_port)) as listener:
            print("Server listening on port", srv_port)
            while True:
                client = listener.accept()

                print("\nHandling new client:")

                # Parse client request
                public_key = client.recv()

                # Close previous active connection if one exists
                if public_key in active["members"]:
                    print("User was already connected as another client! Closing
previous connection.")
                    active["members"][public_key].close()
                    del active["members"][public_key]

                # Reject users not in group members list
                if public_key not in group["members"]:
                    print("Client is not on members list. Received following public
key:")

                    print(public_key.save_pkcs1().decode())
                    print("Use 'add' command to add client to members list.")
                    active["latest_public_key"] = public_key
                    client.send("")
                    client.close()
                    continue

                # Send session key using public-key cryptography
                print("Client is a valid member - Sending encrypted session key...")
                active["members"][public_key] = client
                send_session_key(active, public_key)
                print("Client request complete.")

    except KeyboardInterrupt:
        pass

    except OSError:
        print("Unable to bind to port %s. Is another gryptserver already running?" %
srv_port, file=sys.stderr)

    # Close the connection with every member still connected
    for member in active["members"]:
        active["members"][member].close()
```