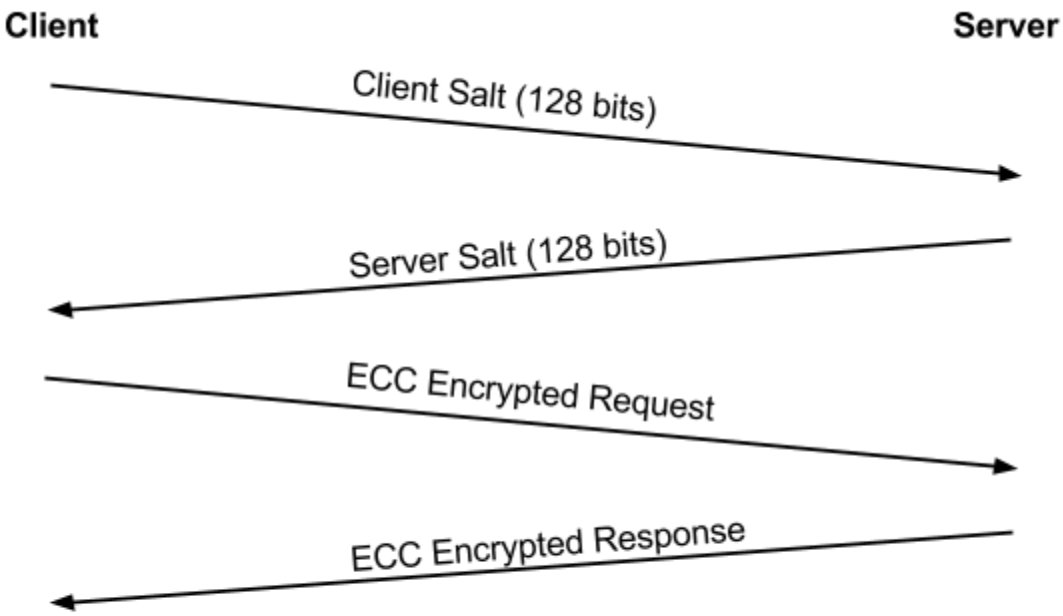
BTLE Secure Message Exchange

Version: 1.0
Date: 11/01/2013
Author: Todd A Ouska



BTLE Secure Message Exchange

Requirements

Eavesdropping on the connection doesn't reveal anything to the snooper.

Replay attacks are not possible.

Man in the middle attacks don't reveal anything to the attacker.

Fast, even on 100Mhz embedded devices, total time in the worst case on the order of 100ms or less.

Small number of roundtrips, small packet sizes, and as little CPU use as possible.

Assumptions

Each side has a secure private key and a public key signed by a trusted Certificate Authority.

Traditional Approaches

1) The simplest approach is to rely on built in BTLE security. The negatives of this approach include user interaction upon first use and replay attacks can't be ruled out.

2) The next basic approach is to simulate SSL/TLS on top of BTLE with client authentication. The negatives of this approach include 4 round trips, 16 messages, and two public key operations on each side in the worst case. Achieving the time requirement would be very difficult if not impossible, not to mention the energy/CPU use needed for each session. From a security standpoint though, it is very hard to beat.

3) The next logical approach is to encrypt and sign every message with RSA. The negatives of this approach are the 4 public key operations needed by each side (2 for creating a message and 2 for verifying one). This will be even slower than SSL even though the roundtrips and packets are greatly reduced. It also opens up a replay attack.

4) An extension of the previous approach substitutes ECC over RSA to increase performance. But 4 public key operations per exchange are still too many to achieve the performance requirement and replay attacks are still possible.

BTLE Secure Message Exchange

An interesting point of ECC encryption is that ECC encryption is never actually done. With RSA, the RSA keys are used to directly encrypt and decrypt a message because the RSA keys are big enough to do this and because RSA provides an easy way to do it. A 2048 bit RSA key is 256 bytes, meaning you can encrypt a message up to about 256 bytes. A 256 bit ECC key, on the other hand, is only 32 bytes, and there's no easy way to "encrypt" directly with ECC keys. Instead, the ECC keys are used to derive a shared secret and that shared secret is used as an encryption key to encrypt the message with AES-128-CBC for example.

Normally, with ECC encryption, one side creates a temporary ECC key and uses that temporary key together with the peer's known key to get a temporary shared secret. Because the one side is using a temporary key there is no way to authenticate the temporary side unless the message is also signed. This is exactly approach 4) above.

Each side has a trusted public key, which should be leveraged as much as possible. The only negative of doing ECC encryption with the static keys is that some random salt should be used for each session, otherwise the same encryption key is derived each time and will leak information over time. But this is exactly the same problem that needs to be solved to prevent replay attacks, i.e., some per exchange uniqueness is needed. In short, having each side exchange salt at the beginning of the session kills two birds, it allows a unique encryption key and prevents replay.

Having the salt in the clear (non secret) is perfectly fine, SSL does the same thing with ClientRandom and the ServerRandom. See HDKF and ECC references for more details. Authentication is built in, no signing is required. Plus, the shared secret can be cached for frequent peers in secure memory, meaning 0 public key operations could be the median expected cost if this optimization path is taken. The worst case is 1 public key operation for each side, and the server side can start computing the shared secret once it has completed the salt exchange.

Details of BTLE Secure Message Exchange

Definitions:

Shared Secret (***SS***) == ECC_DH (Server Public Key, Client Private Key)

likewise, from the Server's perspective

Shared Secret (***SS***) = ECC_DH (Client Public Key, Server Private Key)

yields the same ***SS***, but no one else can derive it

CSalt = Client Salt (128 bits or 16 bytes)
SSalt = Server Salt (128 bits or 16 bytes)

S1 = First 8 bytes of ***CSalt*** || First 8 bytes of ***SSalt***
S2 = Second 8 bytes of ***CSalt*** || Second 8 bytes of ***SSalt***

Where || means concatenation

m = Application message
M = ***m*** || 4 byte random number

info = "BTLE Secure Message Exchange". This creates unique keys and messages per application domain and prevents dictionary attacks.

Encryption algorithm (***e***) is AES-128-CBC because 128 bit symmetric is equivalent to 256 bit ECC and AES is considered very secure.

MAC = HMAC-SHA256 because it provides the same security level on the MAC as with everything else, ***S2*** is used as salt and appended to the encrypted message, the output is (***d***)

KDF (or key derivation function) = HKDF-SHA256 to get same security level, the output is the ***Keys*** for encryption/decryption, IVs, and MAC. HKDF uses ***S1*** as the salt and ***info*** for the app specific id.

Keying Material Length = 112 bytes. Each side needs a 16 byte encryption key, a 16 byte decryption key, an 8 byte encryption IV, an 8 byte decryption IV, a 32 byte HMAC create key, and a 32 byte HMAC verify key. The key output order is as follows:

Client Encrypt Key || Client Encrypt IV || Client HMAC create Key ||
Server Encrypt Key || Server Encrypt IV || Server HMAC create Key

The client uses the Server's Encrypt Key to decrypt the the server message while the server uses the Client Encrypt Key to decrypt the client message.

Equations:

ECC Encrypted Message = ***eM*** || ***d***

eM = AES-128-CBC(Client Encrypt Key, Client Encrypt IV, ***M***)

d = HMAC-SHA256(Client Hmac create Key, ***eM*** || ***S2***)

112 byte ***Keys*** = HKDF-SHA256(***SS, S1, info***)

Message Format:

Pending.

Attacks:

Eavesdropping: A snooper lacks either side's private key, he has no ability to decrypt the exchanged messages.

Replay: Because each side submits 128bits of salt to the exchange, a replay is only possible when an attacker sees the exact salt from each side, all 256 bits. Otherwise, different keys will be generated and a replay attempt with the wrong salt will have the wrong keys, preventing authentication. The chance of an attacker "finding" this repeat possibility is 1 in 2^{256}, or 1.16×10^{77}. The number of atoms in the entire observable universe is estimated to be about 10^{80}. Not only that, but the attacker will have to inject the replay message into the exchange before the real one is sent without detection.

Man in the middle: An attacker could pretend to be the peer and submit "bad" salt on his behalf. This wouldn't reveal anything about the message because the attacker lacks the private keys. Enough good salt is retained to prevent dictionary based attacks. The HKDF RFC recommends using at least half the length of the derived encryption key as the salt length. We're using the full length in case an attacker injects or alters some bad salt for one of the sides. In this case, the required half length of good salt is still retained.

References:

HKDF - RFC 5869 : http://tools.ietf.org/html/rfc5869
HMAC - RFC 2104 : http://www.ietf.org/rfc/rfc2104.txt
AES - FIPS PUB 197 : http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
ECC Static Key Guidance - RFC 6278 : http://tools.ietf.org/html/rfc6278