JAVA CRYPTO

Secure use of the Java Crypto API

Whitepaper v1.5

May 2019

cryptosense

Contents

© Cryptosense 2019

List of Figures

This document is protected by copyright. No part of the document may be reproduced or redistributed in any form by any means without the prior written authorization of Cryptosense. This document is provided "as is" without any warranty of any kind. Cryptosense SA cannot be held responsible for any misconduct or malicious use of this document by a third party or damage caused by any information this document contains. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Cryptosense SA, 231 Rue Saint-Honoré, 75001 Paris France

cryptosense.com

© Cryptosense 2019

About this Document

Java is still the world's most widely used programming language, powering a large proportion of the applications that modern businesses rely on. As security rises steadily up the priority list, these applications use more and more cryptography, to store passwords, to encrypt database fields, to communicate with servers and clients using TLS, and so on. Most developers are aware that writing their own implementations of cryptographic algorithms is inviting security trouble, hence most applications access cryptography through the standard Java Cryptographic Architecture (JCA), using an existing crypto provider or Java Crypto Engine (JCE).

However, using a standard crypto provider is by no means the end of the security story. Developers must take care to choose the right algorithms, manage their parameters in the right way, employ the right cryptographic modes for their applications, take care over random number generation, avoid key-management vulnerabilities and put all this together into secure cryptographic protocols.

This whitepaper is intended for developers who use, or are considering using, the Java crypto API, and for application security testers who review crypto security. It is not intended to be an introduction to cryptography, but rather a concise guide for readers familiar with crypto basics. We will tour the Java crypto API and explain common mistakes that cause security problems and crop up frequently in real applications. Finally, we discuss crypto security audit for Java, and show how Cryptosense's Analyzer can automate the process, drastically improving coverage while reducing time taken.

Acknowledgements

Thanks to all the readers of previous versions of this whitepaper v1.2 readers who submitted feedback or suggestions, including Brian Bordini, Jouni Aro, Lee Johnston, Paul Bottinelli, Brendan Gormley, Daniel Bleichenbacher, Mark Penny, Wojciech Podgórski, Joseph Tinucci, Sylvain Ruhault, and Gerry Weeks. Apologies to anyone we forgot or whose suggestions we couldn't include.

© Cryptosense 2019

1. Introduction

The Java Cryptographic Architecture [1] was designed to offer access to cryptographic functionality with implementation independence. The idea is that a standard interface for access to crypto is provided by Java, and then different implementors can write their own cryptographic service provider [2] that respects the interface and offers some set of cryptographic operations. The API is also designed to offer a degree of algorithm independence, by providing a uniform interface to high-level operations like `KeyFactory`, `Signature`, and `Cipher`. This means the selection of the algorithm for implementing these operations can then be an independent piece of code. Finally, it is also intended to be extensible. Implementors can add new algorithms to providers, however they must respect one of the engine classes, that is they must have a pattern of parameters that "look like" an existing function.

1.1. Some Common Providers

A number of providers are maintained by Oracle since the Sun acquisition including [3]:

» SUN: from JDK 1.1, only signature and digests (U.S. export regulations did not allow encryption)

» SunJSSE: from JDK 1.2, added support for RSA algorithm

» SunRsaSign: from JDK 1.3 improved support for RSA signatures (e.g., includes SHA2 hash function family)

» SunJCE: Java Cryptographic Extension, included from JDK 1.4 (when U.S. export regulations were relaxed), supports many cryptographic functions

There are also non-Oracle alternatives, including:

» BouncyCastle [4]: Open-source provider that includes dozens of cryptographic functions

» IBMJCE [5] - IBM's provider which includes similar algorithms to Oracle's plus a few others, including Mars, IBM's entry to the AES competition, and Seal, an IBM stream cipher.

Even though export restrictions have now mostly been lifted, the fact that early Java crypto providers did not include encryption still affects Java crypto security today as we will see in section 5.

1.2. Using the right provider

Each cryptographic service provider, or just provider, must have a name that allows it to be identified and a list of algorithms that it supports. Once can use a default provider, install additional providers and specify a preference order for providers.

A common mistake is to add a new provider using the `Security.addProvider` method without realising that this will add the provider in the next available slot, i.e typically at the end of the list. If the application then asks to perform a crypto operation without specifying the provider, it is unlikely the new provider will be used, but no error will be visible. If, for example, the provider came from some cryptographic module such as an HSM (Hardware security module), this may result in operations being carried in software that should have been carried out in hardware.

© Cryptosense 2019

To avoid this problem, Java gives a way to ask for a specific provider for a particular algorithm, and also to insert a provider at a given position in the preferences order using the `Security.insertProviderAt` method [1]

1.3. Bugs in Crypto Providers

Implementing cryptography is difficult, even for experts. For time to time bugs emerge in crypto providers which can have a catastrophic effect on crypto security.

Following a number of high-profile issues (Heartbleed, Gotofail, etc.), cryptographic libraries have started to attract more attention from researchers. A recent project by Google [25] discovered bugs in BouncyCastle and OpenJDK that leaked private keys. Kudelski Security also have an open source tool for fuzzing crypto libraries and comparing the results to reference implementations [26].

It is therefore vital to keep providers up to date. However, note that most crypto issues are the result of bad usage of a crypto library, not bugs in the library. In 2014 MIT researchers examined a sample of crypto-related CVEs and found that only 17% were related to bugs in libraries and 83% to misuse of the library by the application [19]. It is this second category that will be the subject of the rest of this whitepaper.

1.4. Crypto in Frameworks

Most ″enterprise″ web applications are constructed within application frameworks such as Weblogic, Websphere, Tomcat etc. These frameworks typically include components that will employ cryptography, for example to implement TLS, single sign-on (SSO), encrypted cookie management and so on. Often source code is not available and so it's hard to see exactly what cryptographic operations these components are employing.

Sometimes, they contain serious vulnerabilities [20]. They may include crypto providers different from those used by the main application. Their behaviour is often controlled by configuration options that may be hidden deep in XML files. Any review of cryptographic security for such an application must also consider the crypto employed by these components.

2. Encryption

In this section we'll take a brief tour of the encryption functionality in the Java crypto API. As a reminder, note that in general it is dangerous to use encryption (or any cryptographic operation) without considering the also the security of the protocol that the operation forms part of, i.e. the exchange of messages between one principal and another. This is because even if the encryption operation is carried out on a perfectly secure way, errors at the protocol level can still lead to a loss of security. We will return to this subject in section 6. Here we will just focus on the algorithms, keylengths, and modes of operation offered by Java crypto, and some common mistakes.

Each provider implements services, and services of type "Cipher" correspond to encryption algorithms. Most providers implement at least a few ciphers which are now considered to be insecure. For example, in table (figure 1), we list some of the algorithms implemented in the

© Cryptosense 2019

standard Oracle SunJCE provider. Of these, DES is now considered obsolete, since specialist hardware can brute-force search its 56-bit keyspace in a couple of hours. RC4 is deprecated since bias in the keystream can be used to mount attacks which are becoming more and more practical. Ciphers with 64-bit block-sizes like Blowfish, 3DES and RC2 are also deprecated [27]. Guides published by standards bodies such as ENISA and NIST give the state-of-the-art in cryptanalysis of cipher algorithms [6].

Algorithm Name	Description
AES	Advanced Encryption Standard as specified by NIST in FIPS 197. To use the AES cipher with only one valid key size, use the format AES_<n>, where <n> can be 128, 192, or 256.
AESWrap	The AES key wrapping algorithm as described in RFC 3394. To use the AESWrap cipher with only one valid key size, use format AESWrap_<n>, where <n> can be 128, 192, or 256.
ARCFOUR	A stream cipher interoperable with the RC4 cipher by Ron Rivest.
Blowfish	The Blowfish block cipher designed by Bruce Schneier.
DES	The Digital Encryption Standard as described in FIPS PUB 46-3.
DESede	Triple DES Encryption (also known as DES-EDE, 3DES, or Triple-DES).
DESedeWrap	The DESede key wrapping algorithm as described in RFC 3217 .
PBEWith....	Password-based encryption algorithm (PKCS5), using the specified message digest, pseudo-random function and encryption algorithm.For example: PBEWithMD5AndDES
RC2	Variable-key-size encryption algorithms developed by Ron Rivest
RSA	The RSA encryption algorithm as defined in PKCS #1

Figure 1 - Some Ciphers supported by the Oracle SunJCE crypto provider.

2.1. Specifying a Cipher

Ciphers need to be initialized by specifying a mode and a key. Possible modes are: `ENCRYPT_MODE`, `DECRYPT_MODE`, `WRAP_MODE`, `UNWRAP_MODE`. These modes are not to be confused with *modes of operation* which describe how the cipher algorithm should be applied to the plaintext, in particular when the plaintext is longer than a single block-size. One can also specify a *padding scheme*, which describes how to pad out a plaintext or plaintext part that is shorter than a single block-size.

These details are specified in *transformations*, which are strings giving the name of a crypto algorithm, and may be followed by a mode and padding scheme. For example, the following are valid transformations:

» *"AES/CBC/PKCS5Padding"*

» *"AES"*

If you employ an algorithm name as the transformation without giving a padding scheme or mode of operation, as in the example "AES", a provider-dependent default will be used. This is not good

© Cryptosense 2019

practice: first because provider defaults are often insecure, and second because the behaviour may change with a different provider or a new version.

2.2. Symmetric-key Mode and Padding Choice

For a detailed description of these modes see the NIST document 800-38 [21], and for a more detailed discussion of their security properties, see Rogaway's survey [7].

2.2.1. ECB

To see why operation mode choice is critical for security, the simplest example is to look at Electronic Code Book (ECB) mode. This mode consists of just encrypting the plaintext chunk by block-sized chunk and concatenating the results (see figure 2). For most applications, this will be highly insecure even if the block cipher is secure. The reason is that this kind of encryption is *deterministic* - encrypting the same plaintext yields the same ciphertext each time - and there is no protection against manipulation of the ciphertext (splicing blocks in a different order etc.).

These flaws lead to real practical attacks. It is well-known that for long messages, ECB mode leaks information (see the famous Tux penguins diagram on the wikipedia page [8]). It is less well-known that using ECB often opens the door to more powerful attacks that recover plaintexts. For example, suppose we use ECB with a cipher with a 16-byte block-size. If an attacker can prepend arbitrary prefix to the plaintext, which is often the case in real protools, he can bruteforce blocks byte after byte. To make the attack he prepends 15 known bytes, bruteforce guesses byte 16, and then iterates over all bytes.

Generally ECB mode should be completely avoided.

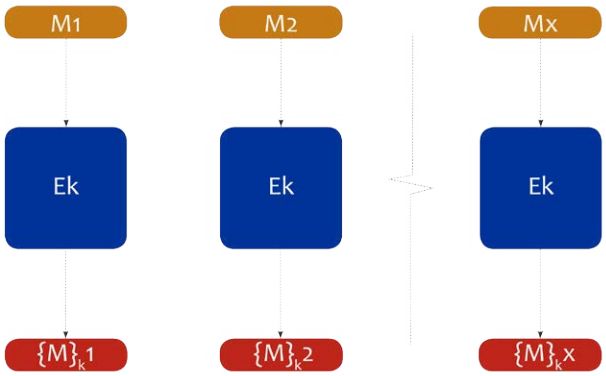

Figure 2 - ECB mode encryption of the x blocks of a plaintext M

2.2.2. CBC

Cipher-block chaining mode (CBC) is a widely-used mode that chains together blocks using XOR (see figure 3). It also introduces an *initialization vector* (IV) that if correctly used (it should be a random value) can ensure that encryption is not deterministic and so remove information leaks seen in ECB. However, CBC mode does not protect against manipulation of the ciphertext. For example, in general an attacker can remove a block from the beginning or end of a message in a way that is not detectable by the decryption mode, which can lead to attacks (see below) [10].

To prevent attacks coming from fake or tampered ciphertexts (chosen-ciphertext attacks), generally it is considered best practice to use authenticated encryption modes such as CCM or

© Cryptosense 2019

GCM (see below) in preference to CBC. If CBC has to be used, it is advisable to use a MAC to authenticate the ciphertext. This MAC should be constructed from a different key than the one used for encryption applied to the ciphertext and the IV, and the MAC should be checked before the ciphertext is decrypted.

Notice that the security of CBC mode is compromised if the IV is predictable. In particular if the IV is fixed or repeated, information is leaked, as identical prefixes in the plaintext will lead to identical prefixes in the ciphertext.

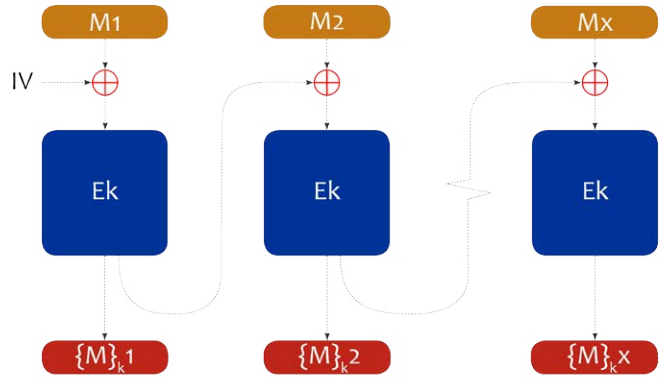

Figure 3 - CBC mode encryption

2.2.3. CTR Mode

Counter mode (CTR) is a conceptually simple mode. The block cipher is used to calculate a keystream that will be XORed against the plaintext. Roughly speaking, the first block of the keystream is obtained by encrypting the initialisation vector (IV). The second block of the stream is obtained by increasing the IV by 1 and encrypting again. This is repeated until the keystream is as long as the plaintext.

CTR mode is efficient and simple, however note it provides no protection against manipulation of the ciphertext (indeed, individual bits of the plaintext can be flipped by flipping the corresponding bit of the ciphertext), and if an IV is reused, the attacker can XOR the resulting ciphertexts together to obtain the XOR of the plaintexts. If he knows some bits of one of the plaintexts, or even just knows what format the plaintext is in, this leads to a direct attack [9].

If CTR mode is to be used, it should be combined with a MAC (using a different key) to protect ciphertext authenticity. Indeed, combining CTR mode with a MAC is the basis for the modern *authenticated encryption* modes described below.

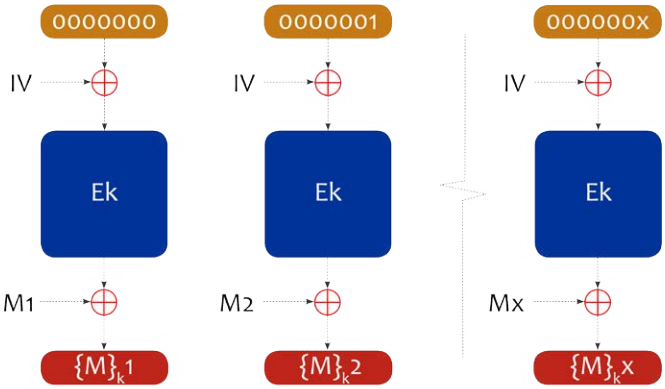

Figure 4 - Counter-mode encryption

© Cryptosense 2019

2.2.4. CCM and GCM

While combining a MAC with a "traditional" encryption mode like CBC or CTR can result in a secure encryption operation, this process is error-prone: if the same key is used for both operations, or composition is carried out in the wrong order (for security, one should encrypt-then-MAC rather than MAC-then-encrypt), security can be lost. For this reason, combined encryption and authentication modes using a single key have been developed and standardised. Those with widest support in common Java crypto providers are Galois counter mode (GCM) and Counter with CBC-MAC (CCM).

Both of these modes have acceptable security proofs, however note that both are insecure if the IV is allowed to repeat.

The IV for these modes consists of a 96-bit random value to which a 32-bit counter is added that is incremented for each block encrypted to created the keystream. This means that after 2^32 blocks, the counter value "wraps around" to its original value resulting in the same keystream block being reused, breaking security. Correct implementations of GCM mode will prevent the application from encrypting more than 2^32 blocks with the same key and IV, but some widely used providers have got this wrong in the past [25].

2.3. Padding Mode Choice

When the plaintext size is not an exact multiple of the block size we need to "pad" the last block in an unambiguous way, i.e. a way which allows us to recover the plaintext after decryption. The most commonly used padding is the so-called `PKCS5Padding` (to use the Java transformation string), whereby if we need e.g. 5 bytes to reach the block size we add 05 05 05 05 05. If no padding is required, then we add a full block of padding to the plaintext, to avoid ambiguity.

When used in combination with a padding mode that does not protect against manipulation of the ciphertext by an adversary, such as CBC mode, this padding can lead to so-called *padding oracle attacks*, where the attacker manipulates the ciphertext and tricks the system into revealing whether the plaintext contained inside is correctly padded or not [10]. This leaks information which can be used to recover the plaintext efficiently. Such attacks frequently occur in practice in web applications where encrypted data is serialized into URL strings or POST requests. Open-source tools exist for detecting and exploiting these vulnerabilities [22].

While padding modes do exist that mitigate these problems (by making the attacks less efficient), they are not widely supported by Java crypto providers. The real solution is to always use authenticated encryption.

2.4. Asymmetric Encryption Modes

There are two available padding modes available for use with RSA encryption in Java: PKCS#1v1.5 and OAEP. The PKCS#1v1.5 scheme has been known to be vulnerable to a chosen ciphertext attack (CCA) since 1998 [18]. The attack has recently been improved to require a median of less than 15 000 chosen ciphertexts on the standard oracle. Instances of the attack in widely-deployed real-world systems continue to be found. It should therefore not be used. One last place it used to have widespread use was in the TLS protocol, but it is finally being dropped from TLSv1.3. Other APIs such as the W3C Webcrypto API have also excluded it.

© Cryptosense 2019

In asymmetric cryptography, there is no "cipher mode" in the sense of combining blocks as there is for symmetric-key cryptography. Hence there is some ambiguity about what to put in the mode position of the Java transformation that specifies an RSA cipher. This leads to inconsistencies between providers. For example, Oracle JCE and OpenJDK specify RSA with PKCS#1v1.5 padding as `RSA/ECB/PKCS1Padding`, whereas BouncyCastle calls it `RSA/None/PKCS1Padding`.

RSA-OAEP mode has a well-studied security proof of preservation of indistinguishability under chosen ciphertext attacks (IND-CCA, the standard desirable notion of security for an encryption scheme), and so should be favoured. Two hash functions are required by the padding mode, a digest function and a mask generation function. The transformation strings specify which function to use in the format `OAEPWith<digest>And<mgf>Padding`, but there are sometimes discrepancies between providers. For example, `RSA/ECB/OAEPWITHSHA256ANDMGF1PADDING` works with Bouncy Castle and some other providers but not Oracle, while `RSA/ECB/OAEPWITHSHA-256ANDMGF1PADDING` works on Oracle and OpenJDK.

3. Hashing and Signing

In this section we will cover hash functions, MAC and signature in the Java crypto API.

3.1. Hash Functions

Cryptographic hash functions take an "arbitrarily" long message and produce a fixed size digest. For security, there are a number of properties a hash function should have. Which ones are necessary depends on the application, as we will see below.

> *first pre-image resistance:*
> from z it is infeasible to compute x such that h(x)=z, i.e. if I give you a digest value, you can't feasibly come up with a file that will hash to give that digest
> *second pre-image resistance:*
> from x it is infeasible to compute x'≠x s.t. h(x)=h(x'), i.e. if I give you a file, it's not feasible for you to come up with a second file that has the same digest
> *collision-free:*
> it is infeasible to compute x'≠x s.t. h(x)=h(x'), i.e. it is not feasible for you to find two values that hash to give the same digest.

Java crypto providers generally support a variety of hash functions, including some which are now considered broken. For example, most providers support MD5, for which collisions can be easily calculated. This means that MD5 is dangerously insecure for use as a digest function for signature, as an adversary could construct two documents with the same digest. These techniques have been used in the wild - the FLAME malware used to attack the Iranian nuclear programme used a MD5 collision in a certificate digest to bypass code signing protection.

The first SHA-1 collision was announced in February 2017. Although collisions are still expensive to compute, this single collision can be used to produce apparently arbitrary pairs of PDF documents with the same hash digest. Hence SHA-1 should no longer be used as a digest function for signature. MD2 is already obsolete.

© Cryptosense 2019

The only hash functions available in most Java crypto providers that are approved by ENISA and NIST are SHA-256, SHA-384 and SHA-512. Support for the new SHA-3 standard recently arrived in the Oracle and OpenJDK providers (in Java 9) and is also in Bouncy Castle. Additionally, Bouncy Castle has implementations of Keccak-256 and Keccak-512, which are earlier versions of the proposal that became SHA-3. These functions are used in the Ethereum blockchain. Note that the IBM provider offers a function called `SHA3` but this is actually SHA-384 (it also offers `SHA5` which is SHA-512).

3.1.1. HMAC

Message Authentication Codes (MACs) are used to demonstrate authenticity and integrity of messages. One way to construct a MAC is by hashing the message and a secret key. The HMAC function provides a secure way of doing this. Providers usually offer several versions of HMAC with different underlying hash function, e.g. in the Oracle provider: `HmacMD5`, `HmacSHA1`, `HmacSHA224`, `HmacSHA256`, `HmacSHA384`, and `HmacSHA512`.

Due to the way HMAC is constructed, it does not rely on the collision-freeness of the underlying hash function for security, only its pseudo-randomness. Therefore, HMAC-MD5, for example, is not such a security concern as a MD5 signature digest. However, standards bodies like ENISA and IETF documents like RFC 6151 recommend not using HMAC-MD5 for any future protocols. One reason is that the output size is fixed to the output size of the hash, in this case 128 bits, which is considered too short for future use. HMAC-SHA1 has a 160-bit output size, hence is less worrying. However HMAC-SHA-1 can be implemented very fast in hardware, which can lead to problems if the data to be hashed has low entropy (e.g. a password, see section 4).

3.1.2. MAC

Another way to build a MAC is to use a standard block cipher and a MAC mode. While the Oracle crypto provider doesn't support this, other providers such as Bouncy Castle do. The CMAC and GMAC algorithms have well-studied security proofs, but other BC-supported MAC modes should be used with care. For example, raw CBC MAC is not secure for variable-length messages, and ISO 9797 alg 3 has a key-recovery attack when used with 3DES (and a large volume of messages).

3.2. Signature

In Java crypto, signature algorithms are specified along with their hash function names. For example, `MD5withRSA`, `SHA1withRSA` are signature functions that use RSA for signing. DSA and ECDSA are also supported by multiple providers, though note that despite widespread adoption current ENISA recommendations do not approve DSA or ECDSA for future use [11].

Note that signature function names beginning RSA will use PKCS#1v1.5 padding for the signature. This mode is deprecated, as it has an attack if low public exponents are used in the keys and verification functions don't parse signatures perfectly. This sounds like a corner-case vulnerability but in practice is continues to occur. New applications should use PSS signatures as defined in RSA PKCS#1v2. The Java names for these signature modes specify a mask generation function (MGF), e.g. `SHA1withRSAandMGF1`.

© Cryptosense 2019

4. Password-based Cryptography

Generally speaking, passwords are something to be avoided. However, sometimes that is not possible: sometimes applications have nowhere to store a key securely (e.g. mobile applications), or they may need an initial credential for bootstrapping, or they might need to transmit a human-readable secret.

4.1. Password-based Key Derivation

A low-entropy secret like a password is not suitable for use as a cryptographic key directly. One must first apply a password-based key derivation function (PBKDF). These rely on two key ideas:

4.1.1. Key Stretching

We apply a computationally-intensive function to the "short" password to produce a "long" key in such a way that the only way to to confirm a guess at the password is to re-apply the same function. Typically this function is constructed from a hash function applied many times. The number of applications of the hash function is referred to as the number of iterations in the PBKDF. In recent PBKDFs, designers try also to make the calculation memory-intensive, to increase cost of hardware-based attacks.

4.1.2. Salting

An idea introduced by Morris in 1979 is to add a unique, public value to each key derivation. This diversifier protects against pre-computation attacks (so-called "rainbow tables") where attackers compute the key that will result from common passwords just once, and then can try their guesses against many keys. Since each key derivation has a unique salt, attackers will have to recompute the keys each time.

4.2. PBKDF2

The only FIPS/NIST approved PBKDF is the so-called PBKDF2, standardized in RFC 2898 and PKCS#5. PBKDF2 replaced version PBKDF1 which had a fixed output length (the size of hash function output) and fixed hash functions (MD2, MD5, SHA1). PBKDF2 can be used with any pseudorandom function, but it most commonly seen with HMAC-SHA-1.

In Java, PBKDF2 can be called by specifying it as the secret key creation method using a snippet of code similar to this:

```
     SecretKeyFactory f = SecretKeyFactory.getInstance("PBKDF2With-
HmacSHA1");
     KeySpec ks = new PBEKeySpec(password,salt,1024,128);
     SecretKey s = f.generateSecret(ks);
     Key k = new SecretKeySpec(s.getEncoded(),"AES");
```

This assumes the provider supports `PBKDF2WithHmacSHA1` (the Oracle provider does, but currently BouncyCastle does not, unless you go through the low-level API).

Note the parameters used in the `KeySpec`, which include the salt and the iteration count.

© Cryptosense 2019

The original PKCS#5 standard mentions 1024 iterations as a recommendation, but this was written in 2000. The most recent NIST standards recommend 10000 as the minimum value [11]. Additionally, HMAC-SHA-512 is preferred to HMAC-SHA-1 because it requires much more memory and so more resources to mount a hardware-based attack. Unfortunately none of the common providers support this directly [12].

Finally, note that here and elsewhere in Java crypto, the password is stored in a char array and not a `String` object. This is because strings are immutable, so if the password were stored in one, it would persist in the memory of the JVM. Using a char array allows the password to be cleared by calling `PBEKeySpec.clearPassword()`.

4.3. Password Storage

To protect their confidentiality, passwords are normally stored on disk as a hash value calculated in a similar way to password-based key derivation. Some objectives are the same, for example to make brute-force guessing hard, but in password storage we don't mind if the stored values exhibit some bias as long as that doesn't help guessing. For use as a cryptographic key, we want uniform random bits. So, one can use a PBKDF as a password-storage function, but not (in general) the other way round. Since Java crypto doesn't provide for a specific category of password-storage functions for the moment, one can use PBKDF2 with appropriate parameters to do the job. There are newer functions such as bcrypt, scrypt and Argon2 which offer particular features such as parameters which determine memory consumption, but support in standard providers is patchy (Bouncy Castle supports bcrypt and scrypt but not Argon2).

4.4. Password-based Encryption Schemes (PBEs)

In theory, after generating a key using a PBKDF, one could use any encryption scheme. However, in practice, a number of standardised schemes are used to promote interoperability. A lot of these date from export-restriction era, hence contain crypto that is now dangerously weak.

PKCS#5v1.5 standardised MD2, MD5 and SHA-1 for use with PBKDF1, and RC2 or DES for encryption. These are still included for compatibility in PKCS#5v2.0 [13]. There are referred to as PBES1 schemes, and are still available in many Java providers. Single DES (56 bits) is breakable in a few hours with specialized hardware, while RC2 (64 bits in PKCS#5v1.5) is also now considered insecure. The Oracle JCE provider also includes `PBEWithSHA1AndRC2_40`,an export-restriction strength encryption scheme that is now trivial to break. Note also that PBES1 ciphers implicitly specify use of CBC mode with PKCS5 padding and a fixed IV (derived from the password and salt). BouncyCastle meanwhile offers dozens of PBEs including some with hidden weaknesses. For example, `PBEWithMD5And128BitAESCBCOpenSSL` is a compatibility mode for OpenSSL, and applies just one iteration of MD5 to generate a key from the password.

© Cryptosense 2019

5. Key Management

"Key management is the hardest part of cryptography and often the Achilles' heel of an otherwise secure system."

B. Schneier, Applied Cryptography [14]

Key management encompasses key generation, distribution, storage, secure use, refresh, backup, audit etc. In practice this is where vulnerabilities often occur.

5.1. Key Generation

Secure key generation requires a secure random number generator. Java supplies `java.security.SecureRandom` for this purpose. It may block while waiting for entropy.

The KeyGenerator class in `javax.crypto` accepts parameters for keylength and/or randomness source. If no randomness source is given, the provider's implementation of `SecureRandom` source is used. If the provider doesn't have one, system `SecureRandom` is used.

If you mistakenly use plain `Random()` you get a Mersenne twister seeded by system time, which is not secure. This kind of bug is often found in applications [15]. Another common problem is hard-coded keys, bypassing key-generation completely. If the hard-coded value appears in the code this creates an immediate vulnerability. Another common mistake is to create a random value but then store it encoded as an ASCII string. The string is then used directly as the key value, giving a far smaller set of possible values for the key. Using Type 4 UUID strings as keys seems to be a particularly frequent error.

An extra consideration in Java crypto is to make sure your provider can generate the kind of keys you want. For example, the Oracle JCE provider by default restricts AES keys to 128 bit length because of US export regulations that still apply to some selected countries. Because this policy problem can arise in an application deployed in an unfamiliar environment, one often sees applications create a AES-256 bit key with a fixed zero value to test to see if that keysize is available.

If you encounter this problem, you need to replace files:

<java-home>/lib/security/US_export_policy.jar
<java-home>/lib/security/local_policy.jar

With those available at: http://www.oracle.com/technetwork/java/javase/downloads/jce8-download-2133166.html

5.2. Key Agreement

Java crypto contains provision for Diffie Hellman key agreement. Example code in the documentation [1] gives a hard-coded 1024 bit group "SKIP 1024" (from https://tools.ietf.org/html/draft-ietf-ipsec-skip-06). Since the LOGJAM work of 2015 (https://weakdh.org), we know that using standard groups facilitates precomputation attacks and that state-level adversaries may well be able to break 1024 bit DH in well-known groups. Hence it is considered safer to generate a fresh group. There is sample code in the Oracle documentation for this, but beware, the sample

© Cryptosense 2019

code creates a 512-bit group. DH in this size of group can be broken by a twitter bot (@DLogBot) whether it is a well-known group or not. The 2014 ENISA recommendations suggest 1024 bits as acceptable for legacy and 3072 bits for new applications.

5.3. Key Usage

A single key should not be used for more than one cryptographic operation. There are many examples of bad things that happen if this rule is not followed: using counter mode encryption and CBC-MAC with the same key can leak the plaintext, using the same key to encrypt data and other keys can leak those keys, etc. Some APIs manage keys with metadata indicating their roles that prevents the application from using them the wrong way. There are no such provisions in the Java crypto API, so applications must be carefully reviewed to check key use.

5.4. Key Storage

Java crypto provides a KeyStore class for managing keys and certificates. Secret values can be protected with a password before they are written to file. The encryption scheme and PBKDF used depends on the exact keystore employed, and each provider offers a number of different implementations, with very different security properties. Here we survey a few examples.

5.4.1. JKS

The original Java Keystore provided with the first Java crypto implementations was subject to export regulations, hence contains no encryption algorithm. Instead, private values are protected by XORing them against a keystream generated by SHA-1 hash of a password and salt. This is an extremely weak scheme permitting a variety of attacks. In particular, since only a single SHA1 hash is required to check a guess at the password, a brute-force dictionary attack can be mounted very efficiently. Recently, a cracking mode for JKS keystores has been added to the popular open-source password recovery tool Hashcat, making these attacks extremely fast and accessible to anyone [23].

5.4.2. JCEKS

After export regulations were lifted, the next iteration of the Java crypto engine included a new keystore, JCEKS, which employs triple-DES encryption to protect private key values and a proprietary PBKDF using 20 iterations of MD5. As discussed in section 4, 20 iterations of MD5 is considered far too little for security: recent NIST recommendations propose a minimum of 10000.

In 2018, after results on the insecurity of Java keystores were presented by Cryptosense and the Univeristy of Venice Ca′ Foscari at the NDSS conference [28], Oracle patched their provider to use 200 000 iterations by default.

5.4.3. BKS

The BouncyCastle provider offers two proprietary keystores, BKS and UBER. BKS also uses triple-DES encryption, The key is derived following the method from PKCS#12v1.0, which is deprecated (PKCS#12 v1.1, published December 2012, recommends using only PBKDF2). Note that the number of iterations of the PBKDF is a random value between 1024 and 2047. This is an unusual construct,

© Cryptosense 2019

and it's hard to say how this offers extra security since the number of iterations has to be recorded in clear in the file, along with the salt. In any case, 2047 is considered too low.

5.4.4. UBER

UBER uses encryption similar to BKS for the private key values. It additionally uses the same PBKDF and iteration count, but a different cipher (Twofish), for encrypting the whole keystore. Twofish was an unchosen finalist for the AES competition and so received a fair amount of cryptanalytic attention in the late 90's. No significant weaknesses have been found. This extra level of encryption avoids leaking information about certificate chains, keylengths etc. (other keystores only encrypt the private key values).

5.4.5. PKCS#12 Keystores

All providers have to offer a PKCS#12 keystore for interoperability [16]. Current PKCS#12 Keystores in the Oracle and Bouncy Castle providers follow the old PKCS#12v1.0 and use a deprecated SHA-1-based PBKDF specific to PKCS#12. This used to use 1024 iterations, but since the work presented at NDSS by Cryptosense and University Ca' Foscari researchers [cite again], the default has been increased to 50 000 iterations in both providers. Encryption of private key values is with Triple-DES. As of Java 9, PKCS#12 will become the default keystore type in Oracle JCE.

5.4.6. BCFKS

The upcoming Bouncy Castle FIPS version contains a new keystore that uses PBKDF2 with 1024 iterations of HMAC-SHA-512 to derive a 256-bit AES CCM key.

5.5. Keystore Integrity

Apart from protecting the confidentiality of the value of private keys, keystores can also protect integrity, i.e. you can make a password-based MAC of the whole keystore and then check it hasn't been tampered with.

In JKS an integrity signature is calculated over the encrypted keystore by SHA-1 hashing the password, the keystore, and the US-ASCII string "Mighty Aphrodite". Nobody seems to know why, but perhaps it dates the design back to 1995, when the Woody Allen film of the same name was in the cinemas, and the first Java beta release was made. This construction facilitates brute force attacks, since only a few SHA-1 calls (depending on the size of the keystore) are required to check a password guess. Strangely, JCEKS also uses this same method.

Meanwhile, BKS creates an HMAC using a key derived using the same PKCS#5v1.0 PBKDF as used for encryption. PKCS#12 Keystores do the same. However UBER does something a little different. Since it has already used the second keystore-wide password to encrypt the whole keystore, it simply calculates a SHA-1 hash of the keystore before encryption and uses this to check integrity after decryption has happened. This MAC-then-encrypt approach is generally considered a bad idea, since it can lead to attacks if, for example, there is a perceptible difference in behaviour (an error message, or execution time) between a decryption that fails because the padding is invalid, or a decryption that fails because the SHA-1 hash is invalid (a so-called padding oracle attack).

© Cryptosense 2019

		JKS	**JCEKS**	**PKCS12**
Provider		Sun	SunJCE	SunJSSE
Support for secret keys		no	yes	yes (since JDK 1.8)
Keys PBE	KDF	Custom (SHA1)	Custom (MD5)	PKCS12 (SHA1)
	Salt	160b	64b	160b
	Iterations	-	20 until 2018, now 200k	1024 until 2018, now 50k
	Cipher	Stream cipher	3DES (CBC)	3DES (CBC)
	Key size	-	192b	192b
Certificates PBE	KDF	x	x	PKCS12 (SHA1)
	Salt			160b
	Iterations			1024 until 2018, now 50k
	Cipher			RC2 (CBC)
	Key size			40b
Store Integrity	KDF	SHA1 with password	SHA1 with password	PKCS12 (SHA1)
	Salt			160b
	Iterations			1024 until 2018, now 100k
	Mechanism			HMAC (SHA1)

Figure 5 - Oracle JDK Keystore

© Cryptosense 2019

		BKS	UBER	BCFKS	BCPKS12
Provider		Bouncy Castle	Bouncy Castle	Bouncy Castle	Bouncy Castle
Support for secret keys		yes	yes	yes	no
Keys PBE	KDF Salt Iterations Cipher Key size	PKCS12 (SHA1) 160b 1024-2047 3DES (CBC) 192b	PKCS12 (SHA1) 160b 1024-2047 3DES (CBC) 192b	PBKDF2 (HMAC-SHA512) 512b 1024 AES (CCM) 256b	PKCS12 (SHA1) 160b 1024-2047 until 2018, now 50k 3DES (CBC) 192b
Certificates PBE	KDF Salt Iterations Cipher Key size	x	x	x	PKCS12 (SHA1) 160b 1024 until 2018, now 50k RC2/3DES (CBC) 40/192b
Store Encryption	KDF Salt Iterations Cipher Key size	x	PKCS12 (SHA1) 160b 1024-2047 Twofish (CBC) 256b	PBKDF2 (HMAC-SHA512) 512b 1024 AES (CCM) 256b	x
Store Integrity	KDF Salt Iterations	PKCS12 (SHA1) 160b 1024-2047 HMAC (SHA1)	SHA1 after decryption	PBKDF2 (HMAC-SHA512) 512b 1024 HMAC (SHA512)	PKCS12 (SHA1) 160b 1024 until 2018, now 100k HMAC (SHA1)

Figure 7 - Bouncy Castle Keystore

6. Implementing Protocols

As discussed at the start of section 2, it is quite possible to choose the right cryptographic algorithms and parameters, manage the keys correctly, and still be insecure because of a flaw at the protocol level. These kind of errors are very hard to spot in practice, and so it is considered

© Cryptosense 2019

unwise to "roll your own" protocol. It is preferable to use an established standard protocol that has been (and continues to be) subject to public analysis. This extends to implementations of protocols, which are also subject to subtle errors even if the specification of the protocol is secure. Java crypto providers furnish implementations for a number of standard protocols. We will look at two of the most widely used here.

6.1. SSL/TLS

The most widely used crypto protocol on the Internet is SSL/TLS. It is used for far more than just securing web traffic as part of the HTTPS protocol, including protecting mail and setting up VPNs. It is also a very old protocol that has gone through a series of revisions and has frequently been attacked.

In a nutshell, TLS is a Client/Server protocol to set up a secure channel, with a first key exchange/authentication handshake step followed by an encrypted channel protocol. Client authentication is optional. There are also lots of extensions including renegotiation, alert messages and so on.

Although configuring SSL/TLS correctly for a given environment is beyond the scope of this whitepaper, we will highlight here some particular issues around using the Java TLS libraries. For TLS configuration advice, try the Qualys SSL Labs Guide [24] or run a test at discovery.cryptosense.com.

Version	Year
SSLv2	1994
SSLv3	1996
TLS 1.0	1999
TLS 1.1	2006
TLS 1.2	2008
TLS 1.3	2018

Figure 8 - SSL/TLS Version History

6.1.1. JSSE

The JSSE (Java Secure Socket Extension) class provides [3] an abstract API to manage TLS connections and also access HTTPS. It uses JCE for all of its crypto algorithms. There is support for client (and server) certificates.

JSSE is very widely used, but how secure is it? The answer is, until recently, not very secure. There was no implementation of TLS1.2 (the latest version and the only one considered to offer good security) until Java 7 (recently backported to Java 6). There was no way to set server ciphersuite preference until Java 8 (this is an issue because a client might choose weak ciphersuites). There was no support for Diffie Hellman groups bigger than 768 bits until Java 6 (see section 5). There have also been several security bugs found in the library, including in January 2015 SKIP-TLS, which bypassed client and server authentication completely, and for which there is an exploit available on Metasploit. If you are using JSSE, make sure you are using the most recent version. Java 11, which became available in September 2018, contains support for TLS1.3.

© Cryptosense 2019

6.1.2. How to misuse JSSE

There are a number of more or less well-known pitfalls in using TLS libraries in general that can often be found in Java applications that employ JSSE [17].

The first is certificate verification. In JSSE, a `TrustManager` class handles whether a certificate is validated or not. Often in development environments, self-signed certificates are used on the test server, which means clients refuse to connect. Developers get around this by defining a new `TrustManager` that accepts anything. This is fine for development, but then often this change is not reverted before the application gets into production. A similar problem comes from over-riding `HostnameVerifier` to always return `true` for convenience in development. This means that a certificate will be accepted even if it is for a different site from the one we are connecting to. Finally, a similar problem arises from using `SSLSocketFactory` directly instead of `HTTPSURLConnection` wrapper for HTTPS: the hostname verification method is silently set to Null

6.1.3. JSSE Ciphersuites

JSSE in Java 8 supports 52 enabled ciphersuites and 40 disabled ones. The disabled suites include NULL (i.e. no encryption), EXPORT suites (40 bit RC2 etc.). The ciphersuites also control the key exchange method, and include anonymous Diffie-Hellman methods which lose authentication. These may be enabled in older versions of JSSE. Use `setEnabledCiphersuites` to choose, and choose wisely following e.g the SSL Labs guide [24].

6.2. S/MIME

S/MIME is a protocol for encrypted mail commonly used in enterprise environments. The Oracle S/MIME implementation uses JCE for crypto, and BouncyCastle also has an S/MIME implementation. S/MIME supports PKCS#12 for encryption of private keys (see section 5), as well as various ciphersuites for encryption, including insecure export ciphers such as 40-bit RC2. In Bouncy Castle S/MIME ciphersuites are selected as capabilities, while in Oracle `SmimeEnveloped` takes a cipher parameter.

7. How to Audit Java Crypto

We've seen in the preceding sections that there is plenty that can go wrong when a Java application uses the JCA/JCE Java crypto API. Typical business applications are often several hundred thousand lines of code. How can we audit the crypto used rigorously and efficiently for the kinds of problems we've highlighted here?

The Cryptosense method involves attaching our custom tracing agent to the JVM of the application under test (this is usually a one-line change to a config file). Then we run the applications existing integration tests or other test suites. This produces a large trace file (which can be compressed on the fly) recording all calls to the crypto API and the responses.

The resulting trace is then uploaded to our analyzer platform at analyzer.cryptosense.com. Once uploaded, you can select an analysis profile, or customize one of the existing ones to fit your own policy on keylengths and algorithms.

© Cryptosense 2019

Finally, we generate the report. Even for a large trace (10G uncompressed) report generation only takes a few minutes. Our analysis algorithms capture the rules described in this whitepaper and more. The report is then viewable as an interactive web application. Clicking on a finding on the left produces details of the analysis and a list of instances on the right. For each instance, clicking on the bug reveals the debug view, with the stacktrace of the call that triggered the analysis rule. This allows easy analysis in the source code.

To understand more completely risks associated with findings, we provide links to the Cryptosense Knowledge Base, where articles are available to explain the latest recommendations by standards bodies for keylengths and algorithms, common key-management errors, the theory behind password-based key derivation, and more.

You can try Cryptosense Analyzer for free by signing up at cryptosense.com/signup

Our analyzer platform has a number of other features to aid efficient analysis workflow:

» Dismiss (and restore) instances to produce a final version

» Filter on strings from stacktraces to blacklist or whitelist results

» Filter on rule categories (key length, algorithm, key-management, crypto usage)

» Export to CSV or to printable HTML view.

» Share a report in view-only mode

Cryptosense Analyzer is available in SaaS or installed on premise. To find out more or request a trial, send an email to java@cryptosense.com or call +33 (0)9 72 42 35 31.

8. About Cryptosense

Based in Paris, France, Cryptosense SA creates software to help users of cryptography stay secure. A spin-off of the French institute for computer science research (INRIA) and a collaborator of the University of Venice Ca' Foscari, Cryptosense's founders combine more than 40 years experience in research and industry.

Cryptosense software is used to test the cryptographic systems used to secure over half of daily interbank messages, over half of daily forex trades, and the world's largest financial transaction database. To find out more got to cryptosense.com.

9. References

[1] Java Cryptography Architecture (JCA) Reference Guide - https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html

[2] How to Implement a Provider in the Java Cryptography Architecture - https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/HowToImplAProvider.html

[3] Oracle JSSE - https://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html#SunJSSEProvider

[4] Bouncy Castle - https://www.bouncycastle.org/

© Cryptosense 2019

[5] Java Cryptography Extension - https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_73/rzaha/rzahajce.htm

[6] ENISA Algorithms, key size and parameters report 2014 - https://www.enisa.europa.eu/publications/algorithms-key-size-and-parameters-report-2014

[7] Evaluation of Some Blockcipher Modes of Operation, Phillip Rogaway, 2011 - http://web.cs.ucdavis.edu/~rogaway/papers/modes.pdf

[8] Wikipedia illustration of a tux penguin encrypted under ECB mode - https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#/media/File:Tux_ecb.jpg

[9] A Natural Language Approach to Automated Cryptanalysis of Two-time Pads - https://www.cs.jhu.edu/~jason/papers/mason+al.ccs06.pdf

[10] Serge Vaudenay. Security flaws induced by CBC padding – applications to SSL, IPSEC, WTLS ... In Lars R. Knudsen, editor, EUROCRYPT, volume 2332 of Lecture Notes in Computer Science, pages 534–546. Springer, 2002.

[11] DRAFT NIST Special Publication 800-63B Digital Authentication Guideline - https://pages.nist.gov/800-63-3/sp800-63b.html#sec5

[12] Parameter Choice for PBKDF2 - https://cryptosense.com/parameter-choice-for-pbkdf2/

[13] PKCS#5v2.0 - https://tools.ietf.org/html/rfc2898

[14] Applied Cryptography, Bruce Schneier, John Wiley & Sons, 1996, 2nd edition

[15] Security vulnerability + preferred disclosure - https://github.com/orientechnologies/orientdb/issues/5597

[16] PKCS#12 v1.1 - https://tools.ietf.org/html/rfc7292

[17] The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. Martin Georgiev et al. - http://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf

[18] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1". In CRYPTO 98.

[19] David Lazar, Haogang Chen, Xi Wang, Nickolai Zeldovich: Why does cryptographic software fail?: a case study and open problems. APSys 2014: 7:1-7:7

[20] Cryptosense blog article - https://cryptosense.com/weak-encryption-flaw-in-primefaces/

[21] NIST document 800-38 - http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf

[22] PadBuster - https://github.com/GDSSecurity/PadBuster

[23] Cryptosense blog article - https://cryptosense.com/cracking-java-keystores-with-hashcat/

[24] Github - https://github.com/ssllabs/research/wiki/SSL-and-TLS-Deployment-Best-Practices

[25] Project Wycheproof - https://github.com/google/wycheproof

© Cryptosense 2019

[26] Crypto Differential Fuzzer - https://github.com/kudelskisecurity/cdf

[27] Cryptosense blog article - https://cryptosense.com/the-end-of-triple-des/

[28] DBLP - https://dblp.org/rec/conf/ndss/FocardiPSST18

© Cryptosense 2019