A Method for Secure, Peer-to-Peer File Transfer Between Devices

Date: 15 October 2024

Author: Micah Borghese

Abstract

The lack of interoperability between electronic devices, particularly across different brands and operating systems, often locks consumers into a single brand’s ecosystem, limiting their ability to switch to devices from other manufacturers. This constraint is driven by the seamless file-sharing features that are typically restricted to devices running the same operating system. In this technical paper, I propose a method for securely transferring and retrieving files between devices, regardless of brand or operating system, using cryptographic primitives and protocols. The solution relies on peer-to-peer technologies, eliminating the need for any centralized server to store sensisitve data, and ensures secure, end-to-end file transfers by establishing a cryptographically secure pairing between devices. This approach enables cross-brand interoperability while maintaining the privacy and security of shared data.

Information discussed is purely theoretical and might change as the application is built out.

Background

Elliptic-curve Diffie–Hellman (ECDH) Key Exchange

ECDH is a key exchange protocol that allows two parties to securely establish a shared secret over an unsecured communication channel. Each party generates a public-private key pair using a chosen elliptic curve. They exchange public keys, and by combining their private key with the other party's public key, they both independently compute the same shared secret. This secret can then be used to encrypt further communication. In this method, the key exchange process enables both the laptop and phone to generate a shared key after scanning a QR code, which is then used to encrypt and decrypt messages or files.

Here is a simple Python example of shared key generation:

```
In [1]:
from ecdsa import ECDH, NIST256p

local = ECDH(curve=NIST256p)
local.generate_private_key()
local_public_key = local.get_public_key().to_pem()

remote = ECDH(curve=NIST256p)
```

```python
remote.generate_private_key()
remote_public_key = remote.get_public_key().to_pem()

print("local_public_key:", local_public_key)
print("remote_public_key:", remote_public_key)
```

```
local_public_key: b'-----BEGIN PUBLIC KEY-----\nMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEz
C4l9KpKgYYi0AnXx6hKolJSPZrJQKx+R2M11Vqc\nlGE9sD/dvl38zmiS9oHV9QbJhsagsMdyG0y80dWhviaa
DA==\n-----END PUBLIC KEY-----\n'
remote_public_key: b'-----BEGIN PUBLIC KEY-----\nMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE
60yY9EzEcDvpSjAGZREsqhTLYL9zmbMAxcyCaxwN\nrUwDgdPZnvcLOqSgfo1+SoQyg6mP2jpUUeoF6xpYFJ7
tOQ==\n-----END PUBLIC KEY-----\n'
```

In [2]:
```python
local.load_received_public_key_pem(remote_public_key)
remote.load_received_public_key_pem(local_public_key)

print("local_sharedsecret_key == remote_sharedsecret_key")
print(f'{local.generate_sharedsecret()} == {remote.generate_sharedsecret()}')
```

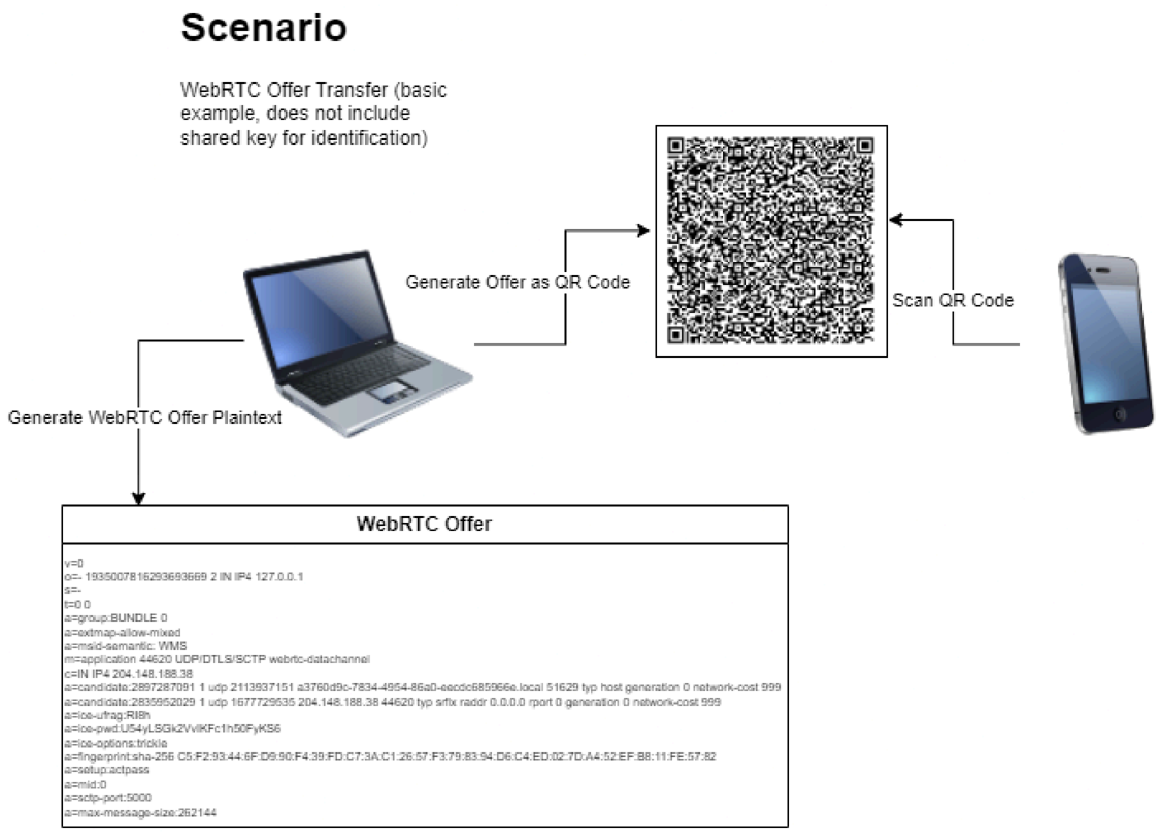
```
local_sharedsecret_key == remote_sharedsecret_key
48979041029840565157942177820486175174510085278310016679644548185509539329627 == 4897
9041029840565157942177820486175174510085278310016679644548185509539329627
```

WebRTC

WebRTC is set of APIs that allow for real-time communication (RTC) working inside of web browsers. The APIs work by establishing a offer-accept pipeline between two devices via a signaling server. After this offer is exchanged and accepted, a peer-to-peer connection is established and all data will be directly communicated between the devices with no central third party. WebRTC is supported on all major browsers. This method will utilize the `RTCDataChannel` endpoint to transfer files with ease. `RTCDataChannel` allows bidirectional communication of arbitrary data between peers. The data is transported using SCTP over DTLS. It uses the same API as WebSockets and has very low latency. Here is a simple Javascript example of `RTCDataChannel`.

QR Codes

When many people thing of QR codes they think of sharing websites. In fact, QR codes are so much more than that. QR codes are a protocol that bundles up information and allows for seamless data transfer. This method for file transfering harnesses the capability of QR codes to effecicently send an offer to the other device.


Scenario
WebRTC Offer Transfer (basic example, does not include shared key for identification)
Generate Offer as QR Code
Scan QR Code
Generate WebRTC Offer Plaintext
WebRTC Offer

```
v=0
o=- 1935007816293693669 2 IN IP4 127.0.0.1
s=-
t=0 0
a=group:BUNDLE 0
a=extmap-allow-mixed
a=msid-semantic: WMS
m=application 44620 UDP/DTLS/SCTP webrtc-datachannel
c=IN IP4 204.148.188.38
a=candidate:2897287091 1 udp 2113937151 a3760d9c-7834-4954-86a0-eecdc685966e.local 51629 typ host generation 0 network-cost 999
a=candidate:2835952029 1 udp 1677729535 204.148.188.38 44620 typ srflx raddr 0.0.0.0 rport 0 generation 0 network-cost 999
a=ice-ufrag:RI8h
a=ice-pwd:U54yLSGk2VvlKFc1h50FyKS6
a=ice-options:trickle
a=fingerprint:sha-256 C5:F2:93:44:6F:D9:90:F4:39:FD:C7:3A:C1:26:57:F3:79:83:94:D6:C4:ED:02:7D:A4:52:EF:B8:11:FE:57:82
a=setup:actpass
a=mid:0
a=sctp-port:5000
a=max-message-size:262144
```

How it works

Putting these three fundamental technologies together, it is possible to fully build out a fully-functional peer-to-peer system for transferring files between any device that can use a web browser connected to the internet. We will make use of ECDH shared keys for unique identification between devices, WebRTC for giving us peer-to-peer communication and file sharing, and QR codes for initial pairing.

There are three steps to set up the WebRTC data pipeline:

1. Set up Peer Connections: WebRTC establishes peer-to-peer connections through the `RTCPeerConnection` input. It needs two peers to connect and exchange SDP (Session Description Protocol) information via signaling. The signaling can be done using WebSockets, a server, or other communication methods (the signaling is out of WebRTC's scope, but is necessary to set up the connection as we discussed).

2. Create an `RTCDataChannel`: Once the peer connection is established, we can create an `RTCDataChannel` on one peer and have the other peer listen for the data channel connection. The data channel will allow file transfer between the peers.

3. Handle File Transfer: Use the `RTCDataChannel` endpoint to send file chunks between peers. Files can be too large to send in a single message, so we'll need to divide the file into smaller chunks, transfer them, and then reconstruct them on the receiving end.

More on Signaling Servers

The signaling server's role is simply to exchange messages (SDP offers, answers, and ICE candidates) between peers. It doesn't handle the actual WebRTC connection—that happens directly between the peers once signaling is done. So, the server's job is *just* to ensure that the correct messages are routed between the right clients.

Here's how the signaling server will handle peer connections:

1. Peer A and Peer B generate ECDH key pairs.
2. They exchange their public keys over the signaling server.
3. Both peers import the received public key and derive a shared secret.
4. The shared secret is used as a unique identifier for that WebRTC session, and optionally, for secure data encryption on the RTCDataChannel.

This shared secret can serve as a unique identifier for the session and can also be used to encrypt communication. The intention for private keys is they are stored in the device's local browser storage once generated. This local device storage allows devices to "remember" past paired devices and enable future features like forgoing QR code scanning once devices have already been paired before.

Resources

- WebRTC Samples
- WebRTC Concepts and Definitions
  - Note: WebRTC is also used for video/audio communication, however, this app does not make use of such endpoints.
- StackOverflow Manual WebRTC Example