Bumblebee Technical Details

The "Two Stages of Locked Boxes" Approach

A “**Bundle**” is what I refer to as the encrypted output construction created by **Bumblebee**. I refer to the **Bumblebee** bundling approach as "Two Stages of Locked Boxes." I would imagine this analogy is not unique and is likely referred to by some other very clever and cryptographer-ey sounding name.

Nevertheless, it is the name of the analogy that I use for describing this approach. The motivation for this approach is the fact that, in our offline scenario, we must encrypt data without the availability of an active network session. As a result, there is no active exchange mechanism that can allow us to derive ephemeral, agreed upon session keys via a DH exchange mechanism.

A complex solution would require User 1 to provide a symmetric key for encrypting the data, which they would then be responsible for sharing both the data and the key with User 2, so that User 2 could decrypt the data. Instead, we use both Asymmetric and Symmetric crypto. This would be similar to SSL/TLS, but without the benefit of DH agreements, and naturally without some of the benefits as well.

We first generate a random, strong symmetric key which we do not reveal to the user directly. We use that symmetric key to encrypt the secret.

In a slightly over simplified description, we then we encrypt that key and some other elements (salt, signature, etc) using asymmetric crypto. The asymmetric crypto uses the public keys which can be shared safely. This prevents having to know and manage the sharing of the symmetric key itself.

To understand this...

You have two boxes, Box 1 and Box 2.
And there are two users, Alice and Bob.

Box 1 requires a single key to open it, which we'll call Key 1. There is only one copy of Key 1. Box 1 is as close to indestructible as possible. Once locked, it can only be opened using Key 1.

Box 2 requires two keys, key A and key B. It is locked with Key A and can only be opened with Key B. Box 2 is also as close to indestructible as possible. Once it is locked with Key A, it can only be opened using Key B.

There is possibly some unknown number of copies of Key A, from one to possibly many. They may be in the possession of friends or enemies of Alice and Bob. Regardless, none of them can open box 2 with their copies of Key A.

There is only one Key B in existence, which is in Bob's possession.

Alice takes her secret, puts it in Box 1 and locks Box 1 with Key 1.

She then takes Key 1 and places it in Box 2.

She also includes a secret note, which she signs in her own handwriting, and puts it in Box 2. The handwriting will confirm for Bob that Alice is the one who sent Box 2.

She then locks Box 2 with Key A.

Now that Box 2 is locked, neither she nor anyone else can open Box 2. Only Bob can open Box 2 with his Key B.

Alice sends both boxes to Bob using any mechanism she wishes. She can hand-deliver them, mail them, leave them out for Bob to come by and pickup, etc.

If an enemy should acquire either of the boxes, they are unable to extract the contents of either without Key B. In that regard, Alice's secret is safe.

Assuming Bob takes possession of both boxes, he uses his Key B to open Box 2.

He examines the secret note at this point, and confirms it is signed with Alice's handwriting. If it is not her handwriting, he discards Box 1 and does not open it.

Otherwise, he then removes Key 1 and opens Box 1, extracting Alice's secret.

Here's a general description of how this analogy correlates to the **Bumblebee** approach technically:

We generate a random, strengthened key and encrypt the secret data with it using symmetric crypto. This is Box 1
and Key 1 in our analogy.

We build a separate data structure called the Header. In this structure, we put various things, including the symmetric key (Key 1 in our analogy) and a data element signed with the sender's private signing key (the secret note). We encrypt this information with asymmetric crypto, using the receiver's Public Key.
This is Box 2 and Key A in our analogy.

Both are delivered to the receiver, who unlocks the asymmetric structure with their private key (Box 2, Key B).

Bumblebee uses the stored signature to affirm the sender is who we expected (the handwriting on the secret note).

If the signature is verified, **Bumblebee** then extracts the symmetric key (Key 1) and decrypts the secret data (Box 1).

This process does not require the sender or the receiver to manage any of the cryptographic elements mentioned in the process, outside of sharing public keys in some way. The sharing of public keys is a one-time process, unless they are changed in the future for some reason.

Extending "Two Stages of Locked Boxes" With Two Shipments

Now, let's extend the analogy a little bit. Perhaps, Alice is not comfortable with the security of putting Key 1 in Box 2 and providing both boxes at the same time. Basically, her concern is that transporting Key 1, which unlocks Box 1, along with Box 1 is inherently insecure, even if it is in Box 2 and that is impenetrable. It just makes her uncomfortable.

So, Alice ships the two boxes separately. She sends Box 1 via USPS and Box 2 via FedEx. Assume that they take different routes and arrive at different times. Alice feels better about this, because Key 1 is never transported in the presence of Box 1. Also, without both boxes, the secret cannot be accessed.

Once Bob has both boxes in his possession, he is able to access her secret, as previously described.

Relating this to the Bumblebee bundling process, when you bundle a secret you can choose to output the bundle to a single "combined stream" or two "split streams." This is effectively what Alice is dealing with in this scenario extension.

You can use set the "Bundle Type" by using the **--bundle-type** flag (or **-b** shortcut). The options are "combined" or "split".

With "combined", the output is emitted to a single stream. The default extension for this "*.bcomb" for "bee combined stream" format, which can be overridden by providing an explicit file name using the "**--output-file**" flag (or **-y** shortcut). When using the combined output encoding, the length of the header is emitted to the output stream, followed by the header data, which is then followed by the payload data. When opened by the receiver, the header and payload are extracted from the combined stream and processed to provide the unencrypted secret.

With "split", two separate output entities are created; one for the header, one for the payload. The header with have a default extension of "*.bhdr" and the payload will have a default extension of "*.bdata".

You may deliver the two artifacts any way you wish. When opening the bundle, you specify the split bundle type, and **Bumblebee** will process the two components accordingly to create the unencrypted output.

It is a bit more work to provide the two separate artifacts using different transport paths. But, both combined and split encodings are functionally identical.

***Note**: While the combined stream should be sufficiently secure for our needs, if you are concerned that a weakness could be exploited with the combined approach, you can choose to use split streams. Perhaps asymmetric key associations are rendered ineffective or insecure due to some emergent tech (e.g. quantum advances), then using split streams will mitigate that concern to some degree. Of course, in that event, most modern crypto systems will be rendered ineffective as well.*

*The soon coming public reveal of "post-quantum solutions" will shed more light on this. Once a post-quantum solution is accepted by the community, perhaps **Bumblebee** would be updated to use those algorithms accordingly.*

Details Of The Bundle Process Flow

The Bundle process receives an input byte sequence (secret) and outputs it as an encrypted byte sequence compromised of two parts:

- A Bundle Header that is encrypted with curve25519
- A Bundle Payload that is encrypted with Chacha20-Poly1305

The header contains various elements of details and metadata. The payload contains the secret data.

The following items are included in the header as of **Bumblebee** release 0.1.0:

```
// SymmetricKey is a random value used to encrypt the payload using
Chacha20/Poly1305
SymmetricKey []byte

// Salt is a random value provided for the payload encryption
Salt []byte

// InputSource records the source type of the data provided for bundling
InputSource      BundleInputSource

// The date the bundle was created
```

```
CreateDate       string // RFC3339

// OriginalFileName records the file name of the source file, IF the source was a
file
OriginalFileName string

// OriginalFileData records the date stamp of the source file, IF the source was a
file
OriginalFileDate string // RFC3339

// ToName indicates the name used to identity the User public keys in the keystore
ToName           string

// FromName indicates the name used to identity the keypair set that encrypted the
bundle
FromName         string

// SenderSig contains the RandomSignatureData
SenderSig        []byte

// HdrVer identifies the version of the bee functionality that built the hdr
HdrVer           string

// PayloadVer identifies the version of the bee functionality that builtthe payload
PayloadVer       string
```

When bundling the input, the header is first populated with the following values:

- SymmetricKey is set to a random 32-byte sequence
- Salt is set to a random 32-byte sequence
- The SenderSig is initialized with a random 32-byte sequence that is signed using ed25519 and the Sender's Private Key from their ed25519 (signing) keypair. Both the random sequence and the signature output are stored in the header.
- All the remaining metadata values are populated as needed

The header itself is then encrypted using the NKEYS XKEYs (curve25519) SEAL functionality. The SEAL functionality uses the receiver's Public Key from their curve25519 keypair.

The header is then emitted to the output stream.

Once the encrypted header is written to the stream, then the payload is encrypted using the previously derived Salt and SymmetricKey. While the random SymmetricKey is potentially a strong one-time sequence, it is still strengthened using Argon2. This is to mitigate any weak random sequences that might be generated.

The payload is encrypted using XChacha20-Poly1305, which is a streaming cipher and supports the output of large payload streams. The output encryption is performed in sealed chunks of 32,000 bytes. Each chunk will result in a small increase in output size, due to nonce and AEAD overhead, so the resulting output stream will be slightly larger than the input stream.

A technical flow of the Bundle process...

Let $Key_{payload}$ be a random symmetric key for the payload data stream
Let $Key_{derived}$ be an Argon2 permutation of $Key_{payload}$

Let $Salt_{payload}$ be a random 32-byte salt for the payload data stream
Let $PUB_{receiver}$ be the curve25519 public key for the receiver
Let $PK_{sign-sender}$ be the ed25519 private signing key for the sender
Let Salt_{sign} represent the random sequence for signing the header
Let Signature represent the signed sequence stored in the header
Let Header_{plain} represent a bundle header structure as described above
Let Header_{encrypted} represent the encrypted bundle header
Let Cipher be an initialized XChacha20-Poly1305 encryptor
Let ADConst be a value for Cipher's Associated Data
Let Secret_{input} represent the provided secret to encrypt
Let Secret_{encrypted} represent the encrypted form of the secret
Salt_{payload} <= Rand[32]
Key_{payload} <= Rand[32]
Header_{plain} <= New**[Key_{payload}, Salt_{payload}]**
Header_{plain}::Salt_{sign} <= Rand[32]
Header_{plain}::Signature <= ed25519.Sign[Salt_{sign}, PK_{sign-sender}]
Header_{plain}::[Metadata] <= Values[Metadata]
Header_{encrypted} <= curve25519[Bundle_{plain}, PUB_{receiver}]
WriteToStream[int16uWithFixedEndian[length(Header_{encrypted})]]
WriteToStream**[Header_{encrypted}]**
Key_{derived} <= Argon2[Key_{payload}, Salt_{payload}, time/mem/threads]
Cipher <= XChacha20-Poly1305::Init**[Key_{derived}]**
Iterate Cipher::[Secret_{input}]
   Cipher::Encrypt[Secret_{input}[blockX], ADConst] => Secret_{encrypted}[blockX]
   WriteToStream[Secret_{encrypted}[blockX]]