The Confidential Computing Certifier Framework

**Purpose**

The certifier presents a simple but comprehensive interface to Confidential Computing services that abstracts the idiosyncrasies of individual “enforcement primitives,” and provides a complete set of functions to establish trusted relationships with peer confidential containers.

This unified interface to confidential computing primitives provides access to the following trusted primitives:

1. Measure
2. Seal
3. Unseal
4. Attest
5. Verify attestation

In addition, as described in the API below, the certifier suite also provides policy support. Policy determines whether another protected program should be trusted (i.e.- relied on or supplied with sensitive data). Policy is rooted in a “policy key” for each trust domain. The certifier also permits the expression and enforcement of policy rules (approved by the policy key) that determines the conditions of trust including whether particular hardware should be trusted or whether a particular program (identified by its measurement should be trusted. It also implements functions and formats to express policy (VSE clauses, claims and signed claims, including attestations) and reason about policy statements with a policy engine, the certifier also supplies a policy store to save and protect sensitive data and policy (the policy store) and many support functions to make programming a protected program and enforcing domain policy simple and reliable.

Among these service functions are:

1. Determining whether another confidential container should be trusted
2. Securely exchanging keys with another trusted confidential container
3. Implementing a scalable trust negotiation for efficient, secure, authenticated communications with other confidential containers
4. Supporting upgrade and confidential container movement
5. Provides simple uniform policy management either bilaterally between protected programs or using a policy service
6. Providing secure, authenticated storage and recovery of critical confidential container data and other helper functions.
7. Initializing authentication key and storage keys
8. Generating and distributing a policy-key and the policy about what measurements, hardware and other keys to trust.
9. Evaluating evidence presented by another protected program.

**Components**

The Certifier consists of several programs and libraries written in C++ and Go that offer services from VSE, SGX, SEV and other trust platforms. The programs include:

1. The certifier library, which implements all the certifier API services
2. A certification server that provides a scalable distributed policy service for a trust domain

Currently cryptographic support is implemented using RSA, AES, SHA256, CBC authenticated encryption and HMAC-256, however, certifier data formats are completely general so you could add support for ECC or even quantum resistant crypto.

The certifier code also includes a set of utilities and a complete application which generally requires little change when combined with your very own application code resulting in a secure, protected program functions.

**Source files**

The existing source files are organized as follows:

1. The directory certifier\_prototype contains the C++ certifier.  There are three make files: (a) make -f certifier\_tests.mak makes the automated tests, (b) make -f cert\_utility.mak builds the cert utility which makes up the policy-key, and (c) make -f example\_app.mak makes the sample app.  This directory also contains the protobuf definition file (certifier.proto) used by C++ and Go.
2. certifier\_prototype/domain\_service contains the go service. To build the service, say "go build simpleserver.go".  This directory also contains a test client in Go. To build the test client, say "go build simpleclient.go".   simpleclient also generates test files for the sample application including a self-signed cert for the policy key, the test platform and attest keys and the platform endorsement ["intel says the attestation key is-trusted"]
3. certifier\_prototype/domain\_service/certprotos contains the Go version of the protobuf interface and certifier\_prototype/domain\_service/certlib contains the certifier interface in Go which is used by simpleclient and simpleserver (it could be used by a Go app).  The tests for the GO certifier corresponding to certifier\_tests in C++, can be run by saying "go test" in the certifier\_prototype/domain\_service/certlib directory.

**Certifier service**

While two protected programs can negotiate trust bilaterally using the certifier, it requires that these programs contain the entire policy affecting trust including all the programs measurements and information about all trust enforcement mechanism it can rely on. This is workable but not scalable. So, the certifier also provides a Certification server that can be used to manage policy in a trust domain.

Policy consists of policy key (public/private key) and signed policy claims. Example policy claims are:

* I trust Intel SGX when its statements are signed by the following Intel key
* I trust the following measurements (programs) if performed in the last day
* I trust statements signed by the following Vmware Sphere key between January, 2021 and December, 2021.

The certifier service takes policy expressed as signed claims and makes a “trust decision,” determining whether the requestor is trusted according to the policy rules in a trust domain. If so, it returns a simple X509 certificate that can be used for subsequent secure interactions between trusted programs in a policy domain.

An example of policy statements for the certifier is:

1. The policy key is trusted (policy)
2. The policy key says the intel key is trusted (policy)
3. The intel key says the attestation-key is trusted (Intel signed claim)
4. The attestation says the “application-key” (presented in the attestation) speaks for Measurement-1 (attestation)
5. The policy key says Measurement-1 is trusted (policy)

These statements are used by the certifier to provide a “proof” that a program can be trusted as follows.

1. 1, 2 🡪 intel key is trusted, so the intel key is trusted
2. 3, 6 🡪 the attestation key is trusted
3. 4, 7 🡪 the “application-key” (presented in the attestation) speaks for Measurement-1 (attestation)
4. 5,8 🡪 the application key is trusted
5. Here’s a signed certificate for the application key

This is provided by a sample service called “simpleserver.go” which can be modified to carry out complex policy determinations. Simpleserver contains the full certification engine, but in our sample application, we only use the very simplest (but overwhelmingly most common) certification protocol where the requester provides the service with its platform cert [e.g.- "intel says the attestation key is-trusted"] and the attestation ["the attestation-key says the app-authorization-key speaks-for the program-measurement"].

**Life cycle of a protected program using the certifier**

A protected program has a stylized life cycle after being isolated and loaded (including measured), it first initializes its trust data as indicated in the figure below

****

Next, the program gets “certified” by communicating with the Certifier service. If the evidence presented by the program is adequate, the service issues an authorization certificate saying the protected program’s authorization, or application key is trusted.



Finally using the authorization certificates, two trusted programs can interact securely as indicated in the following diagram.



**Example application**

The sample app exactly mirrors the standard life cycle described above. Its operations are:

1. cold-init which generates all the keys and data the application uses stores them in the "policy\_store" and serializes and seals the policy\_store for later use.
2. warm-restart which recovers the policy\_store.
3. certify-me which contacts the service to get the certificate for the policy-key managed trust domain.
4. run-me-as-a-server and run-me-as-a-client allows running the app as a TLS client and a TLS server which communicate with each other over a trusted channel (TLS with client auth) to exchange data securely.

To run the example application, build it (make -f example\_app.mak) and the certifier server (go build simpleserver.go) as well as simpleclient (go build simpleclient.go). Make up the domain data with the cert\_utility. Run simpleclient (which generates test files).

Next, run simpleserver and rerun simpleclient. You should see a successful negotiation.

Finally, with simpleserver still running run:

* example\_app.exe –operation=cold-init …. (This does app data initialization as above)
* example\_app.exe –operation=warm-start …. (This test data recovery)
* example\_app.exe –operation=certify-me …. (This does certification – you should see an x509 artifact produced )
* example\_app.exe –operation=rune-me-as-server …. (This runs a copy of the application running as a TLS server )
* example\_app.exe –operation=rune-me-as-client …. (This runs a copy of the application running as a TLS client talking to the server )

**The Programming interface**

The programming interface to the certifier is implemented both in Go and C++. External data structures are implemented with protobuf for simple serialization.

The major functions are:

*Crypto and file support*

* **bool write\_file(string file\_name, int size, byte\* data);**
* **int file\_size(string file\_name);**
* **bool read\_file(string file\_name, int\* size, byte\* data);**
* **bool encrypt(byte\* in, int in\_len, byte \*key, byte \*iv, byte \*out, int\* out\_size);**
* **bool decrypt(byte \*in, int in\_len, byte \*key, byte \*iv, byte \*out, int\* size\_out);**
* **bool digest\_message(const byte\* message, int message\_len, byte\* digest, unsigned int digest\_len);**
* **bool authenticated\_encrypt(byte\* in, int in\_len, byte \*key, byte \*iv, byte \*out, int\* out\_size);**
* **bool authenticated\_decrypt(byte\* in, int in\_len, byte \*key, byte \*out, int\* out\_size);**
* **bool asn1\_to\_x509(string& in, X509 \*x);**
* **bool x509\_to\_asn1(X509 \*x, string\* out);**
* **bool make\_root\_key\_with\_cert(string& type, string& name, string& issuer\_name, key\_message\* k);**
* **bool make\_certifier\_rsa\_key(int n, key\_message\* k);**
* **bool rsa\_public\_encrypt(RSA\* key, byte\* data, int data\_len, byte \*encrypted, int\* size\_out);**
* **bool rsa\_private\_decrypt(RSA\* key, byte\* enc\_data, int data\_len, byte\* decrypted, int\* size\_out);**
* **bool rsa\_sha256\_sign(RSA\*key, int size, byte\* msg, int\* size\_out, byte\* out);**
* **bool rsa\_sha256\_verify(RSA\*key, int size, byte\* msg, int size\_sig, byte\* sig);**
* **bool generate\_new\_rsa\_key(int num\_bits, RSA\* r);**
* **bool key\_to\_RSA(const key\_message& k, RSA\* r);**
* **bool RSA\_to\_key(RSA\* r, key\_message\* k);**
* **bool private\_key\_to\_public\_key(const key\_message& in, key\_message\* out);**
* **bool get\_random(int num\_bits, byte\* out);**
* **void print\_bytes(int n, byte\* buf);**
* **void print\_key\_descriptor(const key\_message& k);**
* **void print\_entity\_descriptor(const entity\_message& e);**
* **void print\_vse\_clause(const vse\_clause c);**
* **void print\_claim(const claim\_message& claim);**
* **void print\_signed\_claim(const signed\_claim\_message& signed\_claim);**
* **void print\_storage\_info(const storage\_info\_message& smi);**
* **void print\_trusted\_service\_message(const trusted\_service\_message& tsm);**
* **void print\_attestation(attestation& at);**
* **void print\_protected\_blob(protected\_blob\_message& pb);**
* **bool vse\_attestation(string& descript, string& enclave\_type, string& enclave\_id,vse\_clause& cl, string\* serialized\_attestation);**
* **bool make\_signed\_claim(const claim\_message& claim, const key\_message& key signed\_claim\_message\* out);**
* **bool verify\_signed\_claim(const signed\_claim\_message& claim, const key\_message& key);**
* **bool verify\_signed\_attestation(int serialized\_size, byte\* serialized, int sig\_size, byte\* sig, const key\_message& key);**
* **bool time\_now(time\_point\* t);**
* **bool time\_to\_string(time\_point& t, string\* s);**
* **bool string\_to\_time(const string& s, time\_point\* t);**
* **bool add\_interval\_to\_time\_point(time\_point& t\_in, double hours, time\_point\* out);**
* **int compare\_time(time\_point& t1, time\_point& t2);**
* **void print\_time\_point(time\_point& t);**
* **void print\_entity(const entity\_message& em);**
* **void print\_key(const key\_message& k);**
* **void print\_rsa\_key(const rsa\_message& rsa);**
* **bool produce\_artifact(key\_message& signing\_key, string& issuer\_name\_str, string& issuer\_description\_str, key\_message& subject\_key, string& subject\_name\_str, string& subject\_description\_str, uint64\_t sn, double secs\_duration, X509\* x509);**
* **bool verify\_artifact(X509& cert, key\_message& verify\_key, string\* issuer\_name\_str, string\* issuer\_description\_str,**
* **key\_message\* subject\_key, string\* subject\_name\_str, string\* subject\_description\_str, uint64\_t\* sn);**

*Policy store*

**class policy\_store {**

**public:**

**policy\_store();**

**policy\_store(int max\_trusted\_services, int max\_trusted\_signed\_claims, int max\_storage\_infos, int max\_claims, int max\_keys);**

**~policy\_store();**

**bool replace\_policy\_key(key\_message& k);**

**const key\_message\* get\_policy\_key();**

**int get\_num\_trusted\_services();**

**const trusted\_service\_message\* get\_trusted\_service\_info\_by\_index(int n);**

**int get\_trusted\_service\_index\_by\_tag(string tag);**

**bool add\_trusted\_service(trusted\_service\_message& to\_add);**

**void delete\_trusted\_service\_by\_index(int n);**

**int get\_num\_storage\_info();**

**const storage\_info\_message\* get\_storage\_info\_by\_index(int n);**

**bool add\_storage\_info(storage\_info\_message& to\_add);**

**int get\_storage\_info\_index\_by\_tag(string& tag);**

**void delete\_storage\_info\_by\_index(int n);**

**int get\_num\_claims();**

**const claim\_message\* get\_claim\_by\_index(int n);**

**bool add\_claim(string& tag, const claim\_message& to\_add);**

**int get\_claim\_index\_by\_tag(string& tag);**

**void delete\_claim\_by\_index(int n);**

**int get\_num\_signed\_claims();**

**const signed\_claim\_message\* get\_signed\_claim\_by\_index(int n);**

**int get\_signed\_claim\_index\_by\_tag(string& tag);**

**bool add\_signed\_claim(string& tag, const signed\_claim\_message, to\_add);**

**void delete\_signed\_claim\_by\_index(int n);**

**bool add\_authentication\_key(string& tag, const key\_message& k);**

**const key\_message\* get\_authentication\_key\_by\_tag(string& tag);**

**const key\_message\* get\_authentication\_key\_by\_index(int index);**

**int get\_authentication\_key\_index\_by\_tag(string& tag);**

**void delete\_authentication\_key\_by\_index(int index);**

**bool Serialize(string\* out);**

**bool Deserialize(string& in);**

**void clear\_policy\_store();**

**};**

**void print\_store(policy\_store& ps);**

*Clauses, claims and signed claims*

* **bool same\_key(const key\_message& k1, const key\_message& k2);**
* **bool same\_measurement(string& m1, string& m2);**
* **bool same\_entity(const entity\_message& e1, const entity\_message& e2);**
* **bool same\_vse\_claim(const vse\_clause& c1, const vse\_clause& c2);**
* **bool make\_key\_entity(const key\_message& key, entity\_message\* ent);**
* **bool make\_measurement\_entity(string& measurement, entity\_message\* ent);**
* **bool make\_unary\_vse\_clause(const entity\_message& subject, string& verb, vse\_clause\* out);**
* **bool make\_simple\_vse\_clause(const entity\_message& subject, string& verb,const entity\_message& object, vse\_clause\* out);**
* **bool make\_indirect\_vse\_clause(const entity\_message& subject, string& verb, const vse\_clause& in, vse\_clause\* out);**
* **bool make\_claim(int size, byte\* serialized\_claim, string& format, string& descriptor, string& not\_before, string& not\_after, claim\_message\* out);**

*Policy statement and verification*

* **bool init\_certifier\_rules(certifier\_rules& rules);**
* **bool init\_axiom(key\_message& pk, proved\_statements\* \_proved);**
* **bool init\_proved\_statements(key\_message& pk, signed\_assertions& assertions, proved\_statements\* shown);**
* **bool convert\_attestation\_to\_vse\_clauseconst (claim\_message& sa, vse\_clause\* cl);**
* **bool verify\_signed\_assertion(const key\_message& key, const signed\_claim\_message& sc, vse\_clause\* cl);**
* **bool verify\_external\_proof\_step(proof\_step& step);**
* **bool verify\_internal\_proof\_step(const vse\_clause s1, const vse\_clause s2, const vse\_clause conclude, int rule\_to\_apply);**
* **bool statement\_already\_proved(const vse\_clause& cl, proved\_statements\* are\_proved);**
* **bool verify\_proof(key\_message& policy\_pk, vse\_clause& to\_prove, signed\_assertions& signed\_statements, int num\_steps, local\_proof\_step \*the\_proof, proved\_statements\* are\_proved);**
* **void print\_trust\_response\_message(trust\_response\_message& m);**

**void print\_trust\_request\_message(trust\_request\_message& m);**

*Trusted computing primitives*

* **bool Getmeasurement(string& enclave\_type, string& enclave\_id,int\* size\_out, byte\* out);**
* **bool Seal(string& enclave\_type, string& enclave\_id, int in\_size, byte\* in, int\* size\_out, byte\* out);**
* **bool Unseal(string& enclave\_type, string& enclave\_id, int in\_size, byte\* in, int\* size\_out, byte\* out);**
* **bool Attest(string& enclave\_type, int what\_to\_say\_size, byte\* what\_to\_say, int\* size\_out, byte\* out);**
* **bool Protect\_Blob(string& enclave\_type, key\_message& key, int size\_unencrypted\_data, byte\* unencrypted\_data, int\* size\_protected\_blob, byte\* blob);**
* **bool Unprotect\_Blob(string& enclave\_type, int size\_protected\_blob, byte\* protected\_blob, key\_message\* key, int\* size\_of\_unencrypted\_data, byte\* data);**

**Running the tests and sample App --- details**

After making all the binaries as described in the source file section., you should be able to run all the tests and sample apps. Here, let $CERTIFIER\_PROTOTYPE be the full directory name of the certifier\_prototype directory. Go is very fussy about directory structures so I’m assuming you set those up correctly. You’ll know when you try to compile simpleclient, simpleserver and the tests.

Init the policy and platform keys

c In $CERTIFIER\_PROTOTYPE/domain\_service/certlib

$CERTIFIER\_PROTOTYPE/cert\_utility.exe --operation=generate-policy-key-and-test-keys \

--policy\_key\_output\_file=key\_file.bin --policy\_cert\_output\_file=policy\_cert\_file.bin \

--platform\_key\_output\_file=key\_file.bin --attest\_key\_output\_file=key\_file.bin

Building the go programs

cd $CERTIFIER\_PROTOTYPE

cd domain\_service/certprotos

protoc --go\_opt=paths=source\_relative --proto\_path=. \

--go\_out=. -go\_opt=Mcertifier.proto=./ certifier.proto

cd ../certlib

go test (This runs the certlib tests)

cd ..

go build simpleclient.go

go build simpleserver.go

./simpleclient (This will fail the first time but generate init files)

Running the go tests

In one window at the directory $CERTIFIER\_PROTOTYPE/domain\_service

./simpleserver –simpleclient=true

In one window at the directory $CERTIFIER\_PROTOTYPE/domain\_service

./simpleclient

Running the C++ tests

cd $CERTIFIER\_PROTOTYPE

./certifier\_tests.exe [--print\_all=true]

Running the sample App

cd $CERTIFIER\_PROTOTYPE

make -f example\_app.mak

mkdir app1\_data

mkdir app2\_data

copy the init files generated by simpleclient from $CERTIFIER\_PROTOTYPE/domain\_service/certlib into both app1\_data and app1\_data

In $CERTIFIER\_PROTOTYPE with simpleserver running in another window.

./example\_app.exe --print\_all=true --operation=cold-init \

--data\_dir=./app1\_data/ \

--policy\_cert\_file=policy\_cert\_file.bin \

--policy\_store\_file=policy\_store

./example\_app.exe --print\_all=true --operation=warm-restart \

--data\_dir=./app1\_data/ \

--policy\_cert\_file=policy\_cert\_file.bin \

--policy\_store\_file=policy\_store

./example\_app.exe --print\_all=true --operation=get-certifier \

--data\_dir=./app1\_data/ \

--policy\_cert\_file=policy\_cert\_file.bin \

--policy\_store\_file=policy\_store

./example\_app.exe --print\_all=true --operation=cold-init \

--data\_dir=./app2\_data/ \

--policy\_cert\_file=policy\_cert\_file.bin \

--policy\_store\_file=policy\_store

./example\_app.exe --print\_all=true --operation=warm-restart \

--data\_dir=./app2\_data/ \

--policy\_cert\_file=policy\_cert\_file.bin \

--policy\_store\_file=policy\_store

./example\_app.exe --print\_all=true --operation=get-certifier \

--data\_dir=./app2\_data/ \

--policy\_cert\_file=policy\_cert\_file.bin \

--policy\_store\_file=policy\_store

In $CERTIFIER\_PROTOTYPE in one window:

./example\_app.exe --print\_all=true --operation=run-app-as-server \

--data\_dir=./app2\_data/ \

--policy\_cert\_file=policy\_cert\_file.bin \

--policy\_store\_file=policy\_store

In $CERTIFIER\_PROTOTYPE in another window:

./example\_app.exe --print\_all=true --operation=run-app-as-client \

--data\_dir=./app2\_data/ \

--policy\_cert\_file=policy\_cert\_file.bin \

--policy\_store\_file=policy\_store

**Notes on running the tests**

Generally, specifying the flag –-print\_all=true, will print a lot of interesting information.

For the example app, you’ll see the policy and platform keys, the app key and the generated artifacts. Here’s an example of a self-signed policy key:

Policy Key Certificate:

Data:

Version: 1 (0x0)

Serial Number: 1 (0x1)

Signature Algorithm: sha256WithRSAEncryption

Issuer: CN=policyAuthority, O=root

Validity

Not Before: Oct 21 22:31:06 2021 GMT

Not After : Oct 20 22:31:06 2026 GMT

Subject: CN=policyAuthority, O=root

Subject Public Key Info:

Public Key Algorithm: rsaEncryption

RSA Public-Key: (2048 bit)

Modulus:

00:c0:dd:d1:e4:7c:64:39:b4:27:56:c2:23:48:af:

89:b1:7a:dd:7a:a1:69:b3:83:06:ef:69:83:61:94:

d5:3e:df:e7:a1:fd:13:80:36:e3:06:24:58:42:87:

53:28:08:90:39:c4:21:5f:af:b1:af:96:38:90:1b:

00:52:6c:e2:57:b1:a8:5a:0e:10:e8:db:d1:94:66:

16:bc:54:cb:d7:49:6b:97:29:91:15:18:ca:5a:cf:

66:b3:bc:4b:54:02:5f:73:7e:72:7b:5d:a6:85:27:

54:2a:4f:36:b5:97:60:5d:ab:67:fb:f1:05:f0:8e:

96:a0:1c:de:82:e0:29:ae:6e:c5:9f:a7:96:a5:21:

16:08:3a:71:db:60:1f:a4:5d:56:fd:f5:3b:43:02:

a7:0f:c4:bc:58:54:21:d6:5f:8d:66:b5:d2:9b:0a:

d3:d5:2c:dc:60:17:91:ce:4c:f3:85:39:10:2a:1f:

4f:0d:68:77:5c:63:31:83:0e:ce:88:e1:4b:14:6f:

f8:bf:25:c8:bc:26:40:9d:87:28:a9:4c:76:b8:ce:

74:af:c2:c3:73:05:48:48:3d:6a:a5:af:e1:ee:27:

62:85:e0:ad:f8:cf:f2:ba:71:35:c1:82:d7:d2:ea:

10:0a:26:2f:0c:b2:0c:75:79:93:a3:6a:e2:44:41:

5b:b3

Exponent: 65537 (0x10001)

X509v3 extensions:

X509v3 Key Usage: critical

Digital Signature, Certificate Sign, CRL Sign

X509v3 Extended Key Usage:

TLS Web Client Authentication, TLS Web Server Authentication

X509v3 Basic Constraints: critical

CA:TRUE

Signature Algorithm: sha256WithRSAEncryption

56:54:60:e6:e5:48:fa:c5:1b:55:3c:93:2f:e9:f9:92:7c:bf:

ae:13:54:21:b4:7a:2a:ff:45:5d:9a:58:8b:12:5b:a4:9b:3f:

19:5b:95:42:d6:c1:61:41:c8:88:4a:99:3c:62:27:d7:01:2a:

1a:88:89:7c:5b:f4:06:ac:46:84:34:51:f9:73:44:32:0b:44:

b7:3d:a6:12:c5:0a:7d:e9:99:41:c7:8f:24:2f:aa:36:64:4b:

ed:72:00:4c:e2:a1:6a:7f:40:80:af:e0:0e:b8:57:1c:f5:4b:

f4:07:c7:3b:92:c1:6f:da:67:a6:df:0a:ee:9e:0b:72:d0:d0:

36:c0:b6:5b:4a:7a:82:d4:73:ca:73:b7:9c:de:82:a4:43:b4:

d9:4a:33:66:d5:ca:4e:e9:11:fa:5b:e8:32:9c:40:16:1f:b3:

f2:c6:02:c6:d4:c9:ba:24:02:27:b9:12:90:25:e3:fd:aa:42:

c7:d2:b2:66:bd:0d:36:e6:e7:80:5a:eb:a7:12:ab:3e:de:dd:

98:a6:e9:ad:39:ee:e8:26:7b:1a:ee:8b:34:08:f7:09:6f:47:

98:86:9a:65:11:7c:70:cc:db:e4:30:1d:6d:30:a9:19:84:3a:

2d:8d:4f:40:6f:43:5b:9e:c8:27:1b:d6:e8:d7:af:f0:34:32:

71:49:ef:f2

During “certify-me”, the service produces an artifact, an X509 certificate signed by the policy key, that the program uses for future authentication. There’s an example below. Note the subject name in the certificate contains both server DNS name (localhost, in the example), which is required to open TLS channels and a descriptive name including the app measurement.

Application Certificate:

Data:

Version: 3 (0x2)

Serial Number: 1634855632022943002 (0x16b02c8915c6351a)

Signature Algorithm: sha256WithRSAEncryption

Issuer: CN=policyAuthority, O=root

Validity

Not Before: Oct 21 22:35:28 2021 GMT

Not After : Oct 21 22:35:28 2022 GMT

Subject: O=Measured-000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f, CN=localhost

Subject Public Key Info:

Public Key Algorithm: rsaEncryption

RSA Public-Key: (2048 bit)

Modulus:

00:d5:9e:4c:eb:61:bb:69:38:23:c0:65:66:7d:ed:

95:f4:34:e2:6d:39:9a:e6:0c:5a:c0:85:85:88:5e:

80:c9:4f:0c:84:64:45:5f:9e:61:1c:c9:1f:29:ad:

b8:d3:2a:fb:08:ff:03:37:f9:b2:74:06:57:9b:45:

68:dc:f8:40:e0:27:00:64:a6:b4:fd:f4:2a:42:fa:

f0:3b:58:50:c3:5e:10:fd:e5:c4:d2:32:37:c5:b8:

59:63:7b:a1:11:05:36:ab:f3:10:16:ec:cb:32:23:

34:17:cd:7d:1a:a2:1f:39:26:70:0d:d7:80:f8:53:

78:f8:97:da:e6:a4:72:5c:0c:ef:7e:a8:37:73:b3:

49:f3:a0:d7:e0:de:aa:89:40:40:11:b1:8f:b7:a7:

ad:2c:25:72:84:0f:a0:d5:3f:c1:90:20:97:1b:93:

ef:38:a8:b2:e5:6d:22:bc:45:44:4d:7c:c3:a9:e8:

d9:88:e3:26:9e:d7:35:38:b0:38:c7:83:22:28:76:

df:eb:80:67:8f:67:ed:4f:ee:e9:22:78:1a:29:92:

d5:a9:8f:21:d0:cc:6a:2d:af:0f:e8:c1:e8:76:3a:

32:0b:15:5b:91:fe:c1:d2:62:f2:94:82:bf:68:d7:

1a:65:f8:e3:6b:d7:60:36:05:86:89:50:38:8e:4c:

3f:0f

Exponent: 65537 (0x10001)

X509v3 extensions:

X509v3 Key Usage: critical

Digital Signature, Certificate Sign

X509v3 Extended Key Usage:

TLS Web Client Authentication, TLS Web Server Authentication

X509v3 Basic Constraints: critical

CA:FALSE

Signature Algorithm: sha256WithRSAEncryption

34:47:cf:55:b2:60:94:55:78:24:65:be:72:f3:6d:fd:29:7e:

ac:71:c3:3b:80:d0:c9:2c:4f:ff:5d:33:ff:82:80:2d:30:76:

3e:43:6f:1d:35:7c:4a:d9:fd:63:54:8a:80:11:ff:13:4f:d2:

52:a6:78:cf:c0:d3:29:23:e4:6c:01:b5:04:1d:b3:61:a8:9b:

d1:7c:47:c3:09:26:48:b4:46:17:85:ee:59:01:33:93:1e:ba:

4d:f9:b5:dd:f9:66:0a:33:82:81:67:65:69:bb:d9:5f:2d:48:

f6:28:56:1a:7c:bf:26:fc:e8:5d:65:ac:b6:a7:49:aa:46:8b:

d6:6d:1d:0e:dd:aa:95:15:00:ff:41:45:39:7c:a0:47:80:4d:

76:97:00:d5:74:92:4d:5e:16:5c:65:e6:c7:c4:bb:36:68:37:

58:c7:47:48:4e:9d:d3:05:96:5b:ef:14:e2:55:20:d9:49:f5:

08:ba:b5:2b:44:59:2d:3c:3f:c8:a6:25:21:e9:99:5d:37:f2:

ca:0a:2a:af:70:a1:e0:52:3d:05:8c:a0:2f:4d:49:26:12:27:

f4:2f:83:8e:8a:80:91:08:c1:23:9a:ac:39:48:ef:36:81:fa:

31:59:c0:a8:8d:12:94:16:90:18:67:b8:c5:ab:79:d7:9a:11:

c2:56:6f:96

Have fun!