CS 6903: Applied Cryptography Project 2

Securing an FTP Server

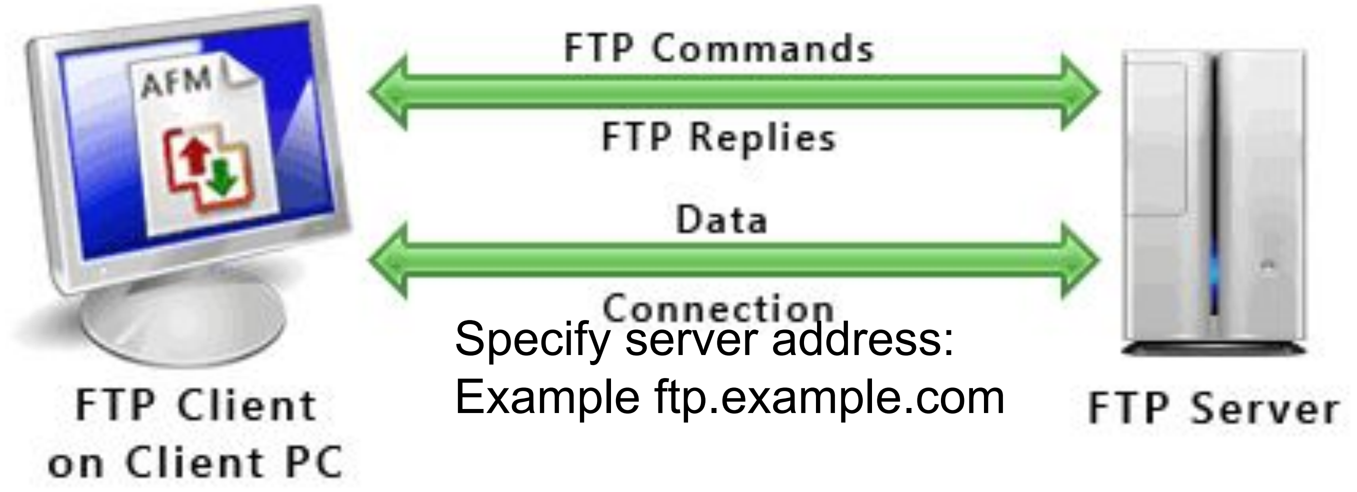
Members:
Redwanul Mutee
William Uchegbu
Abid Siam

Project Overview

- We will be securing a simple FTP (File Transfer Protocol) Server. **What is FTP?**
  - A way of transferring files online
  - The FTP **server** offers access to to a directory (on a computer network)
  - A user connects to the server using an FTP **client**, which is software that lets you download files from the server or upload files to it or typical web browser

Project Overview

Upload File, Download File



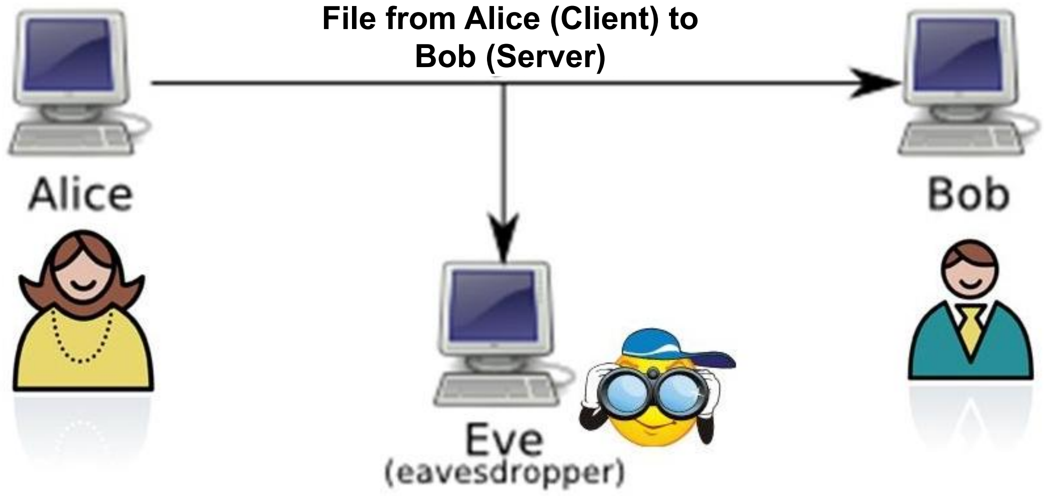
Specify server address:
Example ftp.example.com

Source: https://www.deskshare.com/resources/articles/ftp-how-to.aspx

Project Overview

- FTP was not design to be a secure protocol
  - Does not provide encryption for data
  - Attacker can easily modify traffic by injecting malware into downloads
- The Eavesdropper can gain access to the contents of the files transferred through the server. **Security measures** are necessary to prevent such attacks.
  - FTP Users can authenticate themselves using a username and password
  - FTP is often secured with **SSL/TLS (FTPS)** or replaced with **SSH File Transfer Protocol (SFTP)**
- SSL (Secure Sockets Layer) is the standard security technology to establish an encrypted link between a web server and a browser

Project Overview



Source: https://cs.wellesley.edu/~cs110/reading/cryptography-files/handout.html

Creating the Server

- A simple server was created using Python’s built-in ***socket*** library
    - The server checks if the client is requesting to upload a file, or to receive a file
    - The server prompts if the request was fulfilled, otherwise terminates the session

Securing the Server

- We utilized Python’s ***cryptography*** library common **cryptographic algorithms**
  - Symmetric ciphers - ***cryptography.fernet***
  - Asymmetric ciphers - ***cryptography.hazmat.primitives.asymmetric***
    - ***RSA with OAEP***
  - Message digests
  - Key derivation functions
- Fernet guarantees that a message encrypted using it cannot be manipulated or read without the key
  - Prevents Data Eavesdropping
  - Prevents Data Modification

So what is Fernet?

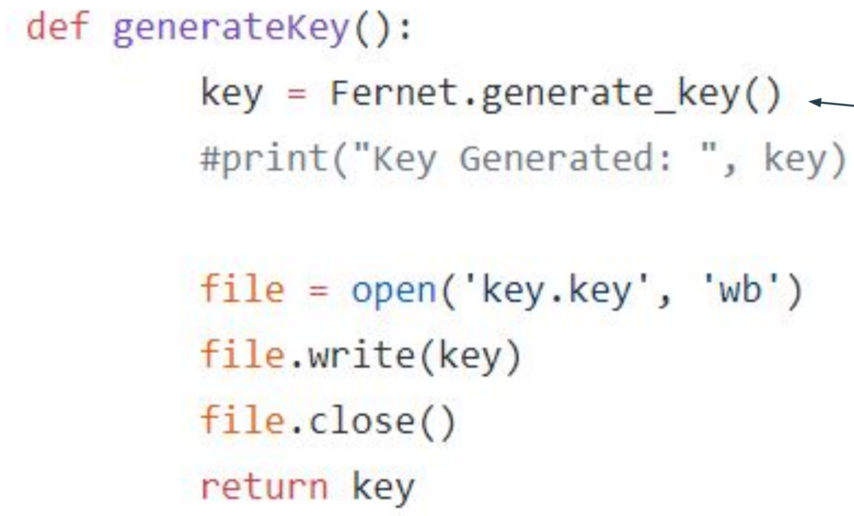
Fernet is a symmetric encryption method which makes sure that the message encrypted cannot be manipulated/read without the key. It uses URL safe encoding for the keys. Fernet also uses 128-bit AES in CBC mode and PKCS7 padding, with HMAC using SHA256 for authentication. The IV is created from os.random(). All of this is the kind of thing that good software needs.

AES is top drawer encryption, and SHA-256 avoids many of the problems caused by MD5 and SHA-1 (as the length of the hash values is too small). With CBC (Cipher Block Chaining) we get a salted output, and which is based on a random value (the IV value). And with HMAC we can provide authenticated access from both sides.

And for PKCS7, that's a standard too. Fernet is used to define best practice cryptography methods, and Hazmat supports core cryptographical primitives:

Securing the Server: Implementation

- Server Generates a Secret Key
    - They key gets stored in a .key file which is used later to verify the key



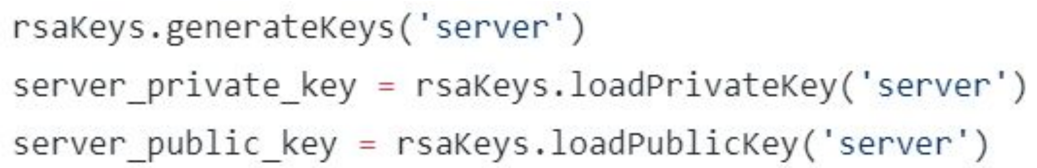
```
def generateKey():
        key = Fernet.generate_key()
        #print("Key Generated: ", key)

        file = open('key.key', 'wb')
        file.write(key)
        file.close()
        return key
```

returns a random URL-safe base64-encoded 32-byte key

Securing the Server: Implementation

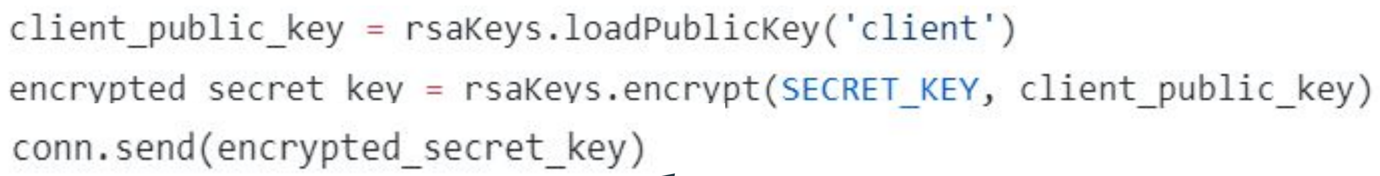
Acquiring the generated key: Asymmetric encryption - Server side

- Generate a public/private key pair for the server, same is done for client



```
rsaKeys.generateKeys('server')
server_private_key = rsaKeys.loadPrivateKey('server')
server_public_key = rsaKeys.loadPublicKey('server')
```

- Encrypt the generated key using the clients public key



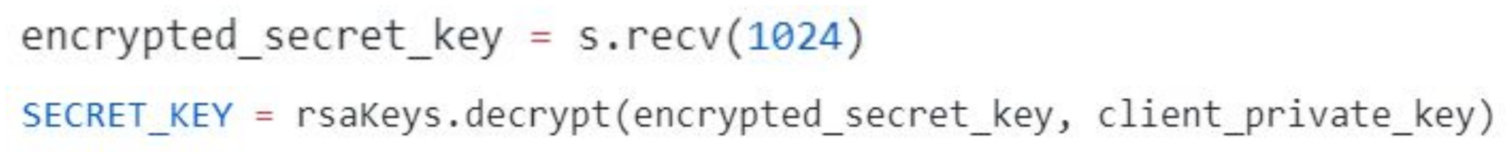
```
client_public_key = rsaKeys.loadPublicKey('client')
encrypted_secret_key = rsaKeys.encrypt(SECRET_KEY, client_public_key)
conn.send(encrypted_secret_key)
```

Only the clients private key can decrypt

Securing the Server: Implementation

Acquiring the generated key: Asymmetric encryption - Client side

- After receiving the key, the client attempts to decrypt using its own private key generated earlier along with its public key.



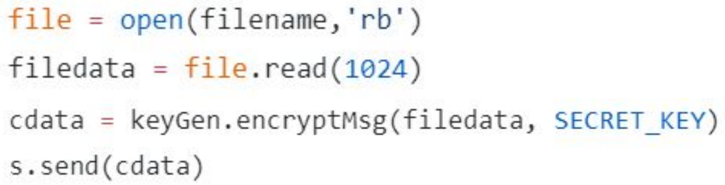
```
encrypted_secret_key = s.recv(1024)
SECRET_KEY = rsaKeys.decrypt(encrypted_secret_key, client_private_key)
```

- The generated key is now available to both server and client!

Securing the Server: Implementation

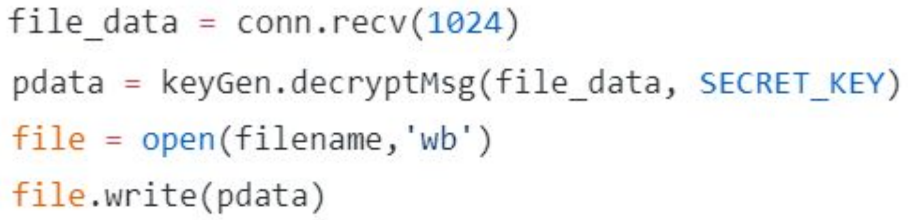
Encrypting the data: Symmetric encryption

- Encrypting outgoing file with shared secret key (Client)



```
file = open(filename,'rb')
filedata = file.read(1024)
cdata = keyGen.encryptMsg(filedata, SECRET_KEY)
s.send(cdata)
```

- Decrypting incoming file with shared secret key (Server)



```
file_data = conn.recv(1024)
pdata = keyGen.decryptMsg(file_data, SECRET_KEY)
file = open(filename,'wb')
file.write(pdata)
```

Future Additions

- Enable the server to handle larger file sizes
- Write MORE options
- Allow repeating listener
- Add checksums
- Use CA to back public keys
- Add error handling for file i/o

Sources

Martindale, Jon. “File Transfer Protocol Explained: What FTP Is and What It Does.” Digital Trends, 30 Mar. 2019, www.digitaltrends.com/computing/what-is-ftp-and-how-do-i-use-it/.

“Welcome to Pyca/Cryptography.” Welcome to Pyca/Cryptography - Cryptography 2.7. https://cryptography.io/en/latest/#