Keyczar: A Cryptographic Toolkit

Arkajit Dey[1] and Stephen Weis[2]

[1] Massachusetts Institute of Technology, Cambridge, MA, USA 02139
[2] Google Inc., Mountain View, CA, USA 94043

Abstract. Keyczar's goal is to make it easier for application developers to safely use cryptography. Keyczar defaults to safe algorithms, key lengths, and modes, and prevents developers from inadvertently exposing key material. It uses a simple, extensible key versioning system that allows developers to easily rotate and retire keys.

1 Introduction and Philosophy

The motivation for Keyczar grew out of a need to make cryptography easier to use for developers. Developers making improperly using cryptography can create serious security vulnerabilities. For instance, developers may use obsolete algorithms, weak key lengths, improper cipher modes, or unsafely compose cryptographic operations. Another common developer mistake is to fail to provision for key rotation or even to hard-code keys in source code.

Keyczar's goal is to address these issues by providing a simple application programming interface (API) for developers that handles basic cryptographic details. Keyczar also provides a simple key versioning and management system based on directories of human-readable flat files, which will be refered to as *keysets*. More information about Keyczar is available from `http://keyczar.org`.

2 Using KeyczarTool

All Keyczar keys are generated with the stand-alone KeyczarTool utility. Two implementations of KeyczarTool are available under `org.keyczar.KeyczarTool` in Java and `keyczart.py` in Python.

2.1 KeyczarTool create

KeyczarTool must first create a new keyset using the `create` command. A newly created keyset will initially contain just a metadata file, described in section 4.2.

`KeyczarTool create` requires `location` and `purpose` command-line flags that specify the location of the key set and its purpose. Valid purposes are currently `crypt` and `sign`. The create command may also take an optional `name` flag to give a newly created keyset a name. If the `asymmetric` flag is specified, the newly created set will contain asymmetric keys of the specified algorithm. Currently DSA is supported for keysets with as sign purpose. RSA is supported for both crypting and signing keysets.

Some example `create` commands:

- Create a symmetric signing (HMAC) keyset:
  `KeyczarTool create --location=/path/to/keyset --purpose=sign`
- Create a symmetric crypting (AES) keyset named "Test":
  `KeyczarTool create --location=/path/to/keyset --purpose=crypt --name=Test`
- Create an asymmetric signing (DSA) keyset:
  `KeyczarTool create --location=/path/to/keyset --purpose=sign --asymmetric=dsa`

2.2 KeyczarTool addkey

All Keyczar keys are created using the `addkey` command. This command requites a keyset `location` flag and may optionally have `status`, `crypter` and `size` flags. Section 4.1 describes the meaning of the status values, but briefly they are *primary*, *active*, and *inactive*. The default status is *active*. User-specified key sizes are supported, although it is recommneded that on default or larger key sizes are used.

The `addkey` command will create a new file in the keyset directory with an integer version number that is one greater than the currently largest version. Version numbers start from 1 and are described in Section 4.3. For example, if the current keyset contains the key file `1`, a new key version will be created in the file `2`. Some example `addkey` commands:

- Create a new primary key:
  `KeyczarTool addkey --location=/path/to/keyset --status=primary`
- Create a new active key:
  `KeyczarTool addkey --location=/path/to/keyset`

Keyczar supports encrypted keysets. The `crypter` flag may be used to encrypt a key set when adding a new key. It will specify the location of an existing keyset which will be used to encrypt a newly generated key:

- Create a new active key and encrypt it with another keyset:
  `KeyczarTool addkey --location=/path/to/keyset --crypter=/path/to/crypting/keys`

2.3 KeyczarTool pubkey

Public keys in Keyczar are exported from existing asymmetric key sets. The `pubkey` command requires both an existing `location` flag and a `destination` where public keys will be exported. If the keyset under `location` was not created with an `asymmetric` flag, then a `pubkey` command will fail. An example `pubkey` command works as follows:
`KeyczarTool pubkey --location=/path/to/keyset --destination=/path/to/dest`

2.4 KeyczarTool promote, demote, and revoke

The `promote`, `demote`, and `revoke` commands are used to change key status values. Each of these commands require a `location` and `version` flag.

Promoting an *active* key will raise its status to *primary*, and promoting an *inactive* status will make it *primary*. There can only be a single *primary* key in a given key set.

Similarly, `demote` will lower a *primary* key to *active*, and an *active* key to *inactive*. The `revoke` command will only work for *inactive* status values. The `revoke` command will permenantly delete key material, so should be used with caution.

Some example `promote`, `demote`, and `revoke` commands. Suppose that key version 1 initially has an *active* status:

- Promote *active* version 1 to *primary*:
  `KeyczarTool promote --location=/path/to/keyset --version=1`
- Demote *primary* version 1 back to *active*:
  `KeyczarTool demote --location=/path/to/keyset --version=1`
- Demote *active* version 1 to *inactive*:
  `KeyczarTool demote --location=/path/to/keyset --version=1`
- Revoke the *inactive* version 1:
- `KeyczarTool revoke --location=/path/to/keyset --version=1`

3 Using Java Keyczar

The `com.google.keyczar` package contains four public classes that developers will use for cryptographic functions: `Crypter`, `Encrypter`, `Signer`, and `Verifier`. All four classes are children of the abstract `Keyczar` class and may be constructed either with a `KeyczarReader` or a simple String location of a keyset:

- Constructor: `Keyczar(KeyczarReader reader)`
- Constructor: `Keyczar(String keySetLocation)`

3.1 KeyczarReaders

The `KeyczarReader` interface may be implemented by any class that reads key material from a keyset. Keyczar includes a `KeyczarFileReader` class that is used by the `Keyczar(String keySetLocation)` constructor to read files from the local disk. Developers may implement their own `KeyczarReaders` to read from arbitrary sources and pass them to `Keyczar` constructors.

3.2 Encrypters

The `Encypter` class is only able to encrypt data. `Encrypter` objects will be initialized by passing it the location of a crypting key set. `Encrypters` can encrypt plaintext encoded in UTF-8 Strings, byte arrays, or ByteBuffers. `Encrypters` must be initialized with a keyset containing a *primary* status key to encrypt any data.

- `String encrypt(String input)`: Encrypt a string and return a WebSafeBase64 output ciphertext.
- `byte[] encrypt(byte[] input)`: Encrypt a byte array and return the ciphertext as a byte array.
- `int encrypt(ByteBuffer input, ByteBuffer output)`: Encrypt the contents of the input buffer and write the ciphertext to the output. Return the number of bytes written.

3.3 Crypters

The `Crypter` class is a child of `Encrypter` and is also able to decrypt data. `Crypter` objects may only be initialized by crypting key sets that contain keys able to decrypt, namely symmetric keys or private keys. `Crypters` cannot be initialzied with public keys. `Crypters` can decrypt ciphertext in WebSafeBase64 encoded Strings, byte arrays, or ByteBuffers.

- `String decrypt(String input)`: Decrypt a WebSafeBase64 string and return a UTF-8 encoded plaintext output.
- `byte[] decrypt(byte[] input)`: Decrypt a ciphertext byte array and return the plaintext as a byte array.
- `int decrypt(ByteBuffer input, ByteBuffer output)`: Decrypt the input ByteBuffer and write its output into the given output ByteBuffer. Return the number of bytes written.

3.4 Verifiers

The `Verifier` class is only able to verify signatures. `Verifier` objects may be initialized by passing it the location of any signing key set. It can verify signatures in WebSafeBase64 format, as byte arrays, or in ByteBuffers:

- `boolean verify(String data, String signature)`: Verifies a WebSafeBase64 encoded signature on the given UTF-8 String of data. Returns a boolean representing whether the signature is valid.
- `boolean verify(byte[] data, byte[] signature)`: Verifies a signature in a byte array on the given byte array of data. Returns a boolean representing whether the signature is valid.
- `boolean verify(ByteBuffer data, ByteBuffer signature`: Verifies a signauture in a ByteBuffer on the given ByteBuffer of data. Returns a boolean representing whether the signature is valid.

3.5 Signers

The `Signer` class is a child of the `Verifier` class and is also able to generate signatures by signing data. `Signer` objects may only be initialized by keysets that are able to sign, namely symmetric keys and private keys. `Signers` may not be initialized by public keys and may sign data in UTF-8 Strings, byte arrays, or ByteBuffers. `Signers` must be initialized with a keyset containing a *primary* status key to sign any data.

- `String sign(String data)`: Signs a UTF-8 string of data and returns a signatures as a WebSafeBase64 string.
- `byte[] sign(byte[] data)`: Signs a byte array of data and returns a signature in a byte array.
- `int sign(ByteBuffer data, ByteBuffer signature`: Signs a ByteBuffer of input and writes the signature to the output ByteBuffer. Returns the number of bytes written.

3.6 Example: Putting KeyczarTool and Java Keyczar Together

The following command-line commands would create a new AES encrypting key:

```
KeyczarTool create --location=/aeskeys --purpose=crypt
KeyczarTool addkey --location=/aeskeys --status=primary
```

Then within a Java program, the following code would encrypt a string of data:

```
KeyczarCrypter crypter = new KeyczarCrypter(‘‘/aeskeys’’);
String ciphertext = crypter.encrypt(‘‘Some data to encrypt’’);
```

This same ciphertext would be decrypted with a call to `Crypter.decrypt()`:

```
String plaintext = crypter.decrypt(ciphertext);
```

Using public keys is similar:

```
KeyczarTool create --location=/rsakeys --purpose=crypt --asymmetric
KeyczarTool addkey --location=/rsakeys --status=primary
KeyczarTool pubkey --location=/rsakeys --destination=/rsa-publickeys
```

Then in Java, we can encrypt using the public key set, but need the private key set to decrypt:

```
// Initialize an Encrypter with the public keys
KeyczarEncypter encrypter = new KeyczarEncrypter(‘‘/rsa-publickeykeys’’);
String ciphertext = encrypter.encrypt(‘‘Some data to encrypt’’);
...
// Initialize a Crypter with the private keys
KeyczarCrypter crypter = new KeyczarCrypter(‘‘/rsakeys’’);
String plaintext = crypter.decrypt(ciphertext);
```

Signing and verifying is similarily easy:

```
KeyczarTool create --location=/hmackeys --purpose=sign
KeyczarTool addkey --location=/hmackeys --status=primary
```

In Java:

```
KeyczarSigner signer = new KeyczarSigner(‘‘/hmackeys’’);
String signature = signer.sign(‘‘Some data to sign’’);
boolean verified = signer.verify(‘‘Some data to sign’’, signature);
```

See Section 7 for a more detailed walkthrough and sample test vectors.

4 Keys

4.1 Statuses

As described in Section 2, Keyczar keys may have one of three status values: *primary*, *active*, *inactive*. Keys with a primary status value are used to generate new cryptomaterial, that is, signatures or ciphertexts. Keys of all status values may be used to verify or decrypt legacy data, but are not used to generate new data.

The idea is that a key will start its life as a primary key and be used to generate cryptographic output. That key can be rotated by generating a new primary key value, and changing the existing primary key to be active. Active keys will be kept around to verify and decrypt existing data, until they are inactive. A key that is inactive is identical to an active key, except its usage statistics are collected by default. When a key that is inactive is no longer used, it can be safely revoked and removed from a key set entirely.

4.2 Key Metadata

Each Keyczar keyset contains a metadata file that contains information about that set's purpose, key type, and a list of key version information (see Section 4.3). Valid purposes are encrypting, verifying, encrypting and decrypting, and signing and verifying. Having the purpose value in the metadata prevents a key from being used for an inappropriate purpose. For example, an RSA key that is typically used for encrypting shouldn't also be used for signing.

These purposes correspond to their respective public and symmetric/private keys, and are usable by the Keyczar classes specified in the table below:

Purpose	**Public, Private, or Symmetric**	**Class used by**
Encrypting	Public	Encrypter or Crypter
Encrypting and Decrypting	Private or Symmetric	Crypter
Verifying	Public	Verifier or Signer
Verifying and Signing	Private or Symmetric	Signer

Each metadata file contains a specific key type that only that set can contain. Currently, the valid key types are AES, HMAC-SHA1, DSA Private, DSA Public, RSA Private, and RSA Public.

The metadata will contain a list of version information, which is described in Section 4.3. See Section 7 for a sample JSON metadata file.

4.3 Versions

Each key metadata (see Section 4.2) contains a list of version numbers corresponding to the keys contained in a given keyset. Each version specifies an integral version number, a status of either *active*, *primary*, or *inactive*, and a boolean specifying whether the key is exportable outside of Keyczar.

By default, key version numbers start from 1 and increase montonically. However, key version numbers may be any positive, non-zero integral values.

4.4 Output Headers

Each Keyczar output is tagged with a Keyczar-specific header. The header format is as follows:

Format	Key Hash
1 byte	4 bytes

See Section 7 for a concrete example of an output header.

Version Byte The first byte of all output is a single byte representing the version of keyczar that generated it. The current version byte is 1. All other values are reserved.

Key hash Each header includes 4-byte truncated SHA-1 hash of the raw key material itself. The following values will be hashed for each key type:

- HMAC: 4-byte integer value representing the length of the HMAC key in bytes, followed by the actual HMAC key bytes.
- AES: 4-byte integer value representing the length of the AES key, followed by the acutal AES key bytes, followed by the 4-byte KeyHash of the attached HMAC key.
-
- DSA Public: 4-byte integer value representing the length of the X.509 representation of this key, followed by the X.509 representation.
- DSA Private: Same as DSA Public.
- RSA Public: 4-byte integer value representing the length of the X.509 representation of this key, followed by the X.509 representation.
- RSA Private: Same as RSA Public.

The SHA-1 output on each of these values is truncated to the first four bytes. For example: The truncated 4-byte truncated SHA-1 values:

1. The 4-byte truncated hash of SHA-1("") = `da39a3ee 5e6b4b0d 3255bfef 95601890 afd80709` encoded in hexadecimal is `da39a3ee.`
2. The truncated hash of SHA-1("The quick brown fox jumps over the lazy dog") = `2fd4e1c6 7a2d28fc ed849ee1 bb76e739 1b93eb12 is 2fd4e1c6` is `2fd4e1c6` .

4.5 Key Formats

All key files are kept in JSON format. All JSON representations contain "type" and "hash" values. The hash value is the key hash described in Section 4.4. Each key type has its own individual fields as well.

HMAC HMAC keys have the type "HMAC_SHA1" and contain an additional "hmacKeyString" field which is a Base64 representation of the raw HMAC key bytes.

AES AES keys have the type "AES" and contain an additional "aesKeyString" field which is a Base64 representation of the raw HMAC key bytes. AES keys also have an attached HMAC key, which is kept in a "hmacKey" field. AES keys also have an additional "mode" field which currently only supports "CBC".

Public RSA Public RSA keys have an "x509" field containing the Base64 representation of a X.509 public key certificate.

Private RSA Private RSA keys contain a public RSA key, and a "pkcs8" field containing a Base64 representation of a PKCS8 encoded RSA private key.

Public DSA Public DSA keys have an "x509" field containing the Base64 representation of a X.509 public key certificate.

Private DSA Private DSA keys contain a public DSA key, and a "pkcs8" field containing a Base64 representation of a PKCS8 encoded DSA private key.

5 Output Formats

5.1 Signatures

A Keyczar signature consists of a header (described in Section 4.4) followed by a algorithm-dependent signature:

Header	Sign(header | M)
5 bytes	Algorithm-dependent length

(Note: | represents string concatenation.)

For example, HMAC keys will produce a 20-byte output, so have the form:

Header	HMAC-SHA1(header | M)
5 bytes	20 bytes

5.2 Ciphertext

A Keyczar ciphertext contains a header (described in Section 4.4), an IV (if needed), the algorithm-dependent ciphertext, and a signature on the preceding fields (for symmetric keys only):

Header	IV (if any)	Encrypt(M)	Sign(header | IV | M) (if any)
5 bytes	Algorithm-dependent	Algorithm-depndent	Algorithm-depndent

For example, 128-bit AES keys (which have an attached HMAC key) used in CBC mode would produce the following output:

Header	IV	AES-CBC(M)	HMAC-SHA1(header | IV | M)
5 bytes	16 bytes	Algorithm-dependent	20 bytes

In contrast, RSA ciphertexts neither use IVs nor are signed. Thus, the RSA ciphertext format is:

Header	RSA-OAEP(M)
5 bytes	Single RSA block

6 Algorithm Details

These are the default key lengths, algorithms, and padding modes used:

- HMAC: Default keys are 256 bits. SHA1 used as the hash algorithm.
- AES: Default keys are 128 bits. 192 and 256 bit keys are also. CBC mode with PKCS#5 padding is used by default.
- DSA: DSA-SHA1 signing algorithm used by default. Default key size is 1024 bits.
- RSA Encryption: RSA-OAEP encryption is used. Default key size is 2048 bits. 1024, 768, and 512 bit keys are also supported.
- RSA Signatures: RSA-SHA1 signing is used. Default key size is 2048 bits. 1024, 768, and 512 bit keys are also supported.

7 Walkthrough and Sample Data

7.1 Creating an HMAC Keyset

A `meta` file is created by a call to the `KeyczarTool create` command. The following example was created usign the command:

```
KeyczarTool create --location=/path/to/keyset --name=Test --purpose=sign
```

```
{
 "name":"test",
 "purpose":"SIGN_AND_VERIFY",
 "type":"HMAC_SHA1",
 "versions":[],
 "encrypted":false
}
```

7.2 Generating a HMAC key

The following command will create a new key in an existing keyset:
`KeyczarTool addkey --location=/path/to/keyset`

This will add key version information to the existing `meta` file:

```
{
 "name":"test",
 "purpose":"SIGN_AND_VERIFY",
 "type":"HMAC_SHA1",
 "versions":[
    { "exportable":false,
      "status":"ACTIVE",
      "versionNumber":1}
    ],
 "encrypted":false
}
```

It will also create a key version file named "1":

```
{
 "hmacKeyString":"9E7yslYyE4GHzq2WbppOOjGqBL7OGXxy5OsFSqBn2ao",
 "size":256
}
```

7.3 Promoting a Key

Since no status was specified, key version 1 was created with an "active" status. That means it can be used for verifying signatures, but not generating new signatures. Since we just created this key, there are no existing signatures to verify. We have to promote it to a "primary" status before it's useful:
`KeyczarTool promote --location=/path/to/keyset --version=1`

This will just alter the meta file to reflect the promotion:

```
{
 "name":"test",
 "purpose":"SIGN_AND_VERIFY",
 "type":"HMAC_SHA1",
 "versions":[
    {
      "exportable":false,
      "status":"PRIMARY",
      "versionNumber":1
    }
  ],
 "encrypted":false
}
```

We can generate a second key version with a primary status as follows:
`KeyczarTool addkey --location=/path/to/keyset --status=primary`

This create a new key version "2" with a primary status. Keysets can only have one primary key at a time, so this automatically demotes key version "1" to be active:

```
{
 "name":"test",
 "purpose":"SIGN_AND_VERIFY",
 "type":"HMAC_SHA1",
 "versions":[
    {
      "exportable":false,
      "status":"ACTIVE",
      "versionNumber":1
    },
    {
      "exportable":false,
      "status":"PRIMARY",
      "versionNumber":2
    }
  ],
 "encrypted":false
}
```

This creates another key version file "2" with the following contents:

```
{
  "hmacKeyString":"-jkgTURAPoBr9SaQ5NLsFq2Xu5Z54RmnHYHlv1RVE7s",
  "size":256
}
```

7.4 Key Hash Values

Note that the Base64 value of the "hmacKeyString" in the previous section decodes to the following 32 bytes in hexadecimal format:

```
fa, 39, 20, 4d, 44, 40, 3e, 80,
6b, f5, 26, 90, e4, d2, ec, 16,
ad, 97, bb, 96, 79, e1, 19, a7,
1d, 81, e5, bf, 54, 55, 13, bb
```

These key bytes will be hashed to obtain a key identifier. The raw key bytes will be prefixed by a byte representation of the length in bytes. In this case, the length is 32, so the following bytes (in hexadecimal format) will be hashed with SHA-1:
`SHA-1(00, 00, 00, 20, fa, 39, ..., 13, bb)`

This produces the following 20-byte SHA1 output:

```
d8, 36, 36, 62, d0, 6d, cb, a7,
35, cd, 6c, 69, 33, 40, df, 87,
ce, da, d9, 8d
```

This is truncated down to the four bytes `d8, 36, 36, 62`, which is this key's hash identifier. As described in Section 4.4, all output will be prefixed with a version byte (currently equal to 1) and this identifier.

7.5 Signing and Verifying

To sign the string "Hello world" with our newly generated keys in Java, we would simply use the following two lines of code.

```
Signer signer = new Signer("/path/to/keyset");
String signature = signer.sign("Hello world");
```

Keyczar will convert the string to the bytes to the following values:

```
48, 65, 6c, 6c, 6f, 20, 77, 6f, 72, 6c, 64
```

The output Base64 encoded signature would be the value `Adg2NmIPTshpioGdunRaGRYWFNgheKmjvg`, which corresponds to the bytes:

```
1, d8, 36, 36, 62, f, 4e, c8,
69, 8a, 81, 9d, ba, 74, 5a, 19,
16, 16, 14, d8, 21, 78, a9, a3,
be,
```

Note that the first output byte is the version byte 1, followed by the 4-byte key hash identifier, and a 20-byte HMAC-SHA1 output. All signatures produced by this key will have this same identifier. Verification of this signature is simple:

```
boolean isValid =
    signer.verify("Hello world", "Adg2NmIPTshpioGdunRaGRYWFNgheKmjvg");
```

8 Licenses and Dependencies

Keyczar is available under an Apache 2.0 license [1]. Java Keyczar depends on the Google GSON package [2]. It also relies on the Java's `javax.crypto` package, which may not be available in all countries due to local laws and regulations.

Python Keyczar depends on the Python Cryptography Toolkit [4] and simplejson [3].

9 Acknowledgements

Thanks to Ben Laurie, Marius Schilder, and Neil Daswani for their original design contributions. Thanks to Sarvar Patel, Loren Kornfelder, Manuel Marquez Garrido, and Laura Krotowski for their various contributions.

References

1. Apache Software Foundation. Apache license, version 2.0. http://apache.org/licenses/LICENSE-2.0.
2. Google Inc. Google GSON project. http://code.google.com/p/google-gson/.
3. Ippolito, B. simplejson. http://pypi.python.org/pypi/simplejson.
4. Kuchling, A. Python crypto toolkit. http://www.amk.ca/python/code/crypto.html.