analysis

August 13, 2024

1 Update Payload Analysis

The update payload is the data that describes an integrity assured, authorized update to a specific DID document. The payload is NOT published on chain, only a hash of the payload is published by one or more beacons. During resolution, a resolver MUST retrieve the update payload that matches the hash for the beacon signal they are processing. They either retrieve this from a CAS (e.g. IPFS) or in a sidecar manner from the DID controller. In either instances, the DID controller SHOULD keep a record of all DID updates across time for the DIDs that they care about.

At a minimum it MUST contain:

1. A representation of the update
2. A signature over this update

The signature will always be a constant size and the update will vary depending on the changes being made to the DID document.

Note: There is a discussion to be had about how we represent an update. In this notebook I will look at JSON patch and JSON0, we could probably also define our own custom thingy for this but we do not advise it.

2 Example Data

2.1 JSON0 or JSON Patch

Both JSON0 and JSON Patch provide a data model to define generalized operations that transform JSON. The below cell shows both formats describing how to mutate a DID document to add a service endpoint. JSON patch is marginally larger (~19 bytes in the below example). We should compare across a few more typical updates for a more complete size analysis.

However, regardless of this size analysis I am advocating that that we use JSON patch for a number of reasons:

- JSON Patch is an RFC specified at IETF - https://datatracker.ietf.org/doc/html/rfc6902 (JSON0 is defined in a repo and seems primarily adopted within JavaScript stacks)
- JSON Patch has multiple implementations: JS, Python and Rust. I only know of a JS implementation of JSON0.
- JSON Patch been used with JSONLD, including a defined @context file that describes the JSON Patch operations. This is work we would have to do ourselves if we wanted to use JSON0.

[1]:
```
import sys
import copy
import json

json_0_update_repr = {'p': ['service', 4],
 'li': {'id': '#linked-domain',
  'type': 'LinkedDomains',
  'serviceEndpoint': 'https://contact-me.com'}}

json_patch_update_repr = [{'op': 'add',
  'path': '/service/4',
  'value': {'id': '#linked-domain',
   'type': 'LinkedDomains',
   'serviceEndpoint': 'https://contact-me.com'}}]



json0_size = sys.getsizeof(json.dumps(json_0_update_repr))

json_patch_size = sys.getsizeof(json.dumps(json_patch_update_repr))



## This is a Ecdsa Secp2561 signature encoded use base58
signature =␣
 ↪'z381yXYmxU8NudZ4HXY56DfMN6zfD8syvWcRXzT9xD9uYoQToo8QsXD7ahM3gXTzuay5WJbqTswt2BKaGWYn2hHhVF

sig_size = sys.getsizeof(signature)
total_size = json_patch_size + sig_size

print(f"JSON0 Update size : {json0_size}\nJSON Patch Update size:␣
 ↪{json_patch_size}")
print("Signature size: ", sig_size)
print("Rough total size (w/ JSON patch) : ", total_size)
```

```
JSON0 Update size : 164
JSON Patch Update size: 183
Signature size:  138
Rough total size (w/ JSON patch) :  321
```

3 1. A Minimal Update Payload

So, the smallest update payload we could define would be JUST these two fields. Infact, if we really care about the bytes we could follow Bitcoin and create some custom encoding format (a la https://en.bitcoin.it/wiki/Transaction). That would be it, maybe some minial flag to indicate the end of the update payload and start of the signature.

Byte requirements: 1-4 additional bytes

Everything else about how to interpret, parse and verify this update payload would be left to the spec. This includes:

1. Which keys/verificationMethods are authorized to sign this update?
2. Which key/verificationMethod did produce the signature?
3. How the data being signed over canonicalized?
4. Which hash function should be used to hash the canonicalized data?
5. How is the signature serialized?

3.1 Limitations

- Everything is custom. For people to support this DID method they have to read, understand and implement the spec in its entirety. There is no overlap with existing DID method implementations and data securing mechanisms (e.g. Data Integrity).
- No obvious link between the DID controller and the signature. How exactly to define which keys/vms are authorized and how to indicate which key was used to sign the update is not clear. It likely requires additional data to be added to this representation.

3.2 Conclusion

We want to use Data Integrity to secure a JSON representation of the update payload. **I think we are aligned with this.**

3.3 Why Data Integrity

Data Integrity describes a standardized way to add a proof to a JSON document such that the canonicalization mechanism, hash function and signature scheme can all be interpretted by inspecting at the proof object.

Note: I have not put together an example of this minima payload as it is very dependent on the encoding serialization format.

4 2. Securing a Minimal JSON Update Payload

If we want to use Data Integrity, we do not necessarily have to use JSON-LD. It is possible to add a Data Integrity proof to a standard JSON object (see Example 2).

So perhaps the next minimal update payload we could define would be a JSON with a single property, say `patch` for the update representation with a proof added.

Note that by using a Data Integrity `proof`, the following fields are required:

- type
- cryptosuite
- proofPurpose
- proofValue

Additionally, including the `verificationMethod` property is optional but it gives a way to specify which public key created the signature. We mayp want to require in our specification that this public key is a public key controlled by the DID being updated.

See example below

4.1 Example

[3]:
```
minimal_json_update_payload = {
    'patch': json_patch_update_repr,
    'proof': {
        "type": "DataIntegrityProof",
        "cryptosuite": "secp-schnorr-2024",
        "verificationMethod": "did:btc:
 ↪6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis#initialKey",
        "proofPurpose": "assertionMethod",
        "proofValue":␣
 ↪"z381yXYmxU8NudZ4HXY56DfMN6zfD8syvWcRXzT9xD9uYoQToo8QsXD7ahM3gXTzuay5WJbqTswt2BKaGWYn2hHhVF
  }
}


minimal_json_update_payload
```

[3]:
```
{'patch': [{'op': 'add',
   'path': '/service/4',
   'value': {'id': '#linked-domain',
    'type': 'LinkedDomains',
    'serviceEndpoint': 'https://contact-me.com'}}],
 'proof': {'type': 'DataIntegrityProof',
  'cryptosuite': 'secp-schnorr-2024',
  'verificationMethod':
'did:btc:6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis#initialKey',
  'proofPurpose': 'assertionMethod',
  'proofValue': 'z381yXYmxU8NudZ4HXY56DfMN6zfD8syvWcRXzT9xD9uYoQToo8QsXD7ahM3gXT
zuay5WJbqTswt2BKaGWYn2hHhVFKJLXaDz'}}
```

4.2 Comments

I actually don't think this is terrible. It perhaps is the minimal thing that could work.

A couple of things to note.

- We are not signing JSON-LD here. This means we don't need a `@context` property. It also means the RDF canonicalization will not be possible, our cryptosuite would have to specify JCS canonicalization. We may need to define an additional cryptosuite for signing JSONLD VCs if we want to support RDF canonicalization.
- The DID controller would identify the key in their current DID document that they used to create this proof using the `verificationMethod` property. It would be up to the resolver to check that this actually is a key controlled by the DID controller in the latest DID document at that point in the resolution chain. The proof would verify with any valid `verificationMethod`.

4.3 Limitations

- This is NOT an authorization object. No clear authorization can be understood from the payload itself.
- The spec is still defining a lot of custom things out of band. How to interpret the verificationMethod and check that it is authorized to sign off on this update being the main one. But also, what `patch` means in this context.
- There is no JSONLD context. The context file provides a human readable description for each of the properties, so it would give a mechanism to define `patch`. However, I actually am not convinced this is an issue. The spec can just say `patch` is a JSON patch and refer to the IETF RFC https://datatracker.ietf.org/doc/html/rfc6902/.
- No ability to support delegation. Who is authorized to sign an update would be fixed by the spec, which would have to define how this authorization is bound to the DID controller.
- Without a `@context` property, the Data Integrity proof must be recognised and understood in some out of band way. Wheras the `@context` property would point to the specification against which the proof can be understood. Implementors would be able to better evaluate if they support this proof type.

4.4 Conclusion

Maybe we should have the `@context` property?

5 3. Securing a Minimal JSONLD Update Payload

If we add the `@context` property, our cryptosuite could now use RDF canonicalization if we want to. If we go down that route, we must also add an additional context that defines our `patch` property and all the nested fields it might have. Fortunately, Digital Bazaar have already done this work for us and the @context for JSON patch can be found here - https://w3id.org/json-ld-patch/v1.

Note: this has much the same limitations as 2. in respect to the spec having to define how authorizations are bound to the DID controller. See above.

[8]:
```
minimal_jsonld_update_payload = {
    "@context": ["https://w3id.org/security/data-integrity/v2", "https://w3id.
 ↪org/json-ld-patch/v1"],
    "patch": json_patch_update_repr,
    "proof": {
        "type": "DataIntegrityProof",
        "cryptosuite": "secp-schnorr-2024",
        "verificationMethod": "did:btc:
 ↪6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis#initialKey",
        "proofPurpose": "assertionMethod",
        "proofValue": signature
  }
}

minimal_jsonld_update_payload
```

```
[8]: {'@context': ['https://w3id.org/security/data-integrity/v2',
       'https://w3id.org/json-ld-patch/v1'],
      'patch': [{'op': 'add',
        'path': '/service/4',
        'value': {'id': '#linked-domain',
         'type': 'LinkedDomains',
         'serviceEndpoint': 'https://contact-me.com'}}],
      'proof': {'type': 'DataIntegrityProof',
       'cryptosuite': 'secp-schnorr-2024',
       'verificationMethod':
     'did:btc:6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis#initialKey',
       'proofPurpose': 'assertionMethod',
       'proofValue': 'z381yXYmxU8NudZ4HXY56DfMN6zfD8syvWcRXzT9xD9uYoQToo8QsXD7ahM3gXT
     zuay5WJbqTswt2BKaGWYn2hHhVFKJLXaDz'}}
```

6 4. Using Capabilities

All of the above examples have NOT been authorization objects. I.e. the data objects by themselves do not provide any indication of their sematics. What they mean and how they should be interpretted are all dependent on the protocol.

Capabilities are a way to define these semantics. They provide a consistent way to define where these capabilities are targetting, such that the updates cannot be taken out of context and misrepresented. This update payload is an invocation of a capability, the capability to update a specific DID document. The previous examples (1,2 and 3) are not authorization objects, they can only be understood within the context of the protocol. However, these data objects can and will be taken out of context. If we wanted to make the previous examples authorization objects we would have to add additional properties to the JSON object that define the target of the capability and a mechanism to identify who is authorized to invoke the capability.

OR, we could use a standard capabilities data format (ZCaps-LD) that has already thought through these problems AND optionally supports the ability for the delegation and scoping of capabilites. For example, it MAY be possible to delegate the ability for a key to ONLY update serviceEndpoints. Note: supporting this is not trivial, but ZCaps-LD gives us a coherent way to support these features if we want to.

6.1 Why ZCaps-LD

If we have decided to use JSON as a data format AND we want to secure this data format using Data Integrity proofs then using ZCaps-LD makes sense. It is a JSONLD (which is JSON) data format specification that provides a coherent mechanism to define capabilities that are secured with Data Integrity proofs.

6.2 Understanding ZCaps

The ZCap spec defines a root capability as a JSON object including the following properties:

- `@context` value of “https://w3id.org/zcap/v1”

- `id` - A root zcap MUST have an id that is a string that expresses a URN. This ID can always be dereferenced by the verifier system if it is a valid root zcap for a particular endpoint.
- `invocationTarget` - A root zcap MUST have an invocationTarget that is a string that expresses a URI. The invocation target identifies where the zcap may be invoked, and identifies the target object for which the root zcap expresses authority. ‘
- `controller`

Now, for our use case the verifier system the *can always dereference the ID of the root capability* needs to be anyone who is resolving the DID. So our spec currently defines a deterministic algorithm to go from a DID to the root capability for updating that DID - see Section 3.3.1 of the spec. The key point is that anyone with a DID, can generate the root capability for updating that DID.

Important: THIS DOES NOT GO IN THE UPDATE PAYLOAD

6.3 What properties do go in the update payload?

> When invoking using a DI proof, a capability invocation proof must be attached to a document that is acceptable by the API, as defined by the specific API being accessed. The capability invocation proof MUST include the intended `invocationTarget`, the root zcap ID in the `capability` property, and the action to be taken in the `capabilityAction` property. The same controller rules apply as in the HTTP signature case. https://w3c-ccg.github.io/zcap-spec/#invoking-root-capability

The following fields would need to be addedto the Data Integrity proof

- `capability` - The id of the root capability e.g. `urn:zcap:root:did%3Abtc%3Az6MkuUCMtGc31Ez1dG19PL8S4X`
- `capabilityAction` - E.g. Write
- `invocationTarget` - The DID being updated e.g. `did:btc1::z6MkuUCMtGc31Ez1dG19PL8S4XHEfwBxZuWGBcF`

Additionally, from the spec `An invocation SHOULD have an id (which may also serve as a nonce). Any other properties are considered arguments to the invocation.` So maybe we want one of those?

Lets look at an example

[4]:
```
zcaps_invocation_update_payload = {
    "@context": ["https://w3id.org/zcap/v1", "https://w3id.org/security/
↪data-integrity/v2", " "],
    "patch": json_patch_update_repr,
    "proof": {
        "type": "DataIntegrityProof",
        "cryptosuite": "secp-schnorr-2024",
        "verificationMethod": "did:btc:
↪6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis#initialKey",
        "invocationTarget": "did:btc:
↪6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis",
        "capability": "urn:zcap:root:
↪did%3Abtc%3A6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis",
        "capabilityAction": "Write",
        "proofPurpose": "assertionMethod",
        "proofValue": signature
```

```
  }
}

zcaps_invocation_update_payload
```

[4]:
```
{'@context': ['https://w3id.org/zcap/v1',
  'https://w3id.org/security/data-integrity/v2',
  ' '],
 'patch': [{'op': 'add',
   'path': '/service/4',
   'value': {'id': '#linked-domain',
    'type': 'LinkedDomains',
    'serviceEndpoint': 'https://contact-me.com'}}],
 'proof': {'type': 'DataIntegrityProof',
  'cryptosuite': 'secp-schnorr-2024',
  'verificationMethod':
'did:btc:6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis#initialKey',
  'invocationTarget': 'did:btc:6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis',
  'capability':
'urn:zcap:root:did%3Abtc%3A6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis',
  'capabilityAction': 'Write',
  'proofPurpose': 'assertionMethod',
  'proofValue': 'z381yXYmxU8NudZ4HXY56DfMN6zfD8syvWcRXzT9xD9uYoQToo8QsXD7ahM3gXT
zuay5WJbqTswt2BKaGWYn2hHhVFKJLXaDz'}}
```

6.4 Comments

The first thing I would emphasise is that the ZCap spec is very much a work in progress. If we went down this route it is something we would likely have to take on and champion. Doing this would also mean we could shape it though. For example, it is not at all clear to my why the `proof` needs the `capabilityAction` property, in fact the spec is a little contradictory here. I also would like to know why we need the `invocationTarget`, when it is specified by the `capability`. It feels like we are duplicating things. So it may be possible that all that is required to use ZCaps, would be an additional property in the `proof` (`capability`) AND an additional `@context` value (https://w3id.org/zcap/v1).

6.4.1 What do we get for this anyway?

The update payload is self contained and self describing. Our spec still defines the custom way to dereference a capability ID to retrieve the root capability. But the payload clearly describes what it is, a capability that is invoked at the target (a DID). The cost for this seems minimal and is greatly reduces the complexity of the spec we have to write. Because the only custom part of this is how to deterministically dereference a capability ID. Additionally AND optionally we get a coherent mechanism to describe delegation and scoping of actions. This of course would require additional fields and increase the size of the update payload. There is also some non-trivial complexity around how we support this with beacons.

However, I think the question of delegation and action scoping should be a separate discussion.

It at least does not feel like a priority, but a nice to have that we might explore in the future. I personally, after working through this notebook, have become more convinced that ZCaps make sense and do not unreasonably increase the number of bytes required for an update payload.

7 5. Size Analysis for Typical Updates

I am going to look at the size of different update payloads for a set of typical updates expected to be performed by a DID controller on a DID document. The updates are as follows:

1. Adding a service
2. Replacing a beacon service
3. Adding a verificationMethod and authentication verificationRelationship & removing a service
4. Rotating (replacing) all (4) verificationMethods in the DID document. Keeping the same IDs - no need to change verificationRelationships

For each of these four updates I will look at the sizes of the following objects

1. The JSON patch update
2. A signature (invariant - always the same size)
3. JSON secured with Data Integrity
4. JSONLD secured with Data Integrity
5. ZCAP_LD

Where the payload is JSONLD I will include numbers for CBORLD compression.

[5]:
```
# 1. Adding a LinkedDomain service
add_service = [{'op': 'add',
  'path': '/service/4',
  'value': {'id': '#linked-domain',
   'type': 'LinkedDomains',
   'serviceEndpoint': 'https://contact-me.com'}}]


# 2. Replacing a beacon service
replace_beacon_service = [{'op': 'replace',
  'path': '/service/4',
  'value': {'id': '#smt_aggregated',
 'type': 'SMTAggregatedBTCBeacon',
 'serviceEndpoint': 'bitcoin:
 ↪tb1pfdnyc8vxeca2zpsg365sn308dmrpka4e0n9c5axmp2nptdf7j6ts7eqhr8'}
  }]


# 3. Adding a verificationMethod and authentication verificationRelationship &
 ↪removing a service
add_vm_auth_remove_service = [
    {'op': 'remove', 'path': '/service/4'},
    {'op': 'add', 'path': '/verificationMethod/1', 'value': {
        "id": "#auth-keys-1",
```

```
        "type": "Ed25519VerificationKey2020",
        "controller": "did:btc:5kq8whVLtvEgLhhY2uKff2GSv3sBKDKcQKiwSTLNuqeh",
        "publicKeyMultibase": "zH3C2AVvLMv6gmMNam3uVAjZpfkcJCwDwnZn6z3wXmqPV"
        }
    },
     {'op': 'add', 'path': '/authentication/1', 'value': '#auth-keys-1'}
]


# 4. Rotating (replacing) all (4) verificationMethods in the DID document.␣
 ↪Keeping the same IDs - no need to change verificationRelationships
rotate_vms = [
    {'op': 'replace', 'path': '/verificationMethod/0', 'value': {
      "id": "#keys-1",
      "type": "EcdsaSecp256k1VerificationKey2019",
      "controller": "did:btc:5kq8whVLtvEgLhhY2uKff2GSv3sBKDKcQKiwSTLNuqeh",
      "publicKeyHex" :␣
 ↪"034ee0f670fc96bb75e8b89c068a1665007a41c98513d6a911b6137e2d16f1d300"
    }
    },
    {'op': 'replace', 'path': '/verificationMethod/2', 'value': {
      "id": "#keys-2",
      "type": "EcdsaSecp256k1VerificationKey2019",
      "controller": "did:btc:5kq8whVLtvEgLhhY2uKff2GSv3sBKDKcQKiwSTLNuqeh",
      "publicKeyHex" :␣
 ↪"034ee0f670fc96bb75e8b89c068a1665007a41c98513d6a911b6137e2d16f1d300"
    }
    },
    {'op': 'replace', 'path': '/verificationMethod/3', 'value': {
      "id": "#keys-3",
      "type": "EcdsaSecp256k1VerificationKey2019",
      "controller": "did:btc:5kq8whVLtvEgLhhY2uKff2GSv3sBKDKcQKiwSTLNuqeh",
      "publicKeyHex" :␣
 ↪"034ee0f670fc96bb75e8b89c068a1665007a41c98513d6a911b6137e2d16f1d300"
    }
        },
    {'op': 'replace', 'path': '/verificationMethod/4', 'value': {
      "id": "#keys-4",
        "type": "Ed25519VerificationKey2020",
        "controller": "did:btc:5kq8whVLtvEgLhhY2uKff2GSv3sBKDKcQKiwSTLNuqeh",
        "publicKeyMultibase": "zH3C2AVvLMv6gmMNam3uVAjZpfkcJCwDwnZn6z3wXmqPV"
    }
    }

]

size1_patch = sys.getsizeof(json.dumps(add_service))
```

```
size2_patch = sys.getsizeof(json.dumps(replace_beacon_service))
size3_patch = sys.getsizeof(json.dumps(add_vm_auth_remove_service))
size4_patch = sys.getsizeof(json.dumps(rotate_vms))


# complex_json_patch_size = sys.getsizeof(json.
 ↪dumps(more_complex_json_patch_update))

# print("Complex JSON Patch")
# print(more_complex_json_patch_update)
# print("Size : ", complex_json_patch_size)
```

[7]:
```
import copy

di_proof = {
        "type": "DataIntegrityProof",
        "cryptosuite": "secp-schnorr-2024",
        "verificationMethod": "did:btc:
 ↪6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis#initialKey",
        "proofPurpose": "assertionMethod",
        "proofValue": signature
}

zcap_proof = copy.deepcopy(di_proof)
zcap_proof["invocationTarget"] = "did:btc:
 ↪6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis"
zcap_proof["capability"] = "urn:zcap:root:
 ↪did%3Abtc%3A6itLKk6UfdvCu4LFdWmJgGZt2JSCZbn4YrNhzhSRTxis"
zcap_proof["capabilityAction"] ="Write"
```

[ ]:

[8]:
```
json_di_1 = {
    'patch': add_service,
    'proof': di_proof
}

jsonld_di_1 = {
    "@context": ["https://w3id.org/security/data-integrity/v2", "https://w3id.
 ↪org/json-ld-patch/v1"],
    'patch': add_service,
    'proof': di_proof
}

zcap_1 = {
```

```
    "@context": ["https://w3id.org/zcap/v1", "https://w3id.org/security/
 ↪data-integrity/v2", "https://w3id.org/json-ld-patch/v1"],
    'patch': add_service,
    'proof': zcap_proof
}

size_json_di_1 = sys.getsizeof(json.dumps(json_di_1))
size_jsonld_di_1 = sys.getsizeof(json.dumps(jsonld_di_1))
size_zcap_1 = sys.getsizeof(json.dumps(zcap_1))
```

[9]:
```
json_di_2 = {
    'patch': replace_beacon_service,
    'proof': di_proof
}

jsonld_di_2 = {
    "@context": ["https://w3id.org/security/data-integrity/v2", "https://w3id.
 ↪org/json-ld-patch/v1"],
    'patch': replace_beacon_service,
    'proof': di_proof
}

zcap_2 = {
    "@context": ["https://w3id.org/zcap/v1", "https://w3id.org/security/
 ↪data-integrity/v2", "https://w3id.org/json-ld-patch/v1"],
    'patch': replace_beacon_service,
    'proof': zcap_proof
}

size_json_di_2 = sys.getsizeof(json.dumps(json_di_2))
size_jsonld_di_2 = sys.getsizeof(json.dumps(jsonld_di_2))
size_zcap_2 = sys.getsizeof(json.dumps(zcap_2))
```

[10]:
```
json_di_3 = {
    'patch': add_vm_auth_remove_service,
    'proof': di_proof
}

jsonld_di_3 = {
    "@context": ["https://w3id.org/security/data-integrity/v2", "https://w3id.
 ↪org/json-ld-patch/v1"],
    'patch': add_vm_auth_remove_service,
    'proof': di_proof
}

zcap_3 = {
```

```
    "@context": ["https://w3id.org/zcap/v1", "https://w3id.org/security/
 ↪data-integrity/v2", "https://w3id.org/json-ld-patch/v1"],
    'patch': add_vm_auth_remove_service,
    'proof': zcap_proof
}

size_json_di_3 = sys.getsizeof(json.dumps(json_di_3))
size_jsonld_di_3 = sys.getsizeof(json.dumps(jsonld_di_3))
size_zcap_3 = sys.getsizeof(json.dumps(zcap_3))
```

[11]:
```
json_di_4 = {
    'patch': rotate_vms,
    'proof': di_proof
}

jsonld_di_4 = {
    "@context": ["https://w3id.org/security/data-integrity/v2", "https://w3id.
 ↪org/json-ld-patch/v1"],
    'patch': rotate_vms,
    'proof': di_proof
}

zcap_4 = {
    "@context": ["https://w3id.org/zcap/v1", "https://w3id.org/security/
 ↪data-integrity/v2", "https://w3id.org/json-ld-patch/v1"],
    'patch': rotate_vms,
    'proof': zcap_proof
}

size_json_di_4 = sys.getsizeof(json.dumps(json_di_4))
size_jsonld_di_4 = sys.getsizeof(json.dumps(jsonld_di_4))
size_zcap_4 = sys.getsizeof(json.dumps(zcap_4))
```

7.1 Results

Below shows the table of results, sizes of objects are in bytes. It is worth highlighting again that only one of these objects, the ZCap is an authorization object. To make any of the other objects authorization objects we would need to add additional properties and hence increase the size of the payload.

[28]:
```
from tabulate import tabulate
```

[29]:
```
print(tabulate([["Update Type","JSON Patch", "Sig", "JSON DI", "JSONLD DI"],
                ["Add Service",size1_patch, sig_size, size_json_di_1,␣
 ↪size_jsonld_di_1, size_zcap_1],
                ["Replace Beacon",size2_patch, sig_size, size_json_di_2,␣
 ↪size_jsonld_di_2, size_zcap_2],
```

```
                ["Add VM, Auth Rel",size3_patch, sig_size, size_json_di_3,␣
↪size_jsonld_di_3, size_zcap_3],
                ["Rotate VMs",size4_patch, sig_size, size_json_di_4,␣
↪size_jsonld_di_4, size_zcap_4]
              ], headers="firstrow"))
```

Update Type	JSON Patch	Sig	JSON DI	JSONLD DI
Add Service	183	138	510	608	829
Replace Beacon	245	138	572	670	891
Add VM, Auth Rel	410	138	737	835	1056
Rotate VMs	1143	138	1470	1568	1789

8 6. What is too big in this context?

The size of these updates will depend on the size of the update that we are doing to our DID document. However, given this is ~829 bytes I would not expect this size to exceed 2000 bytes. The only other reason the size of the update payload MIGHT increase, is if we decide to support delegation. However, I think ZCAPs without delegation are still valuable because of the self describing nature of the payload.

These update payloads need to:

1. Be stored by DID controllers for the duration of the lifetime of their DIDs
2. Be sent over the wire to resolvers, either retrieved from a CAS like IPFS or directly communicated by DID controllers

For 1., too big is the total size of all the update payloads a DID controller has to store is unreasonable. For example, lets imagine this size is 10Gb (Which is fairly small and definitely accessible to most people). And lets imagine the average update is 2000 Bytes. **10Gb / 2000 bytes = 625 000 updates**. Which is the number of updates it is possible to store before this gets unreasonable.

For 2., 2000 bytes is roughly equivalent to a page of text. So that seems fine to send over the wire. However, we should remember that a resolver must retrieve all updates associated with the DID they are resolving. So how many update might this be? Imagine I am especially security conscious and I update my DID every week for 10 years. 52 * 10 = 520 updates. 520 * 2000 =~ 1Mb, or less than the average webpage (~2Mb). I would assert that is also fine to send over the wire without issues.

Note: These rough back of the envolope calculations have been done BEFORE considering compression mechanisms such as CBORLD.

Further Note: These calculations are extremely conservative. I find it unlikely that 10Gb is the size of a store deemed unresonably large for most usecases. An average of 2000 bytes per update also seems exagerated, given it is likely people are making minor changes to their DID documents like adding or removing a service or verificationMethod. Perhaps ~1000 bytes average is more reasonable. I am also unsure of the use case that would require a single DID to be updated every week. Those who security/privacy conscious are much more likely to change DIDs regularly than update the DID document of a single DID.

Byte equivalents taken from here - https://www.techtarget.com/searchstorage/definition/How-many-bytes-for

9 7. What about compression?

If we care about the size, we can look at compression. If we are using JSONLD, then we can make use of CBORLD which compresses JSONLD using the keys from the @context file.

Unfortunately CBORLD appears to only be implemented by Digital Bazaar and only implemented in JavaScript. However, I would expect that to change given the Verifiable Credential Barcode work and the interest in this work by governments

The table below shows the compression sizes for CBORLD and bzip applied to the ZCAP object from above.

I used the Db JS library to compress the above ZCAP payload using JSONLD.

[15]:
```
# CBORLD Size values
# Calculated using https://github.com/digitalbazaar/cborld
cborld_1 = 454
cborld_2 = 517
cborld_3 = 625
cborld_4 = 1225
```

[13]:
```
import bz2

bzip_zcap_1 = bz2.compress(json.dumps(zcap_1).encode('utf-8'))
bzip_zcap_2 = bz2.compress(json.dumps(zcap_2).encode('utf-8'))
bzip_zcap_3 = bz2.compress(json.dumps(zcap_3).encode('utf-8'))
bzip_zcap_4 = bz2.compress(json.dumps(zcap_4).encode('utf-8'))

bzip_size_1 = len(bzip_zcap_1)
bzip_size_2 = len(bzip_zcap_2)
bzip_size_3 = len(bzip_zcap_3)
bzip_size_4 = len(bzip_zcap_4)
```

[24]:
```
print(tabulate([["Update Payload", "JSONLD Size", "CBORLD Size", "BZip Size"],
                ["Add Service", size_zcap_1, cborld_1, bzip_size_1],
                ["Replace Beacon Service", size_zcap_2, cborld_2, bzip_size_2],
                ["Add VM & Auth Rel", size_zcap_3, cborld_3, bzip_size_3],
                ["Rotate VMs", size_zcap_4, cborld_4, bzip_size_4]
               ], headers="firstrow"))
```

Update Payload	JSONLD Size	CBORLD Size	BZip Size
Add Service	829	454	571
Replace Beacon Service	891	517	622
Add VM & Auth Rel	1056	625	685
Rotate VMs	1789	1225	865

10 8. Conclusion

10.1 What do we get if we use ZCaps-LD?

If we decide to go with ZCaps-LD, we get a JSON-LD data model and associated context for a self-describing authorization object with an existing specification draft. The ZCaps-LD specification is currently a community draft at the W3C CCG and needs work, but it exists and provides a foundation to build on. Furthermore, this provides, at a minimum conceptual, alignment with existing efforts within this space such as the VC-API. The size of a ZCaps-LD object is necessarily larger than a plain JSONLD object sercured with Data Integrity, ~221 bytes by the analysis in this notebook, however a plain JSONLD object is not an authorization object and can only be understood within the context of a specific custom protocol. To turn the JSONLD object into a self-describing authorization object we would have to add additional fields making the payload larger. While we could define these fields for ourselves and the write a specification for them, ZCaps-LD has make great progress in this area. Finally, ZCaps-LD because it is linked data can make use of CBOR-LD compression if required.

10.2 What do we get if we don't use ZCaps-LD?

If we decide not to go with ZCaps-LD we would require to either choose or define another authorization object. There are other capabilities specification that define authorization objects, for example UCAN, however these are secured using different mechanisms (e.g. JWT). To our knowledge, ZCaps-LD is the only capabilities specification within the JSON-LD landscape and since we have already agreed that we will be securing JSON-LD VC's and we prefer Data Integrity over JWTs then it makes sense to use Data Integrity to secure the update payload. If we want to define our own authorization object, we have to define the data model for this object. This would mean repeating much of the work that ZCaps-LD has already started.

10.3 Why do we need the update payload to be an authorization object?

Without an authorization object it is impossible to know what signed data object means and how it should be interpretted. Authorization objects or capabilities define the target of the capability, the DID document in our case, furthermore they provide a clear mechanism for interpretting and verifying who is authorized to invoke this capability - who can update a DID document in our case.

10.4 What about the size of these things?

Per the discussion in Section 6, we do not believe the size of the ZCap-LD authorization objects to be a concern for most usecases. Back of the envelope calculations suggest that 10 Gb of storage would enable the storage of 625000 updates of size 2000 bytes. Note, all of the example updates tested in Section 5 were smaller than 2000 bytes. When providing a chain of updates for a single DID document over the wire, 1Mb would be enough to send 520 2000 byte updates. Which would mean a DID that has updated every week for 10 years. All of these numbers seem reasonable. If there are concerns about the size of these objects, Section 7 suggests compression of ~40% is possible using either CBOR-LD or bzip. While the spec could mandate a particular compression format, another option available to any DID controller is to compress the data objects they are required to store in a manner they choose as suitable to their use case. Note, in this case they would have to uncompress the data and send JSON-LD over the wire.

[ ]: