Bumblebee Quick Start Guide

Last updated: Nov 26, 2023

This document will provide a set of steps that will guide you through installing Bumblebee, sharing keys with another user, and using a few methods for sharing secrets with them.

This document is focused on simple use cases that will show you some basic functionality in Bumblebee. While there are a number of additional features and options in Bumblebee, this document will not go into those details.

These steps were tested on Debian arm64. However, they will work fine on a Windows or Mac as well.

Step 1. Installing Bumblebee

Option A: Download runtime from Github repository
Bumblebee is a single runtime. You can get the latest, pre-built version for your platform in the "Releases" section at https://github.com/thoughtrealm/bumblebee. Simply download and place the runtime in a common path in your OS. You can place it in a directory and just execute it directly from there, but that can result in command lines that are longer than necessary. It is recommended to place the runtime in a common path.

Option B: Build and install using the Go compiler
If you have the Go compiler installed, you can clone the repo, then simply run "make install" in the root path of the repo.

If you are on Windows and do not have the *make* utility installed, you can run "go install" instead. This build should work fine, with the one exception that the output of the "bee version" command will not be fully populated with build times.

Once installed, you can verify it is running correctly by simply typing…

```
bee
```

That will output the root help info. You can also run this to see the version info…

```
bee version
```

2. Initialize the Bumblebee environment
The first step is to initialize the Bumblebee environment. This will create the default profile, populate the initial random key sets and some other artifacts that are required that are required for sharing secrets.

To do so, just run…

```
bee init
```

You will be asked about several options.

When asked, "*Enter a default sender key name or leave empty for none*", provide a name you wish to use for the default sender account in this profile. It could be a name, a handle, an email address, whatever you wish to use for identifying yourself. The other user will be able to use whatever name they wish to use in their user store for your identity. Bumblebee will always validate the sending identity, regardless of any name assignment.

However, in a formal environment, like in a corporate environment, it is recommended to use something unique like your email address or an LDAP account name, etc.

Otherwise, for the other questions, just press enter for each to accept the default options for now.

Once the initialization is complete you can view the default profile identities by running…

```
bee list keypairs
```

That will show you the public keys only for the default and system key pairs.

You can use the "**--show-all**" flag to see the seed and private keys as well…

```
bee list keypairs --show-all
```

Of course, be aware that the you must never share your private keys with anyone. By default, they are not printed out when listing the key pairs unless you provide the "**--show-all**" flag.

> ***Note**: Bumblebee makes use of curve25519 key pair cryptography. Specifically, it uses the **NKEYS** repo/packages (https://github.com/nats-io/nkeys). **NKEYS** is provided by the **NATS** messaging server (https://nats.io/).*
>
> ***Note**: Each identity is configured with two key pairs: a **Cipher** and a **Signing** key pair. The **Cipher** key pair is a curve25519 key pair construction and is used for the encrypting and decrypting processes. The **Signing** key pair is an ed25519 key pair and is used for signing secrets sent by that identity, so that the receiving user can validate the sender's identity. The curve25519 support is found in the **XKEYS** package of the **NKEYs** repo.*

You can also see the users that you have setup by running…

```
bee list users
```

Of course, at this point, you will find that your user list is empty. You must add or import users to your local profile(s). We will do this in this Quick Start Guide.

Step 3. Export your keys in order to share them with another user

To share secrets with another user, you must provide them with your public keys. This can be done easily by exporting your keys. There are several ways to do this, but we will just focus on exporting them to a file.

To do this, run the following command. For **<username>**, use the name you provided in Step 2, when you initialized the environment. And when prompted for a password, just press return to not provide a password for now.

```
bee export user <username> --from-keypair --output-file export-user.txt
```

The export will have generated the file **`export-user.txt`**. If you dump the file contents using **`cat export-user.txt`**, or **`type export-user-txt`** on Windows, you will see something formatted similar to the following, but with different values.

```
:start  :export-user  :hex ====================================
0086a44e616d65af7573657240646f6d61696e2e636f6da84461746154797065
01aa4369706865725365656 4c0ab5369676e696e6753656564c0ac4369706865
725075624b6579d938584349354e4f5a4649474c5a58334f4156425355515850
32594d324445564d57574c474c4c4654524553594758575058414b3758425457
4dad5369676e696e675075624b6579d9385544335057353552 5a585341355150
344533424d535355414f594e494e4d564c4c414b4d5a525a4a36465950565657
575144374144435344
:end ============================================================
```

Bumblebee uses that format because it is text safe. Meaning, you can copy it and paste it into a text or slack post, email body, etc. It is simply the hex encoded sequence of the file's binary contents. Bumblebee uses this encoding format for several different features.

If you wish, when exporting the file you may provide a password. If you do so, the contents will be symmetrically encrypted with XChacha20-poly1305 using Argon2 for key derivation. This is a strong crypto encryption.

While the export data only contains public keys, you may wish to protect those by encrypting them with a password. If you do so, keep in mind that you just provide the password to the user that is importing your export file. They will need the password in order to open and import the info.

Alternately, if you run the following it will output the data to the console instead of a file. This time, enter a password when prompted for one.

```
bee export user <username> --from-keypair --output-target console
```

With the output in the console, you can copy it and paste it to the other user. Perhaps paste it in their Slack channel. And in this case, provide them with the password in some way. If you provide it through some public transport outside of your trusted network, you will probably want to use use a password to protect your public keys. However, that is possibly optional, depending on your specific use case.

Step 4. Import the other user's keys in order to add them to your user store

After you supply your exported keys to another user, you will want to import their keys, so you can send them secrets.

To demonstrate this process, we will import your own keys as a user with a different name.

To so, run the following command from the same directory as your export file.

Because the export file was not exported with a password, you will not be prompted to enter a password.

When it asks you what name you want to use to import the user info, just press return to use the exported name. This will be your own name. However, because your identity is actually stored as a key pair set in the key pair store, it will not be a conflict to have a user with the same name in the user store.

```
bee import --input-file export-user.txt
```

Now, lists your users again.

```
bee list users
```

You should now see a user list with your name.

At this point, you can share files and secret items with that user.

Step 5. Send a file to the other user
We will now send a file to another other user.

To make this step simply, copy any file you wish into the same directory. We will refer to the file with a name of "**`testfile.txt`**". You may change the name of your file so that the commands are exactly as provided below, or you may leave your file named whatever it is and just substitute the correct name in the commands you will be entering.

To encrypt a file for another user, we use the **`bundle`** command. We supply the "**`--to`**" flag to tell Bumblebee who the receiving user is so that it knows which keys to use for encrypting the bundle header.

```
bee bundle --input-file testfile.txt --to <username>
```

> ***Note**: If you omit the "**--from**" flag, Bumblebee will use the default key pair identity as the sender. It is possible to have multiple local identities with Bumblebee, as well as multiple profiles, which are basically separate security contexts; but, we will not go into that functionality here. Just know that by omitting the "**--from**" flag, Bumblebee is signing the bundle using the key pair named "**default**". You can refer to other docs for further info on multiple identities in the profile, as well as multiple profiles.*

That will have bundled **`testfile.txt`** into a new file, **`testfile.bcomb`**.

> ***Note**: The **`*.bcomb`** extension refers to the "Bumblebee combined" bundle format. We will not go into the concept of bundles types for now. Just know that the **combined** format means that the two parts of a **bundle**, the **header** and the **payload**, are contained in the same file (or stream). You can refer to other Bumblebee docs for an explanation of bundle formats.*

Now, you can provide the bundled file to the other user. Keep in mind that only the user specified with the "**`--to`**" flag can decrypt the bundle, since they have corresponding private key to the public key in your user store. Not even you can decrypt the new bundle.

Of course, in this case, you are the other user. Otherwise, you could send this file using whatever mechanism you wish. You could attach it to an email, Slack it to them, etc.

Step 6. Decrypt a bundled file from another user
For now, we will just decrypt the bundle using the same user name, but the process is identical.

We can decrypt the bundle to a few different target outputs. For now, we will decrypt the output to a file. If the input source of the bundle was a file, then Bumblebee will include the original file name in the bundle header. When decrypting that bundle to a file, Bumblebee will name the new, decrypted file with the same name as the original file.

To demonstrate this, rename the current test **`testfile.txt`** to something like **`testfile.original.txt`**.

Now, we use the **`open`** command to decrypt the bundle.

```
bee open --input-file testfile.bcomb –from <username>
```

> ***Note****: Similar to the bundle command, in this case, if you omit the "**--to**" flag, Bumblebee will assume that the default key pair identity is the receiving key pair. The receiver key pair's private key is used to decrypt the bundle. Again, it is possible to have multiple local identities with Bumblebee, as well as multiple profiles, which are basically separate security contexts; but, we will not go into that functionality here. Just know that by omitting the "**--to**" flag, Bumblebee is decrypting the bundle using the local key pair named "**default**". You can refer to other docs for further info on multiple identities in the profile, as well as multiple profiles.*

The **`--from`** flag tells Bumblebee what signing keys to use to validate the sender's identity. When you import the other user's public keys, you import their signing public key as well. When opening a bundle, their public signing key is used to validate that they are the one who signed the bundle internally. Bee does the for you and is why you must supply a **`--from`** reference. If the user's public key referenced by the "**`--from`**" flag does not validate correctly with the internally signed structures, then Bumblebee will output an error and will abort decrypting the bundle.

You should now see a new file with the same name as the original file, **`testfile.txt`**. You can compare this file to the original file that is now named **`testfile.original.txt`** using whatever process or compare command or tool you want to use. The two files should be identical.

What we just walked through is the general pattern for bundling and sharing any files with any user.

1. Simply run the **`bundle`** command and create the encrypted bundle for a target receiving user.
2. Supply the encrypted bundle to the intended user.
3. That user decrypts the bundle with their private keys using the open command.
4. Bumblebee validates that your identity was the sending user when it opens the bundle.

Step 7. Share secrets that are not files
It is possible to send secrets that are not originally stored in a file. There are a few ways to do this. We will focus here on directly entering secrets from the console. You can consult the other docs for doing so in other ways.

In this bundle we are going to enter the secret by typing it in. We are also going to write it out to the console, where we will copy and paste it to some system for the other user. They will use the pasted text to decrypt your secret and write it to their console.

In so doing, we will share a secret that was never **explicitly** written to a file. By explicitly, we acknowledge that all operating systems may use files for temporary memory storage any time. So, you may be inadvertently writing data to a file when you don't mean to.

To bundle a secret directly from the console, use the "**`--input-source console`**" flag value.

```
bee bundle --input-source console --to <username>
```

This will provide a prompt to enter text. You enter text, line by line. You stop entering text by pressing return on an empty line, basically entering an empty line.

Once you complete entering text, Bumblebee will use that as the bundle input.

Because we did not provide any output directives, in this case, Bumblebee will default the output to the console as well. Similar to the export output, you should an output like this, but with different values than this example.

```
Starting BUNDLE request...
:start :header+data ============================================
01b1786b763147f42dfe84957afa232848c3d8ee03d299fce66a89a6613f1f9d
5c7507e4606990e713611a6603741fce9a823f096ae3d1cb01dec0d938016279
ee8b1042227315fc569c5063d435a9ffd9c9e70f504275dff87e1ba3eb318e75
20a901dbf42be47d5e7af420065c60db0b862fdb6129c7158f345624162b70ed
1a443ad063858aa4df49fc2895390bb971ea2ac74a77673252abf7a4e7e9d85b
3b07148ec504eff025dd1602ae49b5f5a00b785ba0e2b00b4a505615222c5da3
e3c0af302f7dceb89e9587b02511a0fbb64e1352ca480a24e9482e016615dfc0
4cdcb5d87533250eed4243c05aacd103c14c6ea9751b33d16f9031630635fba2
111d29b56efe7b6a2cd239bfd7647992b3681f748c1d7310ba911fa3174a7e50
14e7a083952070481be96dddfd940ef7a58b36d0f4d1710d2644d8852236660f
c3d8f1c938befad058eafd7653736fa21cb26cbfc0ba4eb29b4926307c203724
b0b7f86c64d279073b174d0aaa255edf006ac27883b1519284f326470168738e
1062b68b794e73bb695fa4ca51b046ce0d260f4f3b98ed0bc204cab30eec683b
c8e539f1ddd4168c47b24c460e49ba982b3294c7ba1c1285efc07bfb04e3a56e
c32c40d461f65a5b1f0ce7c21175d713d654942d1a153bd1cf29c59f2c3892ec
7aac46eea55b9d8a519938d7a6
:end ==========================================================
BUNDLE completed. Bytes written: 493 in 76 milliseconds.
```

That is a binary safe version of the bundle data.