The Bouncy Castle FIPS Java API

20 November 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 provider for the Java Cryptography Architecture (JCA) and the Java Cryptography Extension (JCE) 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 around 499,000 lines of Java.

Overall, the design of the original BC APIs is such that the classes can be used to create objects which can be assembled in many different 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 either 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” (March 2, 2015 update), has indicated there are a number of boundary issues in the current BC APIs which need to be dealt with, in addition to the lack of self tests.

1.1 The Approach Taken

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 have made the FIPS

algorithms available along side a set of non-approved mode algorithms which can be disabled – the result being that it is straightforward to build the other BC APIs against either provider, giving us a bc-fips jar, in addition to the bcprov ones, with bcpkix, bcpg, and bcmail packages buildable and usable against either the bc-fips or bcprov jars.

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 if there is a Java security manager in use unless it is told otherwise by the JVM's policy file.

This has resulted in a new version of the BC APIs for use with FIPS. The new APIs have a new low level API split into two parts, a FIPS approved part and a non-approved part. In order to prevent boundary leakage we have introduced two sets of facades, one set at the low level, and one set at the JCA/JCE level, which hide a majority of the actual working classes which are either in an internal package, or if they contain cryptographic functionality, package protected in the package of use. The set of facades at the JCA/JCE level is primarily about preventing developer confusion; as the JCA/JCE level is built on the lower level an object carrying out an unapproved service cannot be constructed, or used, in approved mode. The facades allow us to prevent the creation and use of objects performing unapproved tasks, in threads which are meant to be in approved only mode.

Start-up validations are done using static methods invoked on class loading, the implementation classes are unusable if the validations do not pass. Other validations, such as key checking, and PRNG health tests have being added to the FIPS versions of the APIs as appropriate. Most of these validations will be performed regardless of whether the code is running in approved mode or not.

Zeroization of key material is managed via the JVM's garbage collection process. Proactive zeroization on de-allocation by finalisers is also used where appropriate as zeroization on de-allocation is a FIPS 140-2 requirement.

1.2 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.

Our primary sponsor, who have funded both API work and initial platform testing, has been:

http://www.tripwire.com/

Additional API work has also been 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 largely made possible through its clients purchasing Bouncy Castle support agreements.

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.

The BC ASN.1 library, BC math library and the BC utils library (including the Base64 and Hex encoders) is also present. These classes are basically the same as the regular BC ones, other than methods and classes deprecated in the BC APIs have been removed. The rest of the operational code is mostly 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 class 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
- FipsDH – Diffie-Hellman key agreement service creation and algorithm definitions.
- FipsDRBG – Determinstic random bit generator service creation.
- FipsDSA – DSA 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), SSH, X9.63, Concatenation Mode, IKEv2, and SRTP.
- 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.

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

The org.bouncycastle.crypto.general Package

This a collection of non-FIPS algorithms, or non-FIPS modes of FIPS algorithms, which are widely used in IETF, NESSIE, and other standards.

2.1 Current Algorithm Set

2.1.1 FIPS Approved

FIPS approved algorithms:

Algorithm	Variation
AES	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB128, CFB8, OFB, CTR, CCM, GCM, CMAC, KW, KWP
DRBG/PRNG	SP800-90 (CTR, Hash, HMAC)
DSA	DSA (SHA-1 SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224. SHA-512/256)
DH	DH, MQV
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
KDF	Feedback Mode, Counter Mode, Double Pipeline Mode, TLS (v1.0-v1.2), SSH, X9.63, Concatenation Mode (SP800-56A), IKEv2, SRTP
PBKD	SP800-132 (PKCS#5 scheme 2 with SHS family digests)
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), KTS-SVE, KTS-OAEP.
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, SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, SHAKE256
TripleDES	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB64, CFB8, OFB, TKW, CMAC

2.1.2 General Algorithms

General algorithms are composed of algorithms which are commonly accepted as sound, but are not FIPS approved, or algorithms that are FIPS approved, being used in modes that are not currently part of the NIST standards. If the jar is used in approved mode these algorithms will be unavailable, but they will be available if the jar is appropriately configured and the thread is not executing in approved only mode.

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, PKCS# Scheme 2 with non-FIPS digests.

KDF algorithms: SCRYPT.

General encryption algorithms:

Algorithm	Variation
AES	OpenPGPCFB, OCB, EAX, RFC3211Wrap
ARC4 (RC4)	RFC 6229
Blowfish	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB8, CFB64, OFB, CTR, EAX, OpenPGPCFB, CMAC
Camellia	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB8, CFB128, OFB, CTR, OpenPGPCFB, CCM, GCM, OCB, EAX, CMAC, GMAC, KW, KWP.
CAST5	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB8, CFB64, OFB, CTR, EAX, OpenPGPCFB, CMAC
DES	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB8, CFB64 OFB, CTR, EAX, OpenPGPCFB, CMAC, CBCMAC, CBCMACwithISO7816-4, CFB8MAC, ISO9797alg3MAC, ISO9797alg3MACwithISO7816-4, CFB8_MAC
DRBG/PRNG	X9.31 (AES TDES)
DSA	DSA (general digests), RFC 6979 – DDSA
DSTU4145	With GOST3411
EC	RFC 6979 – ECDDSA (digests approved, plus others). ECDSA (digests others)
ElGamal	RAW, PKCS1v1.5, OAEP
GOST28147	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB8, CFB64, OFB, CTR, EAX, CMAC, GCFB, GCTR
GOST3410	With GOST3411
ECGOST3410	With GOST3411
IDEA	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7,

Algorithm	Variation
	CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB8, CFB64, OFB, CTR, EAX, OpenPGPCFB, CMAC, CBCMAC, CFB8MAC
RC2	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB8, CFB64, OFB, CTR, EAX, CMAC, CBCMAC, CFB8MAC, RFC3217Wrap
RSA	RAW, PKCS1v1.5, OAEP, PSS, ISO9792-2, ISO9792-2 PSS, X9.31 (other digests plus FIPS digests for non-FIPS algorithms)
SEED	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB8, CFB128, OFB, CTR, GCM, OCB, EAX, CMAC, GMAC, KW, KWP
Serpent	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB8, CFB128, OFB, CTR, CCM, GCM, OCB, EAX, CMAC, GMAC, KW, KWP
SHACAL2	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB8, CFB256, OFB, CTR
TripleDES	OpenPGPCFB, CBCMAC, CBCMACwithISO7816-4, CFB8MAC, EAX, RFC3217Wrap, RFC3211Wrap
Twofish	ECB, ECBwithPKCS7, ECBwithISO10126-2, ECBwithX9.23, ECBwithISO7816-4 ECBwithTBC, CBC, CBCwithPKCS7, CBCwithISO10126-2, CBCwithX9.23, CBCwithISO7816-4, CBCwithTBC, CBCwithCS1, CBCwithCS2, CBCwithCS3, CFB8, CFB128, OFB, CTR, CCM, GCM, OCB, EAX, CMAC, GMAC, KW, KWP, 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.

In JDK 1.5 the java.security.Provider.Service class was introduced to the JCE. This class 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.encodings
- org.bouncycastle.crypto.macs
- org.bouncycastle.crypto.modes
- org.bouncycastle.crypto.paddings
- org.bouncycastle.crypto.params
- org.bouncycastle.crypto.prng.drbg
- org.bouncycastle.crypto.signers
- org.bouncycastle.crypto.wrappers

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
- 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. The class loader is used to run self tests on each primitive and the class providing the associated operators will only load successfully if the self-tests have passed. If not called explicitly FipsStatus.isReady() is triggered if any class providing cryptographic functionality in the API is called. This will happen whether the cryptographic functionality is for approved or unapproved operation.

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.

FipsStatus.isReady() will throw an Error exception if the module has ended up in the error state.

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 as an argument 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 it (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, and NONEwithRSA require 2 additional policy settings if the BCFIPS provider is run with a SecurityManager present and in “FIPS only” mode – as a rule these algorithms are not FIPS approved, except where used for TLS and the policy settings reflect this. In the most general case this will need:

permission org.bouncycastle.crypto.CryptoServicesPermission "tlsAlgorithmsEnabled";

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 fully FIPS compliant. The key store is written out in ASN.1 format.

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. This is done by tagging a key as been created in approved mode or not, any attempt by a thread in the alternate mode to use the key will result in an exception.

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.

2.4.11 System Properties

By default all the below properties are assumed to be false.

`org.bouncycastle.rsa.allow_multi_use` – in approved/unapproved mode the module will attempt to block an RSA modulus from being used for encryption if it has been used for signing, or visa-versa. If

the module is not in approved mode it is possible to stop this from happening by setting **`org.bouncycastle.rsa.allow_multi_use`** to **`true`**.

`org.bouncycastle.dsa.FIPS186-2for1024bits` – this property only has an effect in unapproved mode. If legacy DSA parameters must be generated and the parameter size is 1024 setting this property to **`true`** will result in the FIPS 186-2 algorithm being used for parameter generation.

`org.bouncycastle.tripledes.allow_weak` – setting this property to **`true`** will allow the use of TripleDES weak keys. This is only present as it is a requirement for CAVP testing.

`org.bouncycastle.ec.disable_mqv` – setting this property to **`true`** will disable support for EC MQV.

`org.bouncycastle.pkcs1.not_strict` – some other providers of cryptography services fail to produce PKCS1 encoded block that are the correct length. Setting this property to **`true`** will relax the conformance check on the block length.

3 Examples

3.1 Basic Encryption (AES approved mode)

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


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

SymmetricSecretKey key = keyGen.generateKey();

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

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

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

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

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 {
    // allow installation of the provider
    permission java.security.SecurityPermission "putProviderProperty.BCFIPS";
};

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";

};
```

5 Disclosures

The provider contains implementations of EC MQV as described in RFC 5753, “Use of ECC Algorithms in CMS”. In line with the conditions in:

http://www.ietf.org/ietf-ftp/IPR/certicom-ipr-rfc-5753.pdf

We state, where EC MQV has not otherwise been disabled:

“The use of this product or service is subject to the reasonable, non-discriminatory terms in the Intellectual Property Rights (IPR) Disclosures of Certicom Corp. at the IETF for Use of Elliptic Curve Cryptography (ECC) Algorithms in Cryptographic Message Syntax (CMS) implemented in the product or service.”