Certificate Revocation check

证书签发流程

- 使用**Kse**（KeyStore Explorer）创建根密钥对---> rootCAKeyPair
- 每个节点使用Kse各自创建各自的节点密钥对--->endPointKeyPair
- 各个节点使用新创建的密钥对，生成CSR---> CertificateSigningRequest
  - 可使用Kse来生成csr
  - 也可使用代码来生成csr
- **根密钥对**对各个节点生成的CSR分别进行签名并生成证书（endPointCert）或CAReply（后缀名为p7r的证书链），在此过程，要将crl与ocsp的url写入到证书扩展中，分别是cdp（**CRL分发点**）与aia（**颁发机构信息访问**）
- 各个节点收到根节点签发的证书或者CAReply，使用kse导入到包含有密钥对的keyStore中
- 使用kse将各个节点将根证书导入到trustKeystore中

以上：

- 目前可通过代码实现的有：1、创建CSR；2、基于CSR签发证书（同时将crl与ocsp的url写入到证书扩展）或生成CAReply；3、生成CRL并写入到文件（pem）
- 可通过代码实现的有（未整理，可参考其他文档）：1、生成密钥对
- 可通过kse实现的有：1、创建CSR；2、基于CSR签发证书（只能将ocsp的url写入到证书扩展，crl目前不行）；

Java 证书吊销检查流程

- 开启证书吊销检查

```
-Dcom.sun.security.enableCRLDP=true
-Dcom.sun.net.ssl.checkRevocation=true
```

```
System.setProperty("com.sun.net.ssl.checkRevocation", "true")
System.setProperty("com.sun.security.enableCRLDP", "true")
Security.setProperty("ocsp.enable", "true")
```

- 开启调试模式

```
-Djava.security.debug=certpath
```

- `RevocationChecker`
  - **默认**
    - 开启ocsp

      > 会优先通过ocsp来检查证书状态

    - 不开启ocsp

      > Security.setProperty("ocsp.enable", "false")，会通过crl来检查证书状态

  - **自定义Checker**
    - 优先检查ocsp

      > 默认是ocsp优先，通过向证书AIA中ocsp服务的的url发送请求（content-type : "application/ocsp-request" ）

    - 优先检查crl

      > **可设置**优先检查crl
      >
      > - **先检查是否有自定义设置的crl列表（初始化在CertStore中）**，如果没有的话，检查证书cdp，向url发送请求，下载crl列表
      >
      >   > Java中如果是通过url下载crl列表，默认是30s内缓存，超过30s之后再次下载最新

```
val cpb: CertPathBuilder = CertPathBuilder.getInstance("PKIX") // 使用CertPathValidator也可以
val rc: PKIXRevocationChecker = cpb.getRevocationChecker.asInstanceOf[PKIXRevocationChecker]
rc.setOptions(util.EnumSet.of(
    // prefer CLR over OCSP，设置CRL优先
    PKIXRevocationChecker.Option.PREFER_CRLS,
    PKIXRevocationChecker.Option.ONLY_END_ENTITY
    // don't fall back to OCSP checking，CRL校验失败后，是否进行ocsp检查
    // PKIXRevocationChecker.Option.NO_FALLBACK
  ))
override protected def trustManagers: Array[TrustManager] = {
    val trustManagerFactory = TrustManagerFactory.getInstance(TrustManagerFactory.getDefaultAlgorithm)
    val ts = loadKeystore(SSLTrustStore, SSLTrustStorePassword)

    val pkixParams = new PKIXBuilderParameters(ts, new X509CertSelector)
    pkixParams.addCertPathChecker(rc)
    // 获取初始化CRL
    val crl: X509CRL = CustomSSLEngine.generateCRL(new FileInputStream(SSLTrustCrl))
    crlList.add(0, crl)
    val collectionCertStoreParameters = new CollectionCertStoreParameters(crlList)
    // 初始化CRL到CertStore中
    pkixParams.addCertStore(CertStore.getInstance("Collection", collectionCertStoreParameters))
    pkixParams.setRevocationEnabled(true)
```

```
    trustManagerFactory.init(new
CertPathTrustManagerParameters(pkixParams))
    trustManagerFactory.getTrustManagers
}
```

从代码中可以看到有几个点：

1. 首先生成`PKIXRevocationChecker`，为`PKIXRevocationChecker`设置校验选项：优先检查crl
2. `pkixParams`设置初始化crl到CertStore，其中使用的是`collectionCertStoreParameters`，**该集合非copy，后续可以更改**
3. `pkixParams.setRevocationEnabled(true)`，默认为true，这里可以设置为false，则不执行吊销检查

- **假设现在是从头开始组网**，那么没有区块，也没有leveldb，也就不能从leveldb中检索crl或者ocsp-response，如果使用这两者作为revocationCheck的话，组网肯定会失败，**因此**，最好有个初始化，上面提到的CertStore就可以作为一个初始化入口，可将初始化的crl写入，**然后初始组网成功之后**：1、通过合约写一份crl到leveldb中，CertStore中的crl可通过合约来删除，或者使用定时器自动删除；2、通过合约将staticOcspResponse写到leveldb中；

> 假设CertStore中的crl被删除之后，crl与ocsp-response也都写入到leveldb了，那么之后一切都交给合约来管理了

- 校验顺序

  允许fallback

  - PREFER_OCSP
    - 先检查ocsp
    - ocsp失败之后（**非revoked**），检查crl
      - CertStore中有crl，检查
      - CertStore中没有crl，检查crldp
  - PREFER_CRLS
    - 先检查crl
      - CertStore中有crl，检查
      - CertStore中没有crl，检查crldp
    - crl检查失败之后（**非revoked**），检查ocsp

  从上面可以看出，不论是哪种顺序，都要初始化crl到CertStore中，并且之后要通过合约将crl与ocsp-response写到leveldb中

RepChain集成证书吊销检查

- 方案设计：
  - 各个节点通过Akka-http来下载crl列表：http://127.0.0.1:8081/repchain.crl
    - 需要新增接口，提供crl列表下载
    - 要预先写入到leveldb中
  - 各个节点向akka-http来发送ocsp请求：http://127.0.0.1/ocsp
    - 需要新增接口，可以解析ocsp请求，并可回复ocsp响应
    - 要预先写入staticResponse到leveldb中

让AkkaHttp支持ocsp

- akka默认预定义支持的content-type不包括`application/ocsp-request`与`application/ocsp-response`，因此需要在akkaHttpServer注册这两种类型，Registering Custom Media Types

```
import akka.http.scaladsl.settings.ParserSettings
import akka.http.scaladsl.settings.ServerSettings

val `application/ocsp-request`: Binary = MediaType.applicationBinary("ocsp-
request", NotCompressible)
val `application/ocsp-response`: Binary = MediaType.applicationBinary("ocsp-
response", NotCompressible)
// add custom media type to parser settings:
val parserSettings =
ParserSettings(system).withCustomMediaTypes(`application/ocsp-
request`).withCustomMediaTypes(`application/ocsp-response`)
val serverSettings =
ServerSettings(system).withParserSettings(parserSettings)

Http().bindAndHandle(
      route_evt
        ~ cors() (
          new BlockService(ra).route ~
            new ChainService(ra).route ~
            new TransactionService(ra).route ~
            new CrlService(ra).route ~
            new OcspService(ra, sys).route ~
            SwaggerDocService.routes),
      "0.0.0.0", port, settings = serverSettings)
```

- 服务端是支持了，但是接口还不知道如何解析请求并返回响应，因此接口端要定义如何`Marshalling`与`Unmarshalling`

```
val `application/ocsp-request` = MediaType.applicationBinary("ocsp-request",
NotCompressible)
val `application/ocsp-response` = MediaType.applicationBinary("ocsp-
response", NotCompressible)

implicit val unmarshaller: FromEntityUnmarshaller[OCSPReq] =
Unmarshaller.byteStringUnmarshaller.forContentTypes(`application/ocsp-
request`).map(byteString => new OCSPReq(byteString.toArray))

implicit val marshaller: ToEntityMarshaller[OCSPResp] =
Marshaller.withFixedContentType(`application/ocsp-response`) {
    ocspResp => HttpEntity(`application/ocsp-response`,
akka.util.ByteString(ocspResp.getEncoded))
   //ocspResp => HttpEntity.Strict(`application/ocsp-
response`,akka.util.ByteString(ocspResp.getEncoded))
  }
```

其他解析ocsp-request（在RestService中）：

```
implicit val unmarshaller: FromRequestUnmarshaller[OCSPReq] =
Unmarshaller.strict[HttpRequest, OCSPReq](request => {
    new
OCSPReq(Await.result(request.entity.dataBytes.runFold(ByteString.empty)(_ ++
_), timeout.duration).toArray)
})
```

ocsp-response（在RestService中）：

```
implicit val marshaller: ToResponseMarshaller[OCSPResp] =
Marshaller.withFixedContentType(`application/ocsp-response`) {
    ocspResp =>  HttpResponse(entity = HttpEntity.Strict(`application/ocsp-
response`, akka.util.ByteString(ocspResp.getEncoded)))
  }
```

如果不是用marshaller或者unmarshaller的话，使用如下方式（在RestService中）：

```
post {
    extractRequest { req =>
//  val ocspReqFuture: Future[Seq[ByteString]] =
req.entity.dataBytes.runWith(Sink.seq[ByteString])
//  val ocspByte = Await.result(ocspReqFuture,
Timeout(3.seconds).duration).foldLeft(ByteString.empty)(_++_).toArray
        val ocspReqFuture: Future[ByteString] =
req.entity.dataBytes.runFold(ByteString.empty)(_ ++ _)
        val ocspByte = Await.result(ocspReqFuture, timeout.duration).toArray
        // get request info
        val ocspRequest = new OCSPReq(ocspByte)
// val requestCerts = ocspRequest.getCerts.map(new
JcaX509CertificateConverter().getCertificate(_))
        val requestList = ocspRequest.getRequestList
// println("************" + requestCerts(0))
        println(requestList)
        complete(req.entity.contentType.toString + " = " +
req.entity.contentType.getClass)
//          complete((ra.getRestActor ?
OcspQuery(ocspRequest)).mapTo[HttpResponse])
    }
}
```

构建OCSP响应

目的主要是通过构建静态的ocsp-response，并通过合约将其写入到leveldb中，这样接口接到ocsp-request之后就可以从leveldb中检索并构造ocsp-response，进一步返回响应

- 构建staticOcspResponse，初始时都设置为SUCCESSFUL，且证书状态为GOOD，使用CA的私钥先构造好OCSP-response（利用bouncycastle工具包）

```
CertificateStatus nodeGoodStatus = CertificateStatus.GOOD;
// 准备好创建 OCSP 所需的私钥和证书
KeyStore caKs = KeyStore.getInstance("JKS");
caKs.load(new FileInputStream("jks/trust.jks"), "changeit".toCharArray());
X509Certificate caCertificate = (X509Certificate)
caKs.getCertificate("trust");
```

```
PrivateKey caPrivateKey = (PrivateKey) caKs.getKey("trust",
"changeit".toCharArray());

// 构建响应Id
RespID respID = new JcaRespID(caCertificate.getSubjectX500Principal());
BasicOCSPRespBuilder responseBuilder = new BasicOCSPRespBuilder(respID);
ContentSigner signer = new
JcaContentSignerBuilder("SHA256withECDSA").setProvider("BC").build(caPrivate
Key);
// Generate the id for the certificate we are looking for
CertificateID certificateID = buildCertificateID(serialNumber);
responseBuilder.addResponse(certificateID, certificateStatus, new Date(),
new Date(System.currentTimeMillis() + 10 * 365 * MILLIS_PER_DAY), null);

BasicOCSPResp ocspResponse = responseBuilder.build(signer, new
X509CertificateHolder[]{new JcaX509CertificateHolder(caCertificate)}, new
Date());

OCSPRespBuilder ocspResponseBuilder = new OCSPRespBuilder();
OCSPResp ocspResp = ocspResponseBuilder.build(OCSPRespBuilder.SUCCESSFUL,
ocspResponse);
```

- 将response转为16进制或者pem格式，然后通过调用合约写到leveldb中

```
// 使用十六进制进行保存传输
String ocspRespHexString = Hex.encodeHexString(ocspResp.getEncoded());
System.out.println(ocspRespHexString);
OCSPResp rebackOcspResp = new
OCSPResp(Hex.decodeHex(ocspRespHexString.toCharArray()));
System.out.println("返回来了" + ((BasicOCSPResp)
rebackOcspResp.getResponseObject()).getProducedAt());

// 使用pem保存传输
StringWriter stringWriter = new StringWriter();
JcaPEMWriter pemWriter = new JcaPEMWriter(stringWriter);
pemWriter.writeObject(new PemObject("OCSP RESPONSE",
ocspResp.getEncoded()));
pemWriter.close();
String ocspRespPem = stringWriter.toString();
System.err.println(ocspRespPem);

StringReader stringReader = new StringReader(ocspRespPem);
PemReader pemReader = new PemReader(stringReader);
PemObject pemObject = pemReader.readPemObject();
pemReader.close();
OCSPResp rebackOcspRespPem = new OCSPResp(pemObject.getContent());
System.err.println(rebackOcspRespPem);
```

智能合约

- 删除CertStore中初始化的crl
- 写入或更新crl到leveldb中
- 写入或更新staticOcspResponse到leveldb中

接口ocsp-response的响应逻辑

- 接口接收到ocsp-request之后，首先拿到ocspRequest的certId，然后解析出证书序列号
- 通过证书序列号到leveldb中检索，看是否有对应的ocsp-response（good/revoked）
- 如果有就直接返回，如果没有，则返回一个`"UNAUTHORIZED"`的ocsp-response

```
val ocspRespBuilder = new OCSPRespBuilder
// 如果写入TryLater，就不会failover
var ocspResp = ocspRespBuilder.build(OCSPRespBuilder.UNAUTHORIZED, null)
if (! ocspReq.getRequestList.isEmpty) {
    val req: Req = ocspReq.getRequestList.repr(0)
    val pkey = WorldStateKeyPreFix + "ManageNodeCert" + "_" +
"ocsp_"+req.getCertID.getSerialNumber.toString
    val pvalue = sr.Get(pkey)
    if (pvalue != null) {
      ocspResp = SerializeUtils.deserialise(pvalue).asInstanceOf[OCSPResp]
      val test = SerializeUtils.deserialise(pvalue)
      println(ocspResp)
    }
}
```

参考文献

- java-pki-programmers-guide：https://docs.oracle.com/en/java/javase/11/security/java-pki-programmers-guide.html#GUID-5404B79C-3D49-4668-974C-1BACD1A98B73
- Java Security Standard Algorithm Names：https://docs.oracle.com/en/java/javase/11/docs/specs/security/standard-names.html#trustmanagerfactory-algorithms
- 专门做吊销检查服务的--->revoker：https://github.com/wdawson/revoker
- AkkaHttp
  - Registering Custom Media Types：https://doc.akka.io/docs/akka-http/10.1.11/common/http-model.html#registering-custom-media-types
  - Custom Marshallers：https://doc.akka.io/docs/akka-http/10.1.11/common/marshalling.html#custom-marshallers
  - Custom Unmarshallers：https://doc.akka.io/docs/akka-http/10.1.11/common/unmarshalling.html#custom-unmarshallers
  - Akka HTTP Circe Custom Marshaller and Unmarshaller：https://gist.github.com/mattroberts297/c531a4e9e525d6a18cbf8889ab5f3dec
- openssl
  - OpenSSL 通过OCSP手动验证证书：https://www.cnblogs.com/penghuster/p/6895714.html
  - 使用openssl检测证书ocsp吊销状态：https://yryz.net/post/openssl-ocsp-test-certificate-revocation/
- 其他
  - Java 的 X.509 证书吊销检查：https://zhuanlan.zhihu.com/p/78513242
  - 常见的一些扩展名介绍：https://www.cnblogs.com/bjlhx/p/6565340.html
- PKCS #10: Certification Request Syntax Specification：https://tools.ietf.org/html/rfc2986、https://www.rfc-editor.org/info/rfc2986
- X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP：https://www.rfc-editor.org/rfc/rfc6960.html
- Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile：https://tools.ietf.org/html/rfc5280

- 证书后缀名区别：https://www.huaweicloud.com/articles/2a9f5e8dd7d547f24fbf69dd5d270ea9.html
- 常见的数字证书格式与格式转换：https://www.cnblogs.com/cioliuguilan/p/5525845.html
- ietf工具
  - https://www.rfc-editor.org/info/rfc6960
  - https://datatracker.ietf.org/wg/pkix/charter/
  - https://www.rfc-editor.org/search/rfc_search_detail.php

附录

Windows CertUtil调试ocsp与crl

- 使用windows自带的certutil可以用来调试证书中自带的ocsp与crl连接，与ide结合更佳，可以调试akka-http是否可以正常接收并解析ocsp请求，以及是否可以正常响应

```
$ certutil -URL ./12110107bi45jh675g.node2.cer
```

查看crl

- openssl

```
$ openssl crl -in trust-init.crl -text
```

- keytool

> jdk自带的keytool可以查看crl信息

```
$ keytool -printcrl -file trust-init.crl
```

查看证书的ocsp与crl连接

- openssl

```
$ openssl x509 -in  121000005l35120456.node1.cer -noout -text
```

- keytool

```
$ keytool -printcert -file  121000005l35120456.node1.cer
```