Bouncy Castle FIPS Java API

1 May 2015

1. Introduction

The Bouncy Castle APIs (BC) divide into 3 groups: there is a light-weight API which provides direct access to cryptographic services, a JCA/JCE provider built on top of the light-weight API that provides access to services required to use the JCA/JCE, and another set of APIs which provide handling of protocols such as Cryptographic Message Syntax (CMS), OpenPGP, Time Stamp Protocol (TSP), Secure Mime (S/MIME), Certificate Management Protocol (CMP), as well as APIs for generating Certification Requests (CRMF, PKCS#10), X.509 certificates, PKCS#12 files and other protocol elements used in a variety of standards. The total code base, including porting code for different JVMs, is currently sitting at 499,000 lines of Java.

In terms of an overall design the APIs have been put together in a manner that allows the different classes defined to be used to create objects which can be assembled in a variety of ways, regardless of whether it makes sense. This has given us a very agile API to work with, in the sense that it is easy to formulate new combinations of algorithms, modes, and padding types, as well as to define a wide variety of things like signature types. However the boundaries are loosely defined, and the agility results in widespread leakage – FIPS style programming requires specific paths to access cryptographic functions with defined boundaries between what's in the cryptographic world and what is not. For the purposes of many application developers using BC, the BC approach is fine, or they are simply not aware of the potential issues as they only work at the JCA/JCE level or even at a higher level such as CMS or OpenPGP.

A lot of BC users are now also starting to move into areas that require (or at least are being told to require) FIPS certification. The vast majority of these are interacting with BC cryptographic services at the level of the JCA/JCE or above. There are also some who need to be able to use something along the lines of the light-weight API as they are developing and deploying Java Applet and Java Webstart applications which require strong cryptography without requiring end users to install the unrestricted policy files. It is also interesting to note that for some users they do not necessarily need to be able to use the APIs in FIPS approved mode, they just want the assurance that any NIST defined algorithm they are using is FIPS certified and it is possible to move into a FIPS approved mode should they choose to. It seems that all users that are interested in this are able to use JDK 1.5 or later so we have set JDK 1.5 as the base level for the APIs.

It should be noted that FIPS is not actually written with the purpose of agility in mind, and there is a probably a good case that it should not be. That being said, a review of “Implementation Guidance for FIPS PUB 140-2 and the Cryptographic Module Validation Program” (Jan 17, 2014 update), has indicated there are a number of boundary issues in the current BC APIs in addition to the lack of self tests.

1.1 The Proposed Approach

Originally, we imagined that it would just be necessary to provide an API for the FIPS algorithms supported by BC and have people use the regular provider to fill in the gaps. Unfortunately, due to both the operational requirements of the JCE and the fact that people using approved mode would also need APIs for EC math and ASN.1 to be available we realised that this would also mean making parallel APIs for the BCPKIX, BCMAIL, and BCPG jars as well. To this end we are trying to make the FIPS

algorithms available along side a set of non-approved mode algorithms which can be disabled – the ambition being that it will be straightforward to build the other BC APIs against either provider.

Our proposal for solving this problem is to define a FIPS version of the APIs with a new light-weight API split into two parts, a FIPS approved part and a non-approved part, and to introduce two facades, one set at the light-weight level, and one set at the JCA/JCE level, and to hide a majority of the actual working classes from the non-FIPS version of BC via Java renaming and package protection. The set of facades at the JCA/JCE level is primarily about preventing developer confusion; as the JCA/JCE level is built on the light-weight level an object carrying out an unapproved service cannot be constructed, or used. The facades will allow us to prevent the creation of objects performing unapproved tasks.

For approved/non-approved mode operation we have assumed the defining line is a thread. By default on start up the code will assume approved mode unless it is told otherwise.

Start-up validations will be done using static methods invoked on class loading, the implementation classes will be unusable if the validations do not pass. Other validations, such as key checking, and PRNG health tests are being added to the FIPS versions of the APIs as appropriate.

Zeroization of key material is managed via the JVM's garbage collection process. Proactive zeroisation and finalisers are also used where appropriate.

1.2 Current Progress and Sponsors

For us, FIPS certification involves 3 steps, product review, documentation review, and final testing. Currently we have completed a product review, a documentation review, and we are now in preparation for final testing.

We gratefully acknowledge that the progress of this work would not have been possible without external funding. Currently the work has been largely sponsored by:

http://www.orionhealth.com

http://www.cryptoworkshop.com

http://www.galois.com

http://www.jscape.com

Crypto Workshop would also like to acknowledge that its contribution has been made possible through

its clients purchasing Bouncy Castle support agreements.

We are looking for additional sponsorship to cover the final testing.

Please contact us at office@bouncycastle.org if you are interested in helping sponsor this work.

2. Outline of the new API

The low level FIPS API is currently 6 packages.

The low level ones:

org.bouncycastle.crypto – a set of interfaces sitting over the top of the FIPS approved, and non-FIPS approved (general) APIs, plus the CryptoServicesRegistrar.

org.bouncycastle.crypto.asymmetric – classes containing objects for implementing keys and domain parameters for the public/private key algorithms used in the FIPS and general APIs.

org.bouncycastle.crypto.fips – classes for creating implementations of FIPS approved cryptographic services.

org.bouncycastle.crypto.general – classes for creating implementations of the non-FIPS approved cryptographic services.

org.bouncycastle.crypto.internal – classes for internal use by the FIPS and general packages, that either must be shared or are shared for the purpose of avoiding common code.

We expect we will also add the core of the BC ASN.1 library and the BC utils library (including the Base64 and Hex encoders), but the the rest of the operational code will be copies of existing BC classes renamed and repackaged so as to be invisible outside of where they are being used.

There is also:

org.bouncycastle.jcajce.provider – the classes for supporting the JCA/JCE provider. This has only one publicly visible classes in it: BouncyCastleFipsProvider

Low Level Public Access Points – Service Creation

The low level access points for creating cryptographic services are in org.bouncycastle.crypto.fips and org.bouncycastle.crypto.general.

The org.bouncycastle.crypto.fips Package

The public access classes are named for the algorithms they represent:

FipsAES – AES service creation and algorithm definitions

FipsDRBG – Determinstic random bit generator service creation.

FipsDSA – DSA service creation and algorithm definitions.

FipsDH – Diffie-Hellman key agreement service creation and algorithm definitions.

FipsEC – Elliptic Curve service creation and algorithm definitions.

FipsKDF – KDFs: Counter Mode, Feedback Mode, Double Pipeline Mode, TLS (V1.0/1.1/1.2), X9.63, and SSH.

FipsPBKD – PBE key and IV generation.

FipsRSA – RSA service creation and algorithm definition.

FipsSHS – FIPS Secure Hash Standard service creation and algorithm definition.

FipsTripleDES – Triple DES service creation and algorithm definition.

FipsX931PRNG – Pseudo random bit generator (based on X9.31) service creation.

All other classes with the exception of FipsUnapprovedOperationException have package protected constructors or are not visible outside the package.

The org.bouncycastle.crypto.general Package – a collection of non-FIPS approved algorithms which are widely used in IETF and other standards.

2.1 Proposed Algorithm Set

2.1.1 FIPS Approved

FIPS approved algorithms (note: not all variations are FIPS approved):

Algorithm	Variation
AES	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CFB, CFB8, CFB16, CFB32, CFB64, OFB, OFB8, OFB16, OFB32, OFB64, CTR, CCM, GCM, OCB, EAX, CMAC, WRAP
DRBG/PRNG	SP800-90 (CTR, Hash, HMAC), X9.31 (AES TDES)
DSA	DSA (SHA-1 SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224. SHA-512/256)
DH	DH
EC	ECDSA (SHA-1 SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224. SHA-512/256) ECMQV, ECDH (SSL only), ECCDH

Algorithm	Variation
RSA	PKCS1.5 Signing (SHA-1 SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224. SHA-512/256) RSAPSS (SHA-1 SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224. SHA-512/256), X9.31-1988 (SHA-1 SHA-256 SHA-384 SHA-512), PKCS 1.5 Key Wrap (TLS only), OEAP Key Wrap, SVE.
SHS	SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224. SHA-512/256, HMAC-SHA-1, HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, HMAC-SHA-512, HMAC-SHA-512/224. HMAC-SHA-512/256
TripleDES	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CFB, CFB8, CFB16, CFB32, OFB, OFB8, OFB16, OFB32

2.1.2 General Algorithms

General algorithms are composed of algorithms which are commonly accepted as sound, but are not FIPS approved. If the jar is used in approved mode these algorithms will be unavailable, but they will be available if the jar is appropriately configured.

General message digest algorithms:

MD5, HMAC-MD5, GOST3411, RIPEMD128, HMAC-RIPEMD128, RIPEMD160, HMAC-RIPEMD160, RIPEMD256, HMAC-RIPEMD256, RIPEMD320, HMAC-RIPEMD320, TIGER, HMAC-TIGER, WHIRLPOOL, HMAC-WHIRLPOOL, SipHash.

General PBE algorithms: PKCS#12, PKCS#5 scheme 1. KDFs SCRYPT.

General encryption algorithms:

Algorithm	Variation
AES	OpenPGPCFB
Blowfish	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CFB, CFB8, CFB16, CFB32, OFB, OFB8, OFB16, OFB32, OpenPGPCFB

Algorithm	Variation
Camellia	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CFB, CFB8, CFB16, CFB32, CFB64, OFB, OFB8, OFB16, OFB32, OFB64, CTR, CCM, GCM, OCB, EAX, CMAC, WRAP, OpenPGPCFB
CAST5	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CFB, CFB8, CFB16, CFB32, OFB, OFB8, OFB16, OFB32, OpenPGPCFB
DES	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CFB, CFB8, CFB16, CFB32, OFB, OFB8, OFB16, OFB32, OpenPGPCFB
DH	X9.42, RFC 2631
DSTU4145	GOST3411
ElGamal	RAW, PKCS1v1.5, OAEP
GOST28147	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CFB, CFB8, CFB16, CFB32, OFB, OFB8, OFB16, OFB32, OpenPGPCFB, GCFB, GCTR
GOST3410	GOST3411
ECGOST3410	GOST3411
IDEA	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CFB, CFB8, CFB16, CFB32, OFB, OFB8, OFB16, OFB32, OpenPGPCFB
ARC4 (RC4)	RFC 6229

Algorithm	Variation
RC2	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CFB, CFB8, CFB16, CFB32, OFB, OFB8, OFB16, OFB32
RSA	RAW, PKCS1v1.5, OAEP, other digest signers.
SEED	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CFB, CFB8, CFB16, CFB32, CFB64, OFB, OFB8, OFB16, OFB32, OFB64, CTR, GCM, OCB, EAX, CMAC
TripleDES	OpenPGPCFB
Twofish	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CFB, CFB8, CFB16, CFB32, CFB64, OFB, OFB8, OFB16, OFB32, OFB64, CTR, CCM, GCM, OCB, EAX, CMAC, OpenPGPCFB

2.2 Use of Facades

2.2.1 The Low Level API

Facades come into play in the objects created by the classes in org.bouncycastle.crypto.general. Each implementation class extends a guarded equivalent. For example Camellia.OperatorFactory extends GuardedSymmetricOperatorFactory, the guarded factory assures that a call to a Camellia.OperatorFactory will fail in the approved mode of operation.

2.2.2 The JCA/JCE provider

Two mechanisms are employed in the provider to prevent the handing out of non-approved services in the approved mode of operation.

The java.security.Provider.Service class that arrived in JDK 1.5 has been extended and modified so that it's newInstance() method checks the status of FipsStatus.isReady() and also makes use of a passed in object of the type EngineCreator to create implementations, rather than using the normal reflection mechanism employed in the JCA/JCE.

A special version of EngineCreator, called the GuardedEngineCreator has been defined which carries

out the approved mode check before allowing an object to be created for algorithms which should be disabled in the FIPS approved mode. In the event the executing thread is in approved mode status, the GuardedEngineCreator will return null. The extended service class in the BouncyCastleFipsProvider has been modified to take this into account, and throw a NoSuchAlgorithmException accordingly.

2.3 Underlying Bouncy Castle Classes Used.

Other than the use of repackaging, the FIPS version of Bouncy Castle includes the following original BC class files with key generators:

AESFastEngine	DESedeEngine
DSASigner	ECDSASigner
ECDHBasicAggreement	ECDHCBasicAgreement
ECMQVBasicAgreement	RFC3394WrapEngine
RSADigestSigner	PSSSigner
CustomNamedCurves	ECNamedCurveTable
SHA1Digest	SHA224Digest
SHA256Digest	SHA384Digest
SHA512Digest	SHA512tDigest
CMac	Hmac

Most of the following packages are also included, but as internal APIs

org.bouncycastle.crypto.modes
org.bouncycastle.crypto.paddings
org.bouncycastle.crypto.encodings
org.bouncycastle.crypto.prng.drbg

For the most part the above have ended up in org.bouncycastle.crypto.internal. No direct cryptographic functionality is available in these packages and the OSGI manifest for the internal packages will not allow their export.

The following packages are included as published APIs:

org.bouncycastle.math.ec
org.bouncycastle.asn1
org.bouncycastle.util

2.4 API Usage

2.4.1 Startup

FipsStatus.isReady() will only return true if all the classes providing implementations in the FIPS package have loaded successfully.

e.g. checking if startup complete:

FipsStatus.isReady()

A status message can be gathered from:

FipsStatus.getStatusMessage()

In the event there has been an error this will provide some detail on it.

2.4.2 Configuration of Approved/Unapproved Modes

CryptoServicesRegistrar calculates the default mode of operation based on the granting of

```
permission
   org.bouncycastle.crypto.CryptoServicesPermission “unapprovedModeEnabled”;
```

If this permission is granted by the security manager, then the JVM will start threads in a default of unapproved mode.

If this permission is not granted by the security manager, then the JVM will start threads in the approved mode only.

2.4.3 Use of CryptoServicesRegistrar.setApprovedMode(true)

If the JVM has been granted the use of unapproved mode services then a thread may move into approved mode by calling CryptoServicesRegistrar.setApprovedMode(true) if the permission:

```
permission
   org.bouncycastle.crypto.CryptoServicesPermission “changeToApprovedModeEnabled”
```

is granted.

If the permission is not granted together and a thread is not already in approved mode then the call to `CryptoServicesRegistrar.setApprovedMode(true)` will result in an exception being thrown.

2.4.4 Module Self Verification

The module will always be used as a signed jar. On startup the module will have its signature verified and also verify the class files against a SHA-256 HMAC stored in the jar files manifest in the file HMAC.SHA256.

Note: in the presence of a Java SecurityManager this requires the module to have `java.lang.RuntimePermission "getProtectionDomain"` enabled in order for the module jar to examine its own contents.

2.4.5 Setting a default SecureRandom

The `CryptoServicesRegistrar.setSecureRandom()` method is used to provide a source of randomness to be used in cases where none has been specified by the developer. If no default source is provided and one is requested an IllegalStateException will be thrown.

If the API is being accessed via the JCA/JCE provider a default FIPS approved SecureRandom will created if none has been provided. Which algorithm is used for the SecureRandom can be configured on provider creation. If `CryptoServicesRegistrar.setSecureRandom()` is subsequently called it will override the provider configuration.

Note: for FIPS approved mode operations the default SecureRandom must be a FIPS approved one.

2.4.6 Provider Configuration

The provider constructor is able to take a config string to all for the configuration of its default SecureRandom. This can be used either via the java.security file for the JVM:

```
security.provider.10=org...BouncyCastleFipsProvider C:DEFRND[HmacSHA512];ENABLE{ALL};
```

or through the constructor:

```
Security.addProvider(
           new BouncyCastleFipsProvider("C:DEFRND[HmacSHA512];ENABLE{ALL};"));
```

2.4.7 Use of the Provider With the JSSE

The provider can also be used to run the JSSE in FIPS mode if the host JVM supports (JDK 1.6 or later). In this case the provider name needs to be passed to the constructor of the JSSE provider either via the java.security file for the JVM:

```
security.provider.4=com.sun.net.ssl.internal.ssl.Provider BCFIPS
```

or using the JSSE provider constructor:

```
new com.sun.net.ssl.internal.ssl.Provider("BCFIPS")
```

Further details on using the JSSE in FIPS mode can be found at:

http://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/FIPS.html

It should also be noted here that RSA PKCS#1.5 key wrap, NONEwithDSA, NONEwithECDSA require 2 additional policy settings if the BCFIPS provider is run in "FIPS only" mode – as a rule these algorithms are not FIPS approved, except where used for TLS and the policy settings reflect this.

2.4.8 FIPS Compliant Key Store

The provider introduces the BCFKS KeyStore, which is capable of holding secret keys as well as public keys and certificates. The store acts as a replacement for the traditional BC UBER KeyStore, but is based around PBKDF2, AES, and SHA-512 so is full FIPS compliant.

2.4.9 Use of Keys Between Modes

In line with FIPS policy keys generated with unapproved mode generators cannot be passed to approved mode algorithms without translating the key explicitly and vice-versa.

2.4.10 Key Export and Translation

```
permission org.bouncycastle.crypto.CryptoServicesPermission “exportSecretKey”;
```

and

```
permission org.bouncycastle.crypto.CryptoServicesPermission “exportPrivateKey”;
```

or

```
permission org.bouncycastle.crypto.CryptoServicesPermission "exportKeys";
```

are required to do any exporting of CSPs outside of the module. These permissions are also required to be set to allow repackaging of keys between layers.

If neither of these permissions are set it is possible to import keys into the module and to generate keys within it, however without them the private values can never be displayed or persisted.

3 Examples

3.1 Basic Encryption (AES approved mode)

```
 // ensure a FIPS DRBG in use.
CryptoServicesRegistrar.setSecureRandom(
       new FipsDRBG.Builder(
            new BasicEntropySourceProvider(
                  new SecureRandom(), true))
       .build(FipsDRBG.SHA512_HMAC, null, false)
);


FipsSymmetricKeyGenerator<SymmetricSecretKey>  keyGen =
                            new FipsAES.KeyGenerator(FipsAES.ALGORITHM, 128,
                                CryptoServicesRegistrar.getSecureRandom());

SymmetricSecretKey key = keyGen.generateKey();

FipsSymmetricOperatorFactory<FipsAES.Parameters> fipsSymmetricFactory =
                                                 new FipsAES.OperatorFactory();

FipsOutputEncryptor<FipsAES.Parameters> outputEncryptor =
   fipsSymmetricFactory.createOutputEncryptor(key, new FipsAES.Parameters());

byte[] output = encryptBytes(outputEncryptor, new byte[16]);

FipsInputDecryptor<FipsAES.Parameters> inputDecryptor =
   fipsSymmetricFactory.createInputDecryptor(key, new FipsAES.Parameters());

byte[] plain = decryptBytes(inputDecryptor, output);
```

With the functions referred to above being:

```
static byte[] encryptBytes(
  FipsOutputEncryptor outputEncryptor, byte[] plainText) throws IOException
 {
     ByteArrayOutputStream bOut = new ByteArrayOutputStream();
     CipherOutputStream encOut = outputEncryptor.getEncryptingStream(bOut);

     encOut.update(plainText);

     encOut.close();

     return bOut.toByteArray();
 }
```

and:

```
 static byte[] decryptBytes(FipsInputDecryptor inputDecryptor,
                            byte[] cipherText) throws IOException
{
     ByteArrayOutputStream bOut = new ByteArrayOutputStream();
```

```
        InputStream encIn = inputDecryptor.getDecryptingStream(
                                        new ByteArrayInputStream(cipherText));
        int ch;

        while ((ch = encIn.read()) >= 0)
        {
            bOut.write(ch);
        }

        return bOut.toByteArray();
    }
```

3.2 Provider AES encryption – GCM mode.

A simple example of using the provider for an AES GCM known answer test.

```
        byte[] K = Hex.decode(
              "feffe9928665731c6d6a8f9467308308"
            + "feffe9928665731c6d6a8f9467308308");
        byte[] P = Hex.decode(
              "d9313225f88406e5a55909c5aff5269a"
            + "86a7a9531534f7da2e4c303d8a318a72"
            + "1c3c0c95956809532fcf0e2449a6b525"
            + "b16aedf5aa0de657ba637b391aafd255");
        byte[] N = Hex.decode("cafebabefacedbaddecaf888");

        Key                     key;
        Cipher                  in, out;

        key = new SecretKeySpec(K, "AES");

        in = Cipher.getInstance("AES/GCM/NoPadding", "BCFIPS");
        out = Cipher.getInstance("AES/GCM/NoPadding", "BCFIPS");

        in.init(Cipher.ENCRYPT_MODE, key, new IvParameterSpec(N));

        byte[] enc = in.doFinal(P);

        out.init(Cipher.DECRYPT_MODE, key, new IvParameterSpec(N));

        byte[] dec = out.doFinal(enc);
```

4 Example Policy Files

In the presence of a security manager the FIPS jar will start enforcing FIPS restrictions to ensure the use of the jar is compliant with certification. In this situation a policy file is used to prevent the use of non-FIPS algorithms or to allow some threads to execute as FIPS approved and some to execute in non-approved mode.

4.1 Unrestricted policy – FIPS restrictions enforced, approved and unapproved mode allowed but restricted to individual threads.

```
// policy which allows everything to everyone
grant {
   permission java.security.AllPermission "", "";
};
```

4.2 Minimal restricted policy – allows FIPS approved mode only, does not allow export of keys outside of FIPS module.

```
//
// Policy which grants the minimum required to operate in FIPS approved mode
//
grant codeBase "${lib.dir}${/}bc-fips-1.0.0-SNAPSHOT.jar" {
    // to allow checksum check
    permission java.lang.RuntimePermission "getProtectionDomain";
    // to allow module to examine private/secret keys
    permission org.bouncycastle.crypto.CryptoServicesPermission "exportKeys";
};
```

4.3 Minimal restricted policy with provider – allows FIPS approved mode only, allows installation of the provider by arbitrary code base, does not allow export of keys outside of FIPS module.

```
//
// Policy which grants the minimum required to operate in FIPS approved mode with
a provider installed.
//
grant {
    permission java.security.SecurityPermission "putProviderProperty.BCFIPS"; //
allow installation of the provider
};

grant codeBase "${lib.dir}${/}bc-fips-1.0.0-SNAPSHOT.jar" {
     // to allow checksum check
    permission java.lang.RuntimePermission "getProtectionDomain";
    // to allow module to examine private/secret keys
    permission org.bouncycastle.crypto.CryptoServicesPermission "exportKeys";

};
```