**Auth Header Signing & Verification in Node.js**

## **Libraries used :**

[libsodium-wrappers](https://www.npmjs.com/package/libsodium-wrappers)

Importing libraries :

import \_sodium, { base64\_variants } from 'libsodium-wrappers';

import \_ from 'lodash'

## **Creating Authorization header :**

As per the Beckn specifications the **Authorization header** string format should be as below :

(created): 1402170695

(expires): 1402170995

digest: BLAKE-512=X48E9qOokqqrvdts8nOJRJN3OWDUoyWxBf7kbu9DBPE=

This function is an example to create a header as above :

export const createSigningString = async (message: string, created?: string, expires?: string) => {

if (!created) created = Math.floor(new Date().getTime() / 1000).toString();

if (!expires) expires = (parseInt(created) + (1 \* 60 \* 60)).toString(); //Add required time to create expired

await \_sodium.ready;

const sodium = \_sodium;

const digest = sodium.crypto\_generichash(64, sodium.from\_string(message));

const digest\_base64 = sodium.to\_base64(digest, base64\_variants.ORIGINAL);

const signing\_string =

`(created): ${created}

(expires): ${expires}

digest: BLAKE-512=${digest\_base64}`

console.log(signing\_string);

return { signing\_string, expires, created }

}

It does the following :

1. Generate the **digest** of the request body using the **BLAKE-512** hashing function. The json request body should be stringified before hashing.
2. Generate the **created** field. The `created` field expresses when the signature was created. The value MUST be a Unix timestamp integer value. A signature with a `created` timestamp value that is in the future MUST NOT be processed.
3. Generate the **expires** field. The `expires` field expresses when the signature ceases to be valid. The value MUST be a Unix timestamp integer value. A signature with an `expires` timestamp value that is in the past MUST NOT be processed.
4. Concatenate the three values, i.e the `created`, `expires` and `digest` in the format as shown above. The string is the **signing string** which the BAP is going to use to sign the request
5. Every line in the signing string must end with a new line. All 3 components **digest**, **created** and **expires** should be in their own lines.

## **Generating public and private keys :**

We can create the public and private key pairs using the libsodium-wrappers library as shown below :

export const createKeyPair = async () => {

let {publicKey , privateKey} = sodium.crypto\_sign\_keypair();

const publicKey\_base64 = sodium.to\_base64(publicKey, base64\_variants.ORIGINAL);

const privateKey\_base64 = sodium.to\_base64(privateKey, base64\_variants.ORIGINAL);

return { publicKey : publicKey\_base64, privateKey : privateKey\_base64 };

}

## **Signing a message using private key :**

The private key obtained above can be used to sign a message using the below function :

export const signMessage = async (signing\_string: string, privateKey: string) => {

await \_sodium.ready;

const sodium = \_sodium;

const signedMessage = sodium.crypto\_sign\_detached(signing\_string, sodium.from\_base64(privateKey, base64\_variants.ORIGINAL));

return sodium.to\_base64(signedMessage, base64\_variants.ORIGINAL);

}

## **Verifying a message using public key :**

Similarly a message that was signed can be verified using the corresponding public key with the below function :

export const verifyMessage = async (signedString: string, signingString: string, publicKey: string) => {

try {

await \_sodium.ready;

const sodium = \_sodium;

return sodium.crypto\_sign\_verify\_detached(sodium.from\_base64(signedString, base64\_variants.ORIGINAL), signingString, sodium.from\_base64(publicKey, base64\_variants.ORIGINAL));

} catch (error) {

return false

}

}

## **Creating authorization header**

The authorization header should be created in the below formats :

For BAP/BPP

Authorization : Signature keyId="example-bap.com|bap1234|xed25519",algorithm="xed25519",created="1606970629" expires="1607030629",headers="(created) (expires) digest",signature="Base64(XEd25519(BLAKE-512(signing string)))"

For BG

Proxy-Authorization : Signature keyId="example-bg.com|bg3456|xed25519",algorithm="xed25519",created="1402170695",expires="1402170995" headers="(created) (expires) digest",signature="Base64(XEd25519(BLAKE-512(signing string)))"

export const createAuthorizationHeader = async (message: any) => {

const { signing\_string, expires, created } = await createSigningString(JSON.stringify(message));

const signature = await signMessage(signing\_string, process.env.sign\_private\_key || "");

const subscriber\_id = config.bap\_id;

const header = `Signature keyId="${subscriber\_id}|${config.unique\_key\_id}|ed25519",algorithm="ed25519",created="${created}",expires="${expires}",headers="(created) (expires) digest",signature="${signature}"`

return header;

}

The above function creates the authorization header as per the format. This needs to be sent as **Authorization** or **Proxy-Authorization** header as per formats given above. Message is the body of the request.

## **Verifying authorization header**

For every request received the authorization header should be verified using the sender’s public key.

export const verifyHeader = async (headerParts: any, body: any, public\_key: string) => {

const { signing\_string } = await createSigningString(JSON.stringify(body), headerParts['created'], headerParts['expires']);

console.log("recreated signing string:");

console.log(signing\_string);

const verified = await verifyMessage(headerParts['signature'], signing\_string, public\_key);

return verified;

}

## **Helper functions**

## **Signature verification entry point**

Function that initiates the signature verification

export const isSignatureValid = async (header: any, body: any) => {

try{

const headerParts = split\_auth\_header(header);

const keyIdSplit = headerParts['keyId'].split('|')

const subscriber\_id = keyIdSplit[0]

const keyId = keyIdSplit[1]

const public\_key = await lookupRegistry(subscriber\_id, keyId)

const isValid = await verifyHeader(headerParts, body, public\_key)

return isValid

} catch(e){

console.log('Error', e)

return false

}

}

Functions that lookup the registry and fetches the matching providers (subscribers)

export const lookupRegistry = async (subscriber\_id: any, unique\_key\_id: any ) => {

try {

const reqBody = {

"type":"BPP",

"domain": config.domain,

"subscriber\_id": subscriber\_id

}

const response = await axios.post(`${process.env.REGISTRY}/lookup`, reqBody);

if (!response.data) {

return false

}

const public\_key = await getProviderPublicKey(response.data, unique\_key\_id)

if (!public\_key) {

return false

}

return public\_key

} catch(e){

return false

}

}

Functions that fetches the public key for the matching ukId

export const getProviderPublicKey = async (providers: any, keyId: any) => {

try {

const provider = \_.find(providers, ['ukId', keyId])

return provider?.signing\_public\_key || false

} catch(e){

return false

}

}

function to split authorization header used above :

const split\_auth\_header = (auth\_header: string) => {

const header = auth\_header.replace('Signature ', '');

let re = /\s\*([^=]+)=([^,]+)[,]?/g;

let m;

let parts: any = {}

while ((m = re.exec(header)) !== null) {

if (m) {

parts[m[1]] = remove\_quotes(m[2]);

}

}

return parts;

}

Reference :

[Signing Beckn APIs in HTTP](https://sites.google.com/beckn.org/signing-beckn-apis-draft03/home?authuser=1)