RVI CORE PROTOCOL

This document describes the core protocol between two RVI nodes.

STANDARDS USED

[1] Transport Layer Security - TLS (link)[https://tools.ietf.org/html/rfc5246]
[2] JSON Web Token RFC7519- JWT (link)[https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32]
[3] MessagePack - (link)[http://msgpack.org/index.html]
[4] base64url - (link)[https://en.wikipedia.org/wiki/Base64)
[5] Transport Layer Security (TLS) - (link)[https://en.wikipedia.org/wiki/Transport_Layer_Security]
[6] X.509 Certificates - (link)[https://en.wikipedia.org/wiki/X.509]

FEATURES COVERED BY PROTOCOL

1. **Authorization**
   Prove to the remote RVI node that the local RVI node has the right to invoke a set of services, and the right to register another set of services.
2. **Service Discovery**
   Announce to the remote RVI node local RVI services which the remote node is authorized to invoke.
3. **Service Invocation**
   Invoke services on remote RVI nodes.

FEATURES NOT COVERED BY PROTOCOL

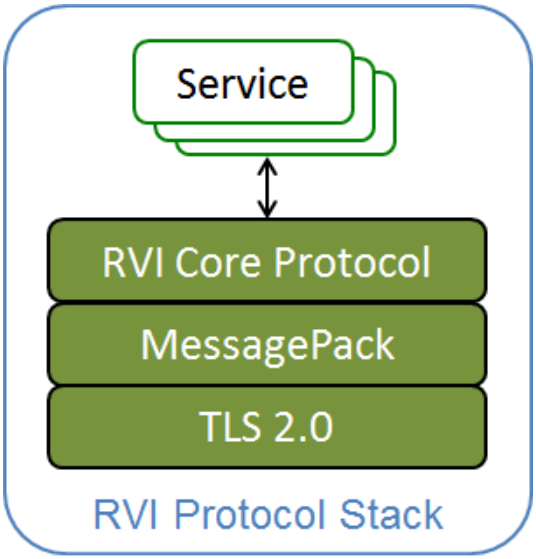
For all but the last item, TLS 1.2 [5] an be used as an underlying protocol to provide the features lacking in RVI Core protocol.

1. **Authentication**
   Prove the identity oof a local RVI node to the remote RVI node.
2. **Encryption**
   Encrypt data between two RVI nodes to avoid eavesdropping.
3. **Replay attack protection**
   Replay an earlier RVI Core protocol session to engage with an RVI node again.
4. **Man in the middle attack protection**
   Terminate an RVI Core protocol connection, modify incoming data and forward it to its original destination.
5. **Key Management**
   Public Key Infrastructure and certificate distribution.
6. **RVI Node Discovery**
   Allowing two unconnected RVI nodes to discover each other so that they can initiate connection.

OVERVIEW

The RVI core protocol is the default protocol used between two RVI nodes once they have become aware of each other's presence.

The stack schematics is shown below.



RVI Core protocol codec

The RVI core protocol uses MessagePack [3] as its encoder/decoder to transmit JSON structures. All JSON structures described in this protocol are encoded as MessagePack prior to transmission to the remote peer.

Certificates and credentials

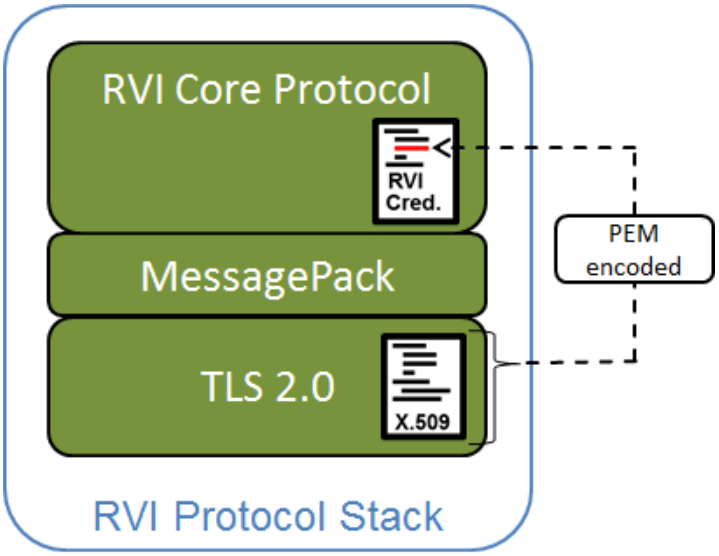
Three types of certificates and credentials are used by the RVI Core protocol in conjunciton with TLS. See [6] for details on X.509.

1. **Root certificate [X.509]**
   Generated by a trusted provisioning server and pre-provisioned on all RVI nodes. Self signed. Used to sign all RVI certificates. Used to sign all device certs.
2. **Device certificate (X.509)**
   Per-device certificate. Signed by root cert. Used by TLS for initial authentication.
3. **RVI credentials (JWT)**
   Describes the services that the device has the right to invoke and the services that the device has right to register. Embeds the device X.509 certificate as a PEM-encoded string. Signed by root cert.

Integration between TLS and RVI Core RVI

Client and server X.509 certificates are exchanged when the original TCP connection is upgraded to TLS. Once a X.509 certificate has been validated by the receiving party party, it will be matched against the PEM-encoded X.509 certificate embedded in received RVI credentials.

The figure below shows how this is done.



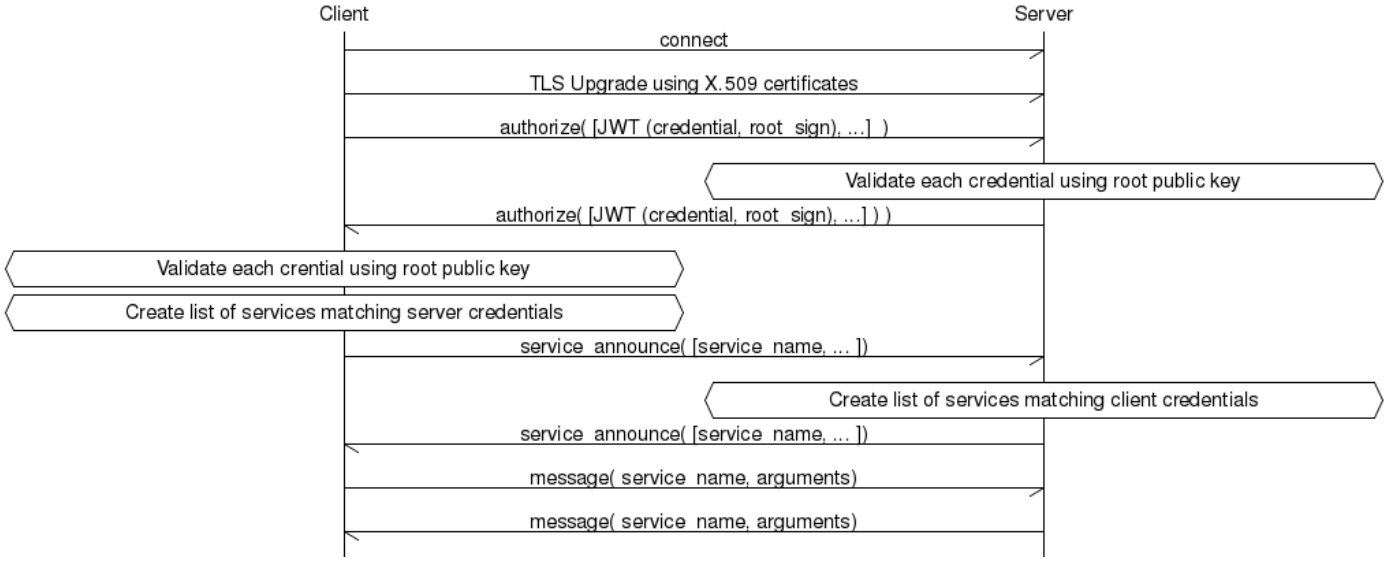
JSON Web token usage

JSON Web Tokens (JWT) [2] are used to encode RVI credentials, which are signed by the root x.509 certificate.

PROTOCOL FLOW

Sequence Diagram

The diagram below outlines the sequence between the client and the server. Please note that the protocol is fully symmetrical and that the client-server terminology only denotes who initiates the connection (client), and who receives that connection (server).



Authorize command

The `authorize` command contains a list of RVI credentials, each specifying a set of services that the sender has the right to invoke on the receiving node, and a set of services that the sender has the right to register.

Please see the "RVI Credentials" chapter for detailss on RVI credentials.

Service Announce command

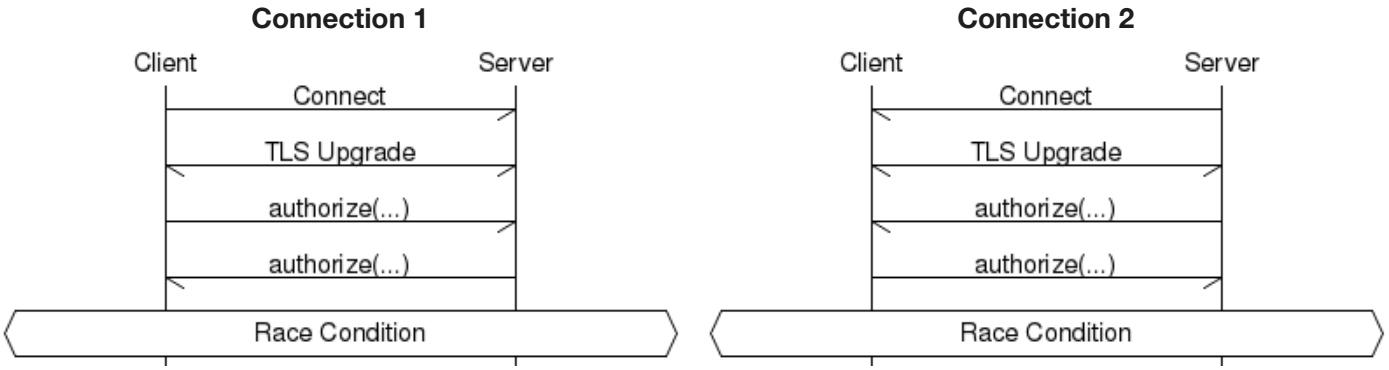
The `service_authorize` command contains a list of services available on the sender that match services listed in RVI credentials received from the remote party.

Message command

The `message` command contains a service name and a number of arguments to be presented to the corresponding service at the receiving end. This is an asynchronous command that does not expect an answer. Replies, publish/subscribe, and other higher-level functions are (for now) outside the scope of the RVI Core protocol.

Double connect resolution

There is a risk that two parties try to initiate a connection to each other in a race condition, creating two connections between them, as shown below.



A double connect can be detected by either side by checking if the remote peer address already has a connection established.

In the diagram above, both the client and the server will initite a connection to the other node at the same time.

Shortly afterwards, both will receive an incoming connection from the other node.

By comparing the incoming connection's peer address against all other connections' peer addresses, a match will be found in the outbound connection just initiated.

Once a double connect has been detected, an implicit agreement is reached to abort the connection initiated by the RVI node with the highest address. If both RVI nodes share the same address (i.e. they run on the same host), the connection with the highest source port is aborted.

Below is a table with a number of double connect scenarios, showing which connection would be termianted.

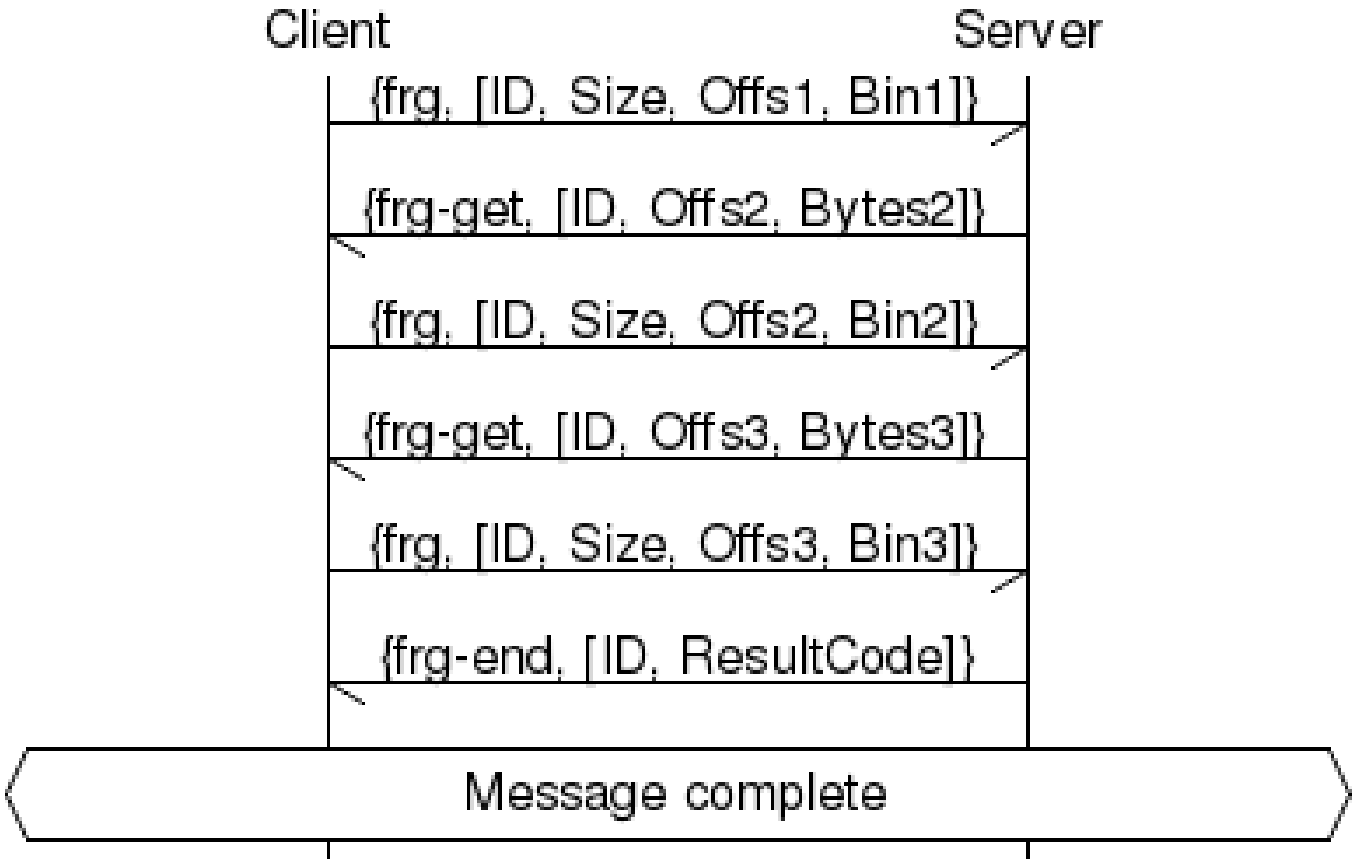
Node1 Address	Node2 Address	Connecting side to be terminated
23.200.227.113:12831	144.63.252.10:33829	Node2
192.26.92.30:11102	192.52.92.31:9884	Node2
192.26.92.30:11102	192.52.92.30:11101	Node1

The connection is terminated regardless of its current protocol session state.

Chunking of large messages

RVI Core is able to split large messages into fragments and deliver them reliably to the receiver; if the receiving end is an RVI node, re-assembly is performed automatically. The fragmentation logic is called at the data link level, so all messages, including RVI Core handshake messages, can be fragmented.

The protocol is as follows:



Enabling fragmentation

Fragmentation can be turned on either per data link type, or per message.

The two options that affect fragmentation are

- "reliable" (true | false): One fragment containing the whole message is sent, and acknowledged with a "frg-end" message. This effectively enables reliable message delivery.
- "max_msg_size" (Bytes): this specifies a maximum window size. RVI Core will try to stay within the window size including the framing overhead, but this will currently be unreliable when using JSON encoding, due to escaping of binary data.

When including these options in the "parameters" list of a message invocation, the names can be prefixed with "rvi.", e.g. "rvi.max_msg_size".

TODO: Introduce timers. Currently there are none.

Re-assembly

The receiving side is responsible for re-assembling the message and detect holes (missing fragments). The sending side will only the first fragment (with a starting offset of 1), and then wait for the receiving side to request more fragments using "frg-get" messages. When the sending side receives a "frg-end" message, it will forget about the message.

Encoding

By default, the fragmentation logic will use the same encoding as the data link layer, but this is configurable. RVI Core currently supports JSON and msgpack encoding. Of these two, msgpack is more efficient and predictable for encoding binary data.

The RVI Core data link layers detect the encoding on a per-message basis. This is possible, as all RVI Core messages are either structs (JSON) or maps (msgpack), and these encodings are distinguishable on the first non-whitespace byte.

Configuring fragmentation encoding in RVI Core is done for the specific data link module, e.g.

```
    { data_link,
      [ { dlink_tcp_rpc, gen_server,
          [
                { frag_opts, {rvi_data_msgpack, []} },
        { json_rpc_address, { 192.168.1.32, 8806 } },
        { server_opts, [ { port, 8807 }]},
        { persistent_connections, [ "192.168.1.10:8807" ]}
          ]
        }
      ]
    }
```

PROTOCOL DEFINITION

This chapter describes the protocol message formats and how the various fields are used.

For all examples below the following certifcates are used:

Sample root certificate

The self signed root certificate used in the examples throughout this document was generated using the following commands:

```
# Create root key and cert signing request
openssl genrsa -out insecure_root_key.pem 1024

# Create a self-signed root CA certificate, signed by the root key created above
openssl req -x509 -new -nodes -key insecure_root_key.pem -days 365 -out insecure_root_cert.crt
```

The content of the sample `insecure_root_key.pem` private key file, which has no password protection, is:

```
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQDg5A1uZ5F36vQEYbMWCV4wY4OVmicYWEjjl/8YPA01tsz4x68i
/NnlMNalqpGCIZ0AwqGI5DZAWWoR400L3SAmYD6sWj2L9ViIAPk3ceDU8olYrf/N
wj78wVoG7qqNLgMoBNM584nlY4jy8zJ0Ka9WFBS2aDtB3Aulc1Q8ZfhuewIDAQAB
AoGAfD+C7CxsQkSc7I7N0q76SuGwIUc5skmUe6nOViVXZwXH2Or55+qqt+VzsbO7
EJphk7n0ZR0wm/zKjXd3acaRq5j3fOyXip9fDoNj+oUKAowDJ9vub0NOPpU2bgb0
xDnDeR0BRVBOTWqrkDeDPBSxw5RlJunesDkamAmj4VXHHgECQQDzqDtaEuEZ7x7d
kJKCmfGyP01s+YPlquDgogzAeMAsz17TFt8JS4RO0rX71+lmx7qqpRqIxVXIsR58
NI2Th7tRAkEA7Eh1C1WahLCxojQOam/l7GyE+2ignZYExqonOOvsk6TG0LcFm7W9
x39ouTlfChM26f8VYAsPxIrvsDlI1DDCCwJBAITmA8lzdrgQhwNOsbrugLg6ct63
kcuZUqLzgIUS168ZRJ1aYjjNqdLcd0pwT+wxkI03FKv5Bns6sGgKuhX3+KECQFm/
Z93HRSrTZpViynr5R88WpShNZHyW5/eB1+YSDslB1FagvhuX2570MRXxybys8bXN
sxPI/9M6prI8AALBBmMCQD+2amH2Y9ukJy10WuYei943mrCsp1oosWjcoMADRCpj
ZA2UwSzj67PBc5umDIAlhVRMX0zH/gLj54rfIkH5zLk=
-----END RSA PRIVATE KEY-----
```

The root key above is checked in as `priv/sample_keys/insecure_root_key.pem`.

The content of the sample `insecure_root_cert.crt` file is:

```
-----BEGIN CERTIFICATE-----
MIICUjCCAbugAwIBAgIJAMI080XZPsPUMA0GCSqGSIb3DQEBCwUAMEIxCzAJBgNV
BAYTAlVTMQ8wDQYDVQQIDAZPcmVnb24xETAPBgNVBAcMCFBvcnRsYW5kMQ8wDQYD
VQQKDAZHRU5JVkkwHhcNMTUxMTI3MjMxMTQ0WhcNMTYxMTI2MjMxMTQ0WjBCMQsw
CQYDVQQGEwJVUzEPMA0GA1UECAwGT3JlZ29uMREwDwYDVQQHDAhQb3J0bGFuZDEP
MA0GA1UECgwGR0VOSVZJMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDg5A1u
Z5F36vQEYbMWCV4wY4OVmicYWEjjl/8YPA01tsz4x68i/NnlMNalqpGCIZ0AwqGI
5DZAWWoR400L3SAmYD6sWj2L9ViIAPk3ceDU8olYrf/Nwj78wVoG7qqNLgMoBNM5
84nlY4jy8zJ0Ka9WFBS2aDtB3Aulc1Q8ZfhuewIDAQABo1AwTjAdBgNVHQ4EFgQU
4Sz8rAMA+dHymJTlZSkap65qnfswHwYDVR0jBBgwFoAU4Sz8rAMA+dHymJTlZSka
p65qnfswDAYDVR0TBAUwAwEB/zANBgkqhkiG9w0BAQsFAAOBgQDFOapf3DNEcXgp
1u/g8YtBW24QsyB+RRavA9oKcFiIaHMkbJyUsOergwOXxBYhduuwVzQQo9P5nR0W
RdUfwtE0GuaiC8WUmjR//vKwakj9Bjuu73ldYj9ji9+eXsL/gtpGWTIlHeGugpFs
mVrUm0lY/n2ilJQ1hzBZ9lFLq0wfjw==
-----END CERTIFICATE-----
```

The root certificate above is checked in as `priv/sample_certificates/insecure_root_key.pem`.

DO NOT USE THE KEYS AND CERTIFICATES ABOVE IN PRODUCTION!
ANY PRODUCTION KEYS SHOULD BE GENERATED BY THE ORGANIZATION AND BE 4096 BITS LONG.

Sample device certificate

The sample device x.509 certificate, signed by the root certificate above, was generated with the following command:

```
# Create the device key. In production, increase the bit size to 4096+
openssl genrsa -out insecure_device_key.pem 1024

# Create a certificate signing requestsigning request
openssl req -new -key insecure_device_key.pem -out insecure_device_cert.csr

# Sign the signing request and creaqte the root_cert.crt file
openssl x509 -req -days 365 -in insecure_device_cert.csr \
             -CA insecure_root_cert.crt -CAkey insecure_root_key.pem \
             -set_serial 01 -out insecure_device_cert.crt
```

The `insecure_device_cert.csr` intermediate certificate signing request can be deleted once the three steps above have been executed.

The content of the sample `insecure_device_key.pem` private key file, which has no password protection, is:

```
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQCbb4jPAESKxarj3NJsgfQbhfTHZAP9kmram2TFnkzlCRxq4wQx
BDC0O85PAMgZou0armGGbOu0si4cpVRioerCQJXnMWx1MI+3GUktW5ijI3ui+tYC
sMQZtjSBVNXFZdoyZU2lPVWITOMZOe8o9vJ5DcUmFj9b2xV9jQ19oh+2+QIDAQAB
AoGAVCYV0rs6YEaTNbke0k+ocB4dXrTu1CCoaKEn9TS2PGiqUdOFOWQjWe/myS6L
JhXmd0Ng2P2uvayY+jknbh5qkNeEgTDhXJlAjiXlCADYArhgib+evRHgKz7RLTjX
tGklbmc7oECTEpjkchJC5XcJhXzHCIjroyOJvBuAVa+SeAECQQDNC+KW7fTKQpiG
YNGIt5MxCMjRparLz0fWod9J9U56wrWzU9Rnb7h9iwzTEJUEcVl9z8rnUdWtYQ8X
3lsz5cDhAkEAwg+kDWbLtXWlIvXhhla7q0+RfKb8vu/gXnkXJa6rcJdJztKRbP3b
9fehVeu9m+1+abahjC1zmQimwd2QVc8BGQJADbtfCGaVPzpoho9TWQmaRO1mrYuf
vZh7IiejEYvpHpWNn53cmrTDsTyvti7lG/APYzqYRxeW7M6UOS/+AaLAYQJAJbEW
AwhZPphoB59MO2RzNPXSYyyn4IoEwTSxuz7uy4KG8mXRmyK/a0m6i06rWDLLn8q6
G9jkH/AfO35GP3RiWQJBAJLWBlKpHf8TxT65jAwxBhd9ZOkC2w0WidbSYjX9wkkD
38K7ZDm1LSIR69Ut6tdwotkytXvDniOMPY6ENar5IUs=
-----END RSA PRIVATE KEY-----
```

The content of the sample `insecure_device_cert.crt` file is:

```
-----BEGIN CERTIFICATE-----
MIIB8zCCAVwCAQEwDQYJKoZIhvcNAQELBQAwQjELMAkGA1UEBhMCVVMxDzANBgNV
BAgMBk9yZWdvbjERMA8GA1UEBwwIUG9ydGxhbmQxDzANBgNVBAoMBkdFTklWSTAe
Fw0xNTExMjcyMzE0NTJaFw0xNjExMjYyMzE0NTJaMEIxCzAJBgNVBAYTAlVTMQ8w
DQYDVQQIDAZPcmVnb24xETAPBgNVBAcMCFBvcnRsYW5kMQ8wDQYDVQQKDAZHRU5J
VkkwgZ8wDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBAJtviM8ARIrFquPc0myB9BuF
9MdkA/2SatqbZMWeTOUJHGrjBDEEMLQ7zk8AyBmi7RquYYZs67SyLhylVGKh6sJA
lecxbHUwj7cZSS1bmKMje6L61gKwxBm2NIFU1cVl2jJlTaU9VYhM4xk57yj28nkN
xSYWP1vbFX2NDX2iH7b5AgMBAAEwDQYJKoZIhvcNAQELBQADgYEAhbqVr9E/0M72
9nc6DI+qgqsRSMfoyvA3Cmn/ECxl1ybGkuzO7sB8fGjgMQ9zzcb6q1uP3wGjPioq
MymiYYjUmCTvzdvRBZ+6SDjrZfwUuYexiKqI9AP6XKaHlAL14+rK+6HN4uIkZcIz
PwSMHih1bsTRpyY5Z3CUDcDJkYtVbYs=
-----END CERTIFICATE-----
```

These files are checked into `priv/sample_certifcates` and `priv/sample_keys`.

DO NOT USE THE KEYS AND CERTIFICATES ABOVE IN PRODUCTION!
ANY PRODUCTION KEYS SHOULD BE GENERATED BY THE ORGANIZATION AND BE 4096 BITS LONG.

RVI credentials format

A credential is a JWT-encoded JSON structure, signed by the root X.509 certificate's private key, describing the rights that the sender has. A received RVI credential is validated as follows.

1. **Receive remote party's X.509 device certificate**
   The TLS handshake process will exchange the X.509 certificates setup in the previous chapter.
2. **Validate remote party's X.509 device certificate**
   The received device X.509 certificate has its signature validated by the root X.509 certificate that is pre-provisioned in all RVI nodes.
   The receiver now knows that the remote RVI node has an identiy generated by a trusted provsioning server using the private root key.
3. **Receive one or more RVI credentials**
   Each credential is encoded as JWT, signed by the root X.509 certificate.
4. **Validate each RVI credential signature**
   The root X.509 certificate is used to validate the signature of each received RVI credential.
   A successful validation proves that the certificate was generated by a trusted provisioning server using the private root key.
5. **Validate the credential-embedded X.509 device certificate**
   Each received RVI credential will have its embedded device X.509 certificate compared with the device X.509 certificate received in step 1 above.
   A match proves that the certificate was generated by a trusted provisioning server explictly for the RVI node at the remote end.

An RVI credential has the following format in its native JSON state:

```
{
    "create_timestamp": 1439925416,
    "right_to_invoke": [
        "jlr.com/vin/"
    ],
    "right_to_register": [
        "jlr.com/backend/sota"
    ],
    "id": "insecure_cert",
    "iss": "jaguarlandrover.com",
    "device_cert": "",
    "validity": {
        "start": 1420099200,
        "stop": 1925020799
    }
}
```

The members are as follows:

Member	Description
create_timestamp	Unix timestamp of when the credential was created
right_to_invoke	A list of service prefixes that the sender has the right to invoke on any node that has registered matching services that start with the given string(s).
right_to_register	A list of services that the sender has the right to to register for other nodes to invoke.
id	A system-wide unique identifier for the credential.
iss	The issuing organization.
device_certificate	The PEM-encoded device X.509 certificate to match against the sender's TLS certificate.
validity.start	The Unix timestamps when the credential becomes active.
validity.stop	The Unix timestamps when the credential becomes inactive.

Generating RVI credentials

To create a credential, tie it to a device X.509 certificate, and sign it with a root X.509 certificate private key, the following command is used:

```
rvi_create_credential.py --cred_out="insecure_credential.json" \
                         --jwt_out='insecure_credential.jwt' \
                         --id="xxx" \
                         --issuer="genivi.org" \
                         --root_key=insecure_root_key.pem \
                         --device_cert=insecure_device_cert.crt \
                         --invoke='genivi.org/' \
                         --register='genivi.org/'
```

The following command line parameters are accepted:

Parameter	Required	Description
--cred_out	No	Output file containing the JSON-formatted un-encoded credential.
--jwt_out	Yes	JWT-encoded, JSON-formatted, root keyp-signed credential.
--issuer	Yes	Organization that issued the credential.
--root_key	Yes	Private, PEM-encoded root key to sign the credential. Must be the same key used to sign the root X.509 certificate.
--device_cert	Yes	The PEM-encoded device X.509 certificate to embed into the credential as the device_cert member.
--invoke	Yes	Space separated list (within quotes) of RVI service prefixes that the owner of the credential has the right to invoke.
--register	Yes	Space separated list (within quotes) of RVI service prefixes that the owner of the credential has the right to register for others to call (with the right credential).
--start	No	The Unix timestamps when the credential becomes active.
--stop	No	The Unix timestamps when the credential becomes inactive.

The generated `insecure_credential.json` and `insecure_credential.jwt` are checked into `priv/sample_credentials`.