Secure Peer-to-Peer Chatting on iOS

Johan Henkens and Shayan Yassami
{jhenkens,syassami} at cs.ucsb.edu

1 Objective-C Elliptic Curve Library

We created an Objective-C framework that can be built both for use as a Framework within Mac OS X apps, as well as a static library used in iOS apps. The framework uses the `BIGNUM` class within the OpenSSL[1] library. The `BIGNUM` class provides all of the necessary modular arithmetic operations, as well as cryptographically sound random number generation for large integers. Within the library, a class is exposed to initialize a NIST D.1.2.1 192-bit curve [1]. Further, we have implemented elliptic point addition using the Jacobian projective coordinate system, and point multiplication using the binary method with the canonical recoding algorithm discussed in [2]. The library allows one to easily generate a private key, request a public point, and compute a shared secret based on a private key and an public point, thus providing all the steps necessary to perform an $ECDH$ exchange.

OpenSSL by default seeds its random number generators from `/dev/random`. The library allows us to manually pass in a seed to the OpenSSL PRNG if desired, which we do in the iPhone app. We call iOS's `SecRandomCopyBytes` function[2] which gives us cryptographically strong random data to seed OpenSSLs PRNG with.

2 Symmetric Encryption

For chose to use RNCryptor [3], made by Rob Napier as our secure encryption scheme due to ease of use and methodology for encryption practices. RNCryptor is a scheme based around Advanced Encryption Standard(AES). Chosen to replace Data Encryption Standard(DES), AES provides stronger cryptography with a keylength of 128, 192, or 256 bits. For this project we chose to use the maximum keylength of 256 bits. To achieve this length given our shared secret of 192 bits, we needed to extend the length of the key by $256-192 = 64$ bits. Although a 192 bit $ECDH$ password is considered secure, chose the maximal strength due to the availability of the iPhone's A9 processor.

2.1 Password Based Key Derivation Function

To extend our shared secret to 256 bits we used the standardized Password Based Key Derivation Function v.2.($PBDKF2$) Our Implementation uses `DK = PBKDF2(HMAC?SHA1, passphrase, Salt, 10000, 256)`

Algorithm 1 PBKDF2

$DK = PBKDF2(PRF, Password, Salt, c, dkLen)$ ▷ %PRF is a pseudorandom function of two parameters with output length hLen (e.g. a keyed HMAC) %
▷ %Password is the master password from which a derived key is generated%
▷ %Salt is a cryptographic salt%
▷ %c is the number of iterations desired%
▷ %dkLen is the desired length of the derived key%
$DK = T1||T2||...||Tdklen/hlen$
$Ti = F(Password, Salt, Iterations, i)$
$F(Password, Salt, Iterations, i) = U1^U2^U3^Uc$
$U1 = PRF(Password, Salt||INT_msb(i))$
$U2 = PRF(Password, U1)$
$Uc = PRF(Password, Uc-1)$

[1] http://www.openssl.org
[2] `http://developer.apple.com/library/ios/#documentation/Security/Reference/RandomizationReference/Reference/reference.html`
[3] `https://github.com/rnapier/RNCryptor`

2.2 Advanced Encryption Standard

We chose to use AES Cipher Block Chaining(CBC) Mode. Each block of plaintext is XOR'ed with the previous ciphertext block pre-encryption. This means that each ciphertext block depends on the previous plaintext block. To initialize this process we use an Initialization Vector(IV) of length 16 bytes. Our scheme also uses encrypt-then-mac, doing encryption then message based authentication.

2.3 Parameters

```
AES_ALGORITHM = "AES/CBC/PKCS5Padding";
HMAC_ALGORITHM = "HmacSHA256";
AES_NAME = "AES";
KEY_DERIV_ALGORITHM = "PBKDF2WithHmacSHA1";
PBKDF_ITERATIONS = 10000;
VERSION = 2;
AES_256_KEY_SIZE = 256 / 8;
AES_BLOCK_SIZE = 16;
```

Table 1: Data Packet Format

Item	Byte Length	Description
Version	1	Data format version. Always 0x02.
Options	1	Options. 0x01 indicates a password was used.
Encryption Salt	8	Salt value used to derive the encryption key. Only present if a password was used.
HMAC Salt	8	Salt value used to derive the HMAC key. Only present if a password was used.
IV	16	Random IV
Ciphertext	n x 16	Encrypted with 256-bit AES, CBC-mode with PKCS #5 padding.
HMAC	32	HMAC calculated with SHA-256.

3 iOS User Interface

The interface for the app is constructed of two separate `UIViews` within a `UINavigationController`. The first view, `ConnectViewController` presents the user with a single connect button, which, when pressed displays a `GKPeerPickerController`, a class within the iOS GameKit framework used to connect to Bluetooth peers. Using the `GKPeerPickerController`, the user is able to see other phones running CS290GChat who are also currently looking for peers. Once a Bluetooth connection is established, the $ECDH$ handshake is performed, and upon completion, a segue is performed to the second view, `ChatViewController`. This view is a subclass of the `MessagesTableViewController`[4], which establishes a standard iOS chat messaging interface. The messages sent and received from this interface are encrypted using the previously mentioned methods of the RNCryptor library. The `MessagesTableViewController` is simply an ease and aesthetic wrapper around a stock iOS `UITableView`, which is used to display the messages in a table. The `ChatViewController` implements the interfaces `JSMessagesViewDelegate` and `JSMessagesViewDataSource` in order to provide the send message hook, and the table data sources to the `MessagesTableViewController`.

4 iOS Bluetooth Connectivity

As previously mentioned, the GameKit session is used to connect to Bluetooth peers using the `GKPeerPickerController`. In order to establish and communicate over a Bluetooth connection using the `GKPeerPickerController`, a class must be implement the `GKPeerPickerControllerDelegate` interface, and assigned as the delegate, which provides methods for handling newly created `GKSessions`, which are the Bluetooth sessions. Additionally, in order to control the changing of states for an existing `GKSession`, there must be a class that implements the `GKSessionDelegate`. Lastly, a class must implement the method `receieveData`[5] in order to handle receieved data from the Bluetooth connection.

`ConnectViewController` implements both of these interfaces, and the `receieveData` method. `ChatViewController` only implements the `GKSessionDelegate` interface, and the `receieveData` method. As the

[4] https://github.com/jessesquires/MessagesTableViewController

[5] http://developer.apple.com/library/ios/#documentation/GameKit/Reference/GKSession_Class/Reference/Reference.html

`ConnectViewController` is responsible for establishing new connections, it must be able to handle the creation of new sessions. Further, as the `ConnectViewController` handles the $ECDH$ key exchange before segueing to the `ChatViewController`, it must be able to handle session state changes, as well as the receiving of data from the peer during the key setup. The `ChatViewController` is only used to handle a single connection, and thus only needs to handle state changes from the corresponding `GKSession` as well a receiving data.

The iOS GameKit framework has reliability built into its Bluetooth transmission, ensuring delivery provided no error is returned. Further, the framework also provides connection timeout information to our client.

References

[1] National Institute of Standards and Technology. FIPS PUB 186-3. `http://cs.ucsb.edu/~koc/ac/docs/w03/fips_186-3.pdf`

[2] Koc, Cetin Kaya. High-Speed RSA Implementation. `http://cs.ucsb.edu/~koc/ac/docs/w01/r01rsasw.pdf`