Design Notes for VSW Verify M3

Wenjing Chu, March 18, 2021

M3 is built by extending the functionality already implemented in M2. These notes here will only address the extensions and not repeat the main functions themselves.

In M3, several extensions are made that enable the verifier a lot more capabilities and flexibilities.

In M2, we simplified the verifier by statically confining it to be able to make a single request only. That single proof request is hard-coded. That simplification allows us to test the underlying protocol without complexity. Now, we must give the verifier all freedom to ask any legitimate questions for the prover to prove. Because

(1) we added another attestation schema, [1]

(2) we added a lot more fields to the software schema, [2]

the verifier is now able to combine these to ask much more complicated questions.

Here we document the new functions we need to implement in M3.

1. **Proof Request**

- **Using proof request config file**

Recall that the verifier can ask for proofs of mainly two types: ***attributes*** and ***predicates***.

Attributes are fields defined in one or more credentials. The way we code up a fixed proof request in M2 is shown below:

```
req_attrs = [
            {
                "name": "softwareName",
                "restrictions": [
                            {"schema_name": "softwareCertificate",
                             “issuer_did”: “XD7vh3QA2SgYMJdNyzEwvv”
                            }]
            },
            {
                "name": "softwareVersion",
                "restrictions": [
                            {"schema_name": "softwareCertificate",
                            “issuer_did”: “XD7vh3QA2SgYMJdNyzEwvv”
                            }]
            },
            {
                "name": "developerDid",
                "restrictions": [
                            {"schema_name": "softwareCertificate",
```

```
                        “issuer_did”: “XD7vh3QA2SgYMJdNyzEwvv”
                        }]
                },
                {
                    "name": "hash",
                    "restrictions": [
                        {"schema_name": "softwareCertificate",
                        “issuer_did”: “XD7vh3QA2SgYMJdNyzEwvv”
                        }]
                },
                {
                    "name": "url",
                    "restrictions": [
                        {"schema_name": "softwareCertificate",
                        “issuer_did”: “XD7vh3QA2SgYMJdNyzEwvv”
                        }]
                }
            ]
```

The above is hard coded in M2.

In M3, we will move this spec to a config file in JSON and the vsw verify command will read from the config file.
`% vsw verify --proof-request filename`

Additionally, we should have a default proof-request which simply asks for all available information. Other ease of use extensions can be made by giving verifier users premade config files to cover common uses. This is nice to have but not absolutely required.

Note: we have plans to make the usability a lot better by implementing a human friendly inquiry language of some sort. But that is beyond the scope of M3.

- **Supporting new test schema and combined proof with 2 schemas (Proof Request)**

In M3, there are 2 types of schemas: “softwareCertificate” and “testCertificate”. So the above list can include items mixed together from these two schemas (and any more in the future).

`For example,`

```
    req_attrs = [
                {
                    "name": "softwareName",
                    "restrictions": [
                        {"schema_name": "softwareCertificate",
                        “issuer_did”: “XD7vh3QA2SgYMJdNyzEwvv”
                        }]
                },
                {
                    "name": "softwareVersion",
```

```
        "restrictions": [
                    {"schema_name": "softwareCertificate",
                    “issuer_did”: “XD7vh3QA2SgYMJdNyzEwvv”
                    }]
    },
    {
        "name": "developerDid",
        "restrictions": [
                    {"schema_name": "softwareCertificate",
                    “issuer_did”: “XD7vh3QA2SgYMJdNyzEwvv”
                    }]
    },
    {
        "name": "hash",
        "restrictions": [
                    {"schema_name": "softwareCertificate",
                    “issuer_did”: “XD7vh3QA2SgYMJdNyzEwvv”
                    }]
    },
    {
        "name": "url",
        "restrictions": [
                    {"schema_name": "softwareCertificate",
                    “issuer_did”: “XD7vh3QA2SgYMJdNyzEwvv”
                    }]
    }
      {
        "name": "testerDid",
        "restrictions": [
                    {"schema_name": "testCertificate",
                    “issuer_did”: “AKSo8LotgVyQSKzYah9wny”
                    }]
    }
      {
        "name": "testSpecDid",
        "restrictions": [
                    {"schema_name": "testCertificate",
                    “issuer_did”: “AKSo8LotgVyQSKzYah9wny”
                    }]
    }
]
```

Now, in addition to request proof about what software it is, developed by whom, we request that it has been tested by a known tester, for example, maybe a Software Safety Certification Inc. And more specifically, we ask it to prove it has passed a specific test (e.g. a national standard).

Note: The above example may be simplified while still achieving the same inquiry. I’ll leave that out for now. Once we have a tool to play with, we should be able to figure the obvious out soon.

- **Supporting Predicates (Proof Request)**

In M2, we decided to avoid using predicates to reduce workload. Now, we have the fields necessary for practical uses of predicates. There are few examples available showing how to set predicates however. [5] is a rare example which I rely on. So this is going to be a bit of guessing and trial and error here.

The following example shows using predicates in proof request:

```
type RequestedPredicate struct {
	Restrictions []Restrictions `json:"restrictions"`    // Required when using Names,
otherwise empty slice instead of nil
	Name         string         `json:"name,omitempty"`  // XOR with Names
	Names        []string       `json:"names,omitempty"` // XOR with Name | Requires
non-empty restrictions
	PType        PredicateType  `json:"p_type"`
	PValue       int            `json:"p_value"`
	NonRevoked   NonRevoked     `json:"non_revoked"` // Optional
}
```

From here we can construct similarly:

```
req_preds = [
		“name”: “ranking”,
		“p_type”: ">=",
		“p_value”: 4,
		“restrictions”: [
				{	"schema_name": "testCertificate",
					“issuer_did”: “AKSo8LotgVyQSKzYah9wny”
				}
				]
	]
```

Essentially, the verifier wants to see that not only the software passed a particular test by a particular test organization but also that it has been ranked by that organization with 4 stars or better.

Note: the code example above is not proven so please test first.

- **Supporting Proof of Non-revocation (Proof Request)**

Another feature that we skipped in M2 is non_revoked setting. In M3, the proof request will set non_revoked field. The details are already documented in [6]. In a nutshell, we will insist From=To. But the verifier can either specify From=To=Now(), or From=To=Now()-a time period. Now we are done with proof request extensions, let’s look at what we need to extend presenting proofs by the prover.

2. **Present Proof**

When the prover, i.e. vsw-repo, receives a Proof Request, it responds with a Proof Presentation, or simply Proof. To construct a proof in response to a Proof Request, it needs to parse what is

being asked to prove (attributes, predicates) and then query its wallet to see if it can find matching credential information. The information may have to come from a combination of several credentials.

In the M2 release, we only use one specific fixed Proof Request hard-coded. Now, we must support any possible proof requests the verifier may send. The prover, therefore, has to parse the request information and construct a proof accordingly. This is the major change.

The following Alice example code [4] does seem to cover multiple schemas, predicates, and non-revocation, so it should still be a good example to reference. We should continue to skip the self_attested list. If we can't find matching information, we should reply with a Presentation Reject message instead [1].

```
if state == "request_received":
    log_status(
        "#24 Query for credentials in the wallet that satisfy the proof request"
    )

    # include self-attested attributes (not included in credentials)
    credentials_by_reft = {}
    revealed = {}
    self_attested = {}
    predicates = {}

    # select credentials to provide for the proof
    credentials = await self.admin_GET(
        f"/present-proof/records/{presentation_exchange_id}/credentials"
    )
    if credentials:
        for row in sorted(
            credentials,
            key=lambda c: int(c["cred_info"]["attrs"]["timestamp"]),
            reverse=True,
        ):
            for referent in row["presentation_referents"]:
                if referent not in credentials_by_reft:
                    credentials_by_reft[referent] = row

    for referent in presentation_request["requested_attributes"]:
        if referent in credentials_by_reft:
            revealed[referent] = {
                "cred_id": credentials_by_reft[referent]["cred_info"][
                    "referent"
                ],
                "revealed": True,
            }
        else:
            self_attested[referent] = "my self-attested value"
```

```
    for referent in presentation_request["requested_predicates"]:
        if referent in credentials_by_reft:
            predicates[referent] = {
                "cred_id": credentials_by_reft[referent]["cred_info"][
                    "referent"
                ]
            }

    log_status("#25 Generate the proof")
    request = {
        "requested_predicates": predicates,
        "requested_attributes": revealed,
        "self_attested_attributes": self_attested,
    }

    log_status("#26 Send the proof to X")
    await self.admin_POST(
        (
            "/present-proof/records/"
            f"{presentation_exchange_id}/send-presentation"
        ),
        request,
    )
```

3. Verify Proof

The verification procedure is executed by the verifier after it receives the Proof Presentation from the prover. We again follow the example in Acme [2].

The first step is to ask the agent to verify that the received proof is valid cryptographically.

```
if state == "presentation_received":
    proof = await self.admin_POST(
        f"/present-proof/records/{presentation_exchange_id}/verify-presentation"
    )
    self.log("Proof = ", proof["verified"])
```

I think this step also verifies non_revoked status by the agent. Please confirm.

Once we are sure it is valid, we can then verify its information as corresponding to what we asked in the Proof Request.

```
    pres_req = message["presentation_request"]
    pres = message["presentation"]
    is_proof_of_software_certificate = (
        pres_req["name"] == "Proof of Software Certificate"
    )
    if is_proof_of_software_certificate:
        # check claims
        for (referent, attr_spec) in pres_req["requested_attributes"].items():
```

```
self.log(
    f"{attr_spec['name']}: "
    f"{pres['requested_proof']['revealed_attrs'][referent]['raw']}"
)
...
```

In M3, we have to assume the prover may send any kind of proof and have code to deal with all other cases. If the verifier detects that the received proof is not exactly as it is expecting, print out error messages to show verification failure.

The verifier now has *true information* and can then follow up with more verifications. The main step we want to implement is to verify that the software package hosted at the URL can be downloaded (available) and that it matches the hash in the credential. To implement this step of verification, we could implement the following pseudo code.

1. Check URL and download software package from the URL
2. Calculate SHA256 (note: we know it is SHA256 by checking the “hash” field which encodes algorithm type (0x12)
3. Compare the SHA256 with the “hash”
4. Print out result

Note
In M3, we are still not recursively verifying dependencies.

References

[1] Aries RFC 0037:
https://github.com/hyperledger/aries-rfcs/tree/4fae574c03f9f1013db30bf2c0c676b1122f7149/features/0037-present-proof

[2] Acme example code:
https://github.com/verifiablesoftware/aries-cloudagent-python/blob/vsw/demo/runners/acme.py

[3] Ldej blog post:
https://ldej.nl/post/becoming-a-hyperledger-aries-developer-part-7-present-proof/

[4] Alice example code:
https://github.com/verifiablesoftware/aries-cloudagent-python/blob/vsw/demo/runners/alice.py

[5] Ldej predicate code sample:
https://github.com/ldej/go-acapy-client/blob/88274d54e257f9ae09fbd850853f2ab06a44bf0d/present_proof.go#L249

[6] Design notes on VSW revocation:
https://docs.google.com/document/d/1Xyz0Q7U74wBdceg1oo6baqj5znNzaVDjC4v3qSbNuW4/edit