

SD-JWT with Commitments DRAFT

Abstract

In order to allow zero knowledge proofs on attributes within SD-JWTs, the hash algorithm definition is extended to allow the usage of commitment schemes. With commitment schemes, we can for example use simple sigma protocols to derive proofs about age, or to bind various VCs without revealing the actual properties (e.g. when using claim based binding for bridging).

1. Introduction

There are various use cases where selective disclosure alone is not enough to secure privacy, or is too restricted for arbitrary use cases. A good example is age verification, where the current state of the art is to include multiple boolean flags based on the age during issuance. This has obvious drawbacks, as for example a person whose age reaches a threshold needs a new credential (as the boolean flag is fixed in the signed JWT-part). With this draft we could use a simple sigma protocol to proof “older than” for any threshold.

When using batch issuance to reduce link-ability (as currently used signature schemes don’t allow for rerandomization), there needs to be a way of refreshing batches. In the current process this is only possible in redisclosing all personal data to the issuing party. In this draft we sketch a method in Appendix C on how rerandomization of a VC can be done without revealing attributes inside the SD-JWT by adding a new blinding factor to the present commitments.

1.1. Hash Function Claim

The `_sd_alg` of the SD-JWT *MUST* be one of the list members in Appendix A. If `_sd_alg` is one of the values in that list, the SD-JWT *MUST* add a `_sd_alg_param` value containing the required parameters for the commitment scheme.

1.1.1. Commitment Scheme specification

The `_sd_alg_param` contains the following fields:

`commitment_scheme`: REQUIRED. The commitment scheme to use. In Appendix A the parameters for commitment schemes defined in this specification are given.

1.1.2. Commitment Linking

For zero knowledge proofs, a unique linking of attributes to commitments needs to be given. As such, if `_sd_alg` is one of the elements in Appendix A the top level object needs a map of attribute names to indices in the `_sd` array. This object itself can be selectively disclosable to not reveal more information than necessary:

TODO: Explain exactly how this works for deeply nested structures

`com_link`: OPTIONAL. A map of attribute name to index in the `_sd` object. This map needs to be present if commitments in `_sd` of this level of disclosures is needed.

1.2. Keybinding JWT

TODO: Can we have ZKPs without key binding? If so how?

The keybinding JWT is extended with the following fields:

zk_proofs: OPTIONAL. An array of ZkpProof structs containing zero knowledge proofs over the respective inputs.

1.2.1. ZkProof

The ZkpProof struct contains the values required to verify the zero knowledge proof on the properties:

TODO: should we allow random linear equations, and if so how does this impact the serialization mechanism.

inputs: REQUIRED. An array of public and private inputs. Public input means a revealed value, such as a age threshold. Private inputs are for example commitments from other credentials that are not revealed but used within the zero knowledge proof

system: REQUIRED. Array of length of input variables, specifying the *linear* equation that should be satisfied.

context: REQUIRED. Byte-Array containing the relevant proof transcript bytes used for the Fiat-Shamir transform.

proof: REQUIRED. The proof value that can be verified.

proof_type: REQUIRED. A proof type specifying how to deserialize proof.

1.2.1.1. equality_proof

An equality proof is serialized as bytes (each scalar and RistrettoPoint has 32 bytes), where com1 and com2 are the randomized commitments from stage one of the sigma protocol:

```
struct EqualityProof {
    s1: Scalar,
    r1: Scalar,
    s2: Scalar,
    r2: Scalar,
    com1: RistrettoPoint,
    com2: RistrettoPoint,
}
```

1.2.1.2. Public Inputs

Public inputs are clear text values that are lifted into the scalar field of the respective scheme. Requirements and algorithms to do so are defined in the respective commitment schemes.

public_value: OPTIONAL. JSON value that can be lifted into the scalar field. Provided as plaintext.

1.2.1.3. Private Inputs

Private inputs are for example commitments from other credentials for which an equality should be proven. Those are already byte representations of elements of the respective group/ring of the commitment scheme.

path: REQUIRED. ClaimsPointer to com_link or array entry for the relevant commitment

value: REQUIRED. The base64-url-safe encoded commitment used in the proof.

Appendix A

A.1 List of Commitment Schemes

- ec_pedersen

A.2 ec_pedersen

A.2.1 Parameters

When using the ec_pedersen format the following parameters are used:

public_params: REQUIRED. Object containing the generator used for the value (g) and the blinding (h)

crv: REQUIRED. Curve used for the Pedersen commitment.

A.2.2 Hashing

To produce commitments that can be used, number types should be converted to values on the scalar field, and any other type should be converted using hash_to_curve algorithm. For nested objects the JCS (<https://www.rfc-editor.org/rfc/rfc8785.html>) **MUST** be used. The first value of the disclosures array **MUST** be used as the blinding, and the attribute name **MUST** be added to the proof transcript.

TODO: Is there a better way to handle non numerical values? E.g. lexicographical representation of strings to allow ordering?

TODO: Do we need to consider too big numerical values?

TODO: How are floating point numbers encoded? Fixed precision and then using integers ("multiply out" the decimals?)

Appendix B

This Appendix goes over a fictitious example of presenting a Diploma, which is not bound to a specific user hold key material, together with an identity statement, which is device (aka user) bound. Normally, to have confidence that both were issued to the same person, we would need to disclose various properties and show equality between them (as the by default used sha hash does not allow for ZKP).

Here we present both credentials, but only disclose the subject and the grade of the diploma, while providing a zero knowledge proof of equality between multiple different attributes.

B.1 Test Vectors

B.1.1 Identity Card

B.1.1.1 Issuer Verification Key

```
{
  "kty": "EC",
  "crv": "P-256",
  "x": "N3g_o1SqMpLQMqSdyGbG9nK01fWUBNy-h-xxlFQlFk",
  "y": "5P7RM7u1-bCrTfcnkWrBGKgwfd0fG7RFwhNay1kvLu8"
}
```

B.1.1.2 Holder Private Key

```
{
  "kty": "EC",
```

}

B

 $\{$

B

eyJhbGciOiJFUzI1NiIsImtpZCI6IjEyMyJ9.eyJzdWIiOiJlc2VyXzQyIiwidXBkYXRlZm9hdCI6MTU3MDAwMDAwMCwiY25mIjp7Imp3ayI6eyJrdHkiOiJFQyIsImNydiI6I1AtMjU2IiwieCI6Ik9xMWF6VEktaWw4ZW94cXdrdGJTYXlFN1NkLTJTMGtwMk1kQ1hkSk0iLCJ5IjoieWh0ZXFhZzZicTAXN3RDdHFzeDk3S2lKemdmMemJTUXZPe1FKQkZhc1hLRSJ9fSwiX3NkIjpbImpsLXh6a0ZlQUhjeHl0TkFwR1dHQkx1ZHhmamhabWtpcTRmark1VEpQRkUiLCJHRFV1OWtmUWxISVV5bXh0aEdQZEhLeXBQWUWY3ZURmFuekcXZHduUkJJiwiWkVRT0NZMEhYNC0yZzhvU2tSN11fV204NnFVVU9sZDNBOFFTT1k4U0FUbYIsImV0d0F4WnQ3UHHVMVQxbmhPY1hTbkRMU2s3MXk3Y2pfakNHcd1YSDZTUTQiLCI0Z0tkSGZCc1FXUWdZRzdDT1hWMU16SThBeDlrOE5XQ0xnUzBYSFFFRWt3IiwiZG9VZFRRM1pJZ25YODlacU1zLWk4MWNiRnZtOGJrSERjUU9heTZ6dFJnYyIsImtpNTZjY3V1aHF1clpYdGpNeE1QNWNwZDFac2V1RVZKOUY0N251OHBUSGMiLCJkc1hNTVVSRFZWMTBjdkdLeXAtbDc1R1VJRkVjUGhwZ0hXN1VmVE1Nem04I10sI19zZF9hbGdfcGFyYW0iOnsiY29tbW10bWVudF9zY2h1bWUiOnsicHVibGljX3BhcmFtcyI6eyJnIjoieNUFsOUJCRTdOU2pacXhRSktrQjF4bnZCM2t4R1FHdm1fbHBpNW1uQUlWVSI6ImgiOiJqa3h1dk50S1Y0X2FRS1pxSUZSYU11allpSk92VFZSOW5hb1NPdWVmalFJIn0sImNydiI6ImVkmjU1MTkifX0sI19zZF9hbGciOiJlY19wZWRLcnNlbiIsImNvbV9saW5rIjpb7ImpdpdVUx25hbWUiOjAsImZhbWlseV9uYW11IjoxLCA1bWFPbCI6MiwicGhvbWVfbnVtYmVYIjozLCJwaG9uZV9udW1iZXJfdmVyaWZpZWQiOjQsImFkZHZH1c3MiOjUsImJpcnRoZGF0ZSI6NiwiYmF0aW9uYWxpZG1lcYI6N319.GkA6zkXjYPhjtpnpBGEKAavrHLUVUbR4JnkPI4qe4h1oKCxh6RfgRZAFLjYU3BmMM2do3Ooz_SXQi18wIr7Ulcw~WyJPV21TX05sTEFCRVVLRfHxZ2ZDWM1HRU5DSkliOVQ0b3M4eDVxTUprc2drIiwiZ212ZW5fbmFtZSI6IkpvaG4iXQ~WyJyVX03emJFZGVYR1Q3VGZqRzRnRk9DWU1u0VAYz3p2eHBGSUVUqjRhaXdZiwiZmFtaWx5X25hbWUiLCJEb2UiXQ~WyJzUj1NX0JCYzFvVzB0Mi1ZUDdJcE5BOXRiXz1EMHN3aH1MUC1LZ1cwOXdvIiwiZW1haWwiLCJqb2huZG91

eyJhbGciOiJFUzI1NiIsImtpZCI6IjEyMyJ9.eyJzdWIiOiJlc2VyXzQyIiwiaXNkIjpjb
ImhwLWUtZX3o4RjB4UVh0TUfYbEpMbjiZlVvdktZaXZwT3c1Y3dWb3pIQ1kiLCJKQ0pUd
XozTHNleUd5WHBsYUhMRVprZVVNMewzNV93T3NLr2xFaWJxb2x3IiwiTmkyVENVTVVWmt
Q1cWFsYW0tVmVPQWdOYmdtdDQtTF9fc3ZBRU45SGN4QSIsIm5KLVBiu1dmbmU5Sk9PeDB
ZNnNiQjNkUnlfR0pmVnprV3FXcWd6ZlVfQ0EiXSuiX3NkX2FsZl9wYXJhbSI6eyJjb21t
aXRtZW50X3NjaGVtZSI6eyJjcniOiJlZDI1NTE5IiwicHViGljX3BhcmFtcyI6eyJoI
joIWmd2SVJPNHZpUlJVMGhCWfh2WEM2c2lfaWhsQmhoTVAXTW9VZFJIb3VIayIsImciOi
IzUGVPSmwxZlRHQTd4SC1jV0ZwaENURERtdDcwUDZORld0Y295aVVoQWxBIn19fSwiX3N
kX2FsZyI6ImVjX3BlZGVyc2VuIiwiaXNkIjpjb21tZW50bmFtZSI6MCwiZmFt
aWx5X25hbWUiOjEsInN1YmplY3QiOjIsImZpbmFsX2dyYWRlIjozfX0.gDs0zxTqgJKMu
BuJv1UvSC8R_tshugj3-
HpWVO62H2wrwnos1kDwMtUknCSnNPLeQdVloMpl3sSZ0hZVbc
JBZQ~WyJYTWpmTXdjUWw0czZMNkdZYVFFM1lHMlZMZTG0VXFVdGlQTUEtanNpUlEwIiwi
Z2l2ZW5fbmFtZSIsIkpvag4ixQ~WyJEMl90clNtNzYxQzc0RDhyQWNHaExhb1dVb3JaUF
pkbtBRVMjNBep2QndziIiwiaXNkIjpjb21tZW50bmFtZSI6MCwiZmFt
aWx5X25hbWUiLCJEeb2UiXQ~WyJ2MXl1akU5bDloYmxs
RllnY3hPaUlWS19jcuZfUGYyR0dHTUhpVkxaZGc0Iiwic3ViamVjdCI6eyJkbXB1dGVyI
FNjaWVvuY2UiXQ~WyIwWW9tWldPRmU1V0o0OGF1Z2FZMjR3M1R1UGhha2Z5NGt5V2x6aEN
SNXdBIiwiZmluYWxfZ3JhZGUilDQuOF0~

eyJhbGciOiJFUzI1NiIsImtpZCI6IjEyMyJ9.eyJzdWIiOiJlc2VyXzQyIiwidXBkYXR1Zi9hdCI6MTU3MDAwMDAwMCwiY25mIjp7Imp3ayI6eyJrdHkiOiJFQyIsImNydiI6I1AtMjU2IiwieCI6Ik9xMWF6VEktbmZYUmp4ZW94cXdrdGJTYXlFN1NkLTJTMGtwMk1kQ1hkSk0iLCJ5IjoioiWhO0ZXFaRzZicTAXN3RDdHFzeDk3S2lKemdmMemJTUXZPe1FKQkZhc1hLRSJ9fSwiX3NkIjpbImpsLXh6a0ZlQUhjeHl0TkFwR1dHQkx1ZHhmamhabWtpcTRmark1VEpQRkUiLCJHRFV1OWtmUWxISVV5bXh0aEdQZEhLeXBQUUwY3ZURmFuekcXZHduUkJJIiwiWkVRT0NZMEhYNC0yZzhvU2tSN1lfV204NnFVVU9sZDNBOFFTT1k4U0FUbyIsImV0d0F4WnQ3UHHVMVQxbmhPY1hTbkRMU2s3MXk3Y2pfakNHcd1YSDZTUTQiLCi0Z0tkSGZCc1FXUWdZRzdDT1hWMU16SThBeDlrOE5XQ0xnUzBYSFFFFRwt3IiwiZG9VZFRRM1pJZ25YODlacUlzLWk4MWNiRnZtOGJrSERjUU9heTZ6dFJnYyIsImtPNTZjY3V1aHF1clpYdGpNeE1QNWNwZDFac2V1RVZKOUY0N25lOHBUSGMIcJkc1hNTVVsrFZwMTBjdkdLeXAtbDc1R1VJRkVjUGhwZ0hXN1VmVElNem04I10sI19zZF9hbGdfcGFyYW0iOnsiY29tbWl0bWVudF9zY2h1bWU0iOnsicHVibGljX3BhcmFtcyI6eyJnIjoioiNUFsOUJCRTd0U2pacXhRSktrQjF4bnZCM2t4R1FHdm1fbHBpNW1uQUlWVSIsImgiOiJqa3hldk50S1Y0X2FRS1pxSUZSYU11a1lpSk92VFZSOW5hb1NPdWVma1FJIn0sImNydiI6ImVkMjU1MTkiXfX0sI19zZF9hbGciOiJlY19wZWRLcNlbiIsImNvbV9saW5rIjpb7ImpdpdmVuX25hbWUiOjAsImZhbWlseV9uYW11IjoxLCJlbWFpbCI6MiwicGhvbmlfYmVtYmVYIjozLCJwaG9uZV9udW1iZXJfdmVyaWZpZWQiOjQsImFkZHU1c3MiOjUsImJpcnRoZGF0ZSI6NiwiYmF0aW9uYWxpZGl1cyI6N319.GkA6zkXjYPhjtpnBGEKAavrHLUVUBR4JnkPI4qe4h1oKCxh6RfgrZAF1jYU3BmMM2do3Ooz_SXQi18wIr7Ulcw~eyJ0eXAI0iJrYitqd3QiLCJhbGciOiJFUzI1NiJ9.eyJub25jZSI6IjFWMTNRRXJQYjhhqR1NpSU5kR2RnWUV1OWU2bHhVU09nIiwiaWF0IjoixNzY4NDc2NjU3LCJzZF9oYXNoIjoioiQXBzd2FGNlM4VHk1VWFLa2h1cXfVYzdMTnlQU11JaTl3R1VNZ2hGRDlnOCIsInprX3Byb29mcyI6W3siaW5wdXRzIjpbeyJQcm12YXR1Ijpb7InBhdGgiOl1siY29tX2xpbmsiLCJnaXZlbi19uYW11I10sInZhbHV1IjoiamwteHprRmVBSGN4eWh0QXBHV0dCTGVkeGZqaFpta21xNGZqeTVUSlBGRSJ9fSx7I1ByaXZhdGUiOnsicGF0aCI6WyJjb21fbGl1ayIsImdpdmVuX25hbWUiXSwidmFsdWUiOiJocC1lc196OEYweFFfNE1BcmxKTG4yM2dVVXZLWW12cE93NWN3Vm96SENZIn19XSwic3lzdGVtIjpbMSwtMV0sImNvbnRleHQiOiJaMmwyW1c1ZmJtRnRaZVFKZ1FRUK96VW8yYXNVQ1NwQWRjWjd3ZDVNUmtCcjV2NWFZdVpzd0NGVmpreGV2TnRKvjRfYVFLWnFJR1JhSXVqWW1KT3ZUV1I5bmFvU091ZWZqUUXjOTQ0bVhXQk1ZRHZFZjV4WVdtRUptU9hM3ZR28wVmExeWpLS1NFQ1VHWUx5RVR1TDRrVZOSVFWMTcx3VYsXY0b1pRWV1URD1US0ZIVVI2TGg1QVFJRCIsInByb29mIjoioiRMQ1YyX2NaeTBqSVFrSlhfVVDfE5oS1JwN2JWd2s1VXdhNS1KWGFBM1E3U3lHRk4yMk9iTzJfa01MbJNHMK04V2dCeEk3QkhkdzlsN0pld2xRQjhBeXdWZHYzR2N0SX1FSkNWXzFGaE1UWVNVYUyMWNKT1ZNR3VmaVYyZ05VM2xadWdrenBTMmd1N0VXV2c2VjdxZk5QT3JHc21iWjZudmdPdFc5bmd6eVpZTmFTS3RmeDJWazJELUg2Y11DQjIyY1I5YXNqSjhaZ0RCdy1TWWKhUTZOSmFGNjRhamgwRDdmZHZN1Jfb1BfCGZJeDFqRTYxQWtnYlBBdzlkVCIsInByb29mX3R5cGUiOiJlcXVhbGl0eV9wcm9vZiJ9LHsiaW5wdXRzIjpbeyJQcm12YXR1Ijpb7InBhdGgiOl1siY29tX2xpbmsiLCJmYW1pbH1fbmFtZSJdLCJ2YWx1ZSI6IkdEVXU5a2ZRbEhJVX1teE5oR1BkSEt5cFBZRTBjd1RGYW56RzFkd25SQkkiXfX0seyJQcm12YXR1Ijpb7InBhdGgiOl1siY29tX2xpbmsiLCJmYW1pbH1fbmFtZSJdLCJ2YWx1ZSI6IkpDS1R1ejNMc2V5R31YcGxhSExFWmt1VU0wTDM1X3dPc0tHbEVpYnFvbHcifX1dLCJzeXN0ZW0iOl1sXC0xXSwiY29udGV4dCI6I1ptRnRhV3glWDI1aGJXWGTdWDBFRVRzMUtObXJGQWtxUUhYR2U4SGVURVpBYS1iLVdtTG1hY0FoVlkl1TVhyemJTVmVQMmtDbWFPQ1VXaUxvMk1pVHIwMVVwMjJxRWpYbm40MEMzU

Appendix C

C.1 Refresh of batch issued credentials in OID4VCI context

Instead of using a refresh token to refresh the batches of a credential, the issuer could allow issuing of new credentials after presentation of a valid original credential. Instead of reissuing the credential, the issuer would just take all commitments in the sd object and add a random blinding factor to the respective commitments (using the generator defined in commitment_scheme). The returned SD-JWT's disclosures would then only contain a delta to the blinding factor, the wallet could use to calculate the actual blinding (by adding it to the blinding factor of the credential used in the request). Furthermore, using request and response encryption as defined in Section 10, OpenID for VCI, the wallet could use a TOR like routing to hide its network trail to the issuer.

Appendix D

D.1 Sample Code for JSON canonicalization

```
pub fn canonicalize_object(v: &heidi_util_rust::value::Value) -> String {
    let Some(obj) = v.as_object() else {
        return String::new();
    };
    let mut keys = obj.keys().collect::<Vec<_>>();
    keys.sort();
    let mut output_string = String::new();
    output_string.push_str("{");
    for key in keys {
        output_string.push_str(r#"{"#);
        output_string.push_str(key);
        output_string.push_str(r#"":#);
        output_string.push_str(":");
        output_string.push_str(&stringify_value(obj.get(key).unwrap()));
        output_string.push_str(",");
    }
    if output_string.contains(",") {
        output_string = (&output_string[..output_string.len() - 1]).to_string();
    }
    output_string.push_str("}");
    output_string
}

fn canonicalize_array(v: &Value) -> String {
    let mut output_string = String::new();
    output_string.push_str("[");
    for item in v.as_array().unwrap() {
        output_string.push_str(&stringify_value(item));
        output_string.push_str(",");
    }
    if output_string.contains(",") {
        output_string = (&output_string[..output_string.len() - 1]).to_string();
    }
    output_string.push_str("]");
    output_string
}

fn canonicalize_primitive(v: &Value) -> String {
    let serde_json_value: serde_json::Value = v.into();
    serde_json::to_string(&serde_json_value)
```

```
        .unwrap()
        .trim()
        .to_string()
    }
pub fn stringify_value(value: &heidi_util_rust::value::Value) -> String {
    match value {
        Value::Array(_) => canonicalize_array(value),
        Value::Object(_) => canonicalize_object(value),
        _ => canonicalize_primitive(value),
    }
}
```