ESAPI Symmetric Encryption

Overall Design

The overall design for ESAPI symmetric is to either use an “encrypt-then-MAC” approach or to use an authenticated encryption cipher mode. (Note that NIST refers to these as “combined” modes.)

This has been discussed in other ESAPI documentation. See [1] and [2] for details.

Key Derivation

Prior to ESAPI 2.1.1

Let K be a secret key of length L (measured in bits). This is our “key derivation key”. By default, if a key is not specified for a symmetric encryption operation, then this key is taken from the ESAPI property ‘Encryptor.MasterKey’.

When a non-authenticated encryption mode (i.e., a cipher mode not referenced in the property ‘Encryptor.cipher_modes.combined_modes’) is used, secret key K is used to derive two sub-keys, one of which is used for the actually encryption and one of which is used to provide authenticity via an HMAC. We calculate these keys as follows:

Let K_{enc} be the symmetric key used for encryption with the preferred symmetric cipher as specified by the ESAPI property ‘Encryptor.CipherTransformation’.

Let K_{mac} be the HMAC key used to ensure authenticity (i.e., data integrity).

Let PRF be a pseudo-random function used in the Key Derivation Function (KDF). [See ESAPI property ‘Encryptor.KDF.PRF’; the default is HmacSHA256.]

The Key Derivation Function, KDF, can conceptually be defined as:

KDF(prf_alg, keyDerivationKey, keySize, label, context)

The KDF is defined to closely follow section 5.1 of NIST SP 800-108.

Then we define K_{enc} and K_{mac} as follows:

K_{enc} = KDF(PRF, K, L, “encryption”, “”)

K_{mac} = KDF(PRF, K, L, “authenticity”, “”)

Where PRF is the pseudo –random function used by the KDF.

After ESAPI 2.1.1

Starting with ESAPI release 2.1.1 and in later releases, K_{enc} and K_{mac} are computed with a “context” other than the empty string. Specifically, are , K_{enc} and K_{mac} computed thusly:

K_{enc} = KDF(PRF, K, L, “encryption”, context)

K_{mac} = KDF(PRF, K, L, “authenticity”, context)

where ‘context’ is defined as:

context = cipherXform + Integer.toString(kdfVersion)

PRF_algorithm_name + Integer.toString(keySize);

where ‘+’ is Java String concatenation.

Because the cipher transformation string, the KDF version information, a PRF index #, and key size are all either password or derivable from information passed as part of the serialized ciphertext so the ‘context’ can be computed as appropriate when decrypted. Note that if the KDF version is less or equal to 20130914 (which corresponds to the ESAPI 2.1.0 release), then the empty string is used for ‘context’. However, if an adversary were to attempt a downgrade attempt on the KDF version, the incorrect MAC key (K_{mac}) would be computed and the authenticity verification would fail because of the encrypt-then-MAC design that ESAPI uses when a non-authenticated cipher mode is used for encryption. Similarly, if an adversary were to alter the cipher transformation to refer to a authenticated encryption (i.e., NIST “combined” mode), then one of those cipher modes would be used to attempt the decryption and they authenticity check built into that particular authenticated mode would ensure the authenticity. The assumption is that authenticated modes do not leak information when the IV or ciphertext is altered and furthermore traditional padding oracle attacks would fail because those modes almost (always?) streaming modes and therefore do not use padding. Also, should a padding mode be specified as part of an adversary’s alteration to the cipher transformation—e.g., “AES/CCM/PKCS5Padding”, the call to Cipher.getInstance() would likely fail with a NoSuchAlgorithmException. Hence no information would be leaked.

Encrypt, then MAC

By default, ESAPI uses an “Encrypt, then MAC” approach to provide authenticity whenever authenticated encryption mode is not used for encryption.

It works as follows.

Let P be some plaintext message.

Let IV be some randomly chosen Initialization Vector.

Let $\mathcal{E}$(k, IV, P) designate the encryption operation for plaintext message P for the chosen cipher under key k and initialization vector IV.

Let $\mathcal{D}$(k, IV, C) designate the encryption operation for ciphertext message C for the chosen cipher under key k and initialization vector IV.

Let C be the ciphertext resulting from encrypting plaintext P with key K_{enc} (see above "Key Derivation" section for details) using initialization vector IV (when appropriate; that is, when the cipher mode is not ECB…which of course should generally be avoided).

That is,

$$C = \mathcal{E}(K_{enc}, IV, P)$$

Let MAC be the message authentication code defined as

$$MAC = \text{HMAC-SHA1}(k, M)$$

for key k and message M.

Then we compute our MAC as for the encrypted data as:

$$MAC = \text{HMAC-SHA1}(K_{mac}, IV \mid\mid C)$$

where K_{mac} is as defined above (see "Key Derivation" section), and '||' denotes bitwise concatenation.

For the decryption operation, the MAC is recalculated and compared to the attached MAC. If the MACs are identical, ESAPI proceeds with the decryption operation. If it fails and exception is thrown. Care is taken to always throw the same exception message and to ensure all failure paths take the same amount of time to avoid providing any oracles that a remote adversary may be able to exploit. (Local adversaries running on the same server are another matter and ESAPI provides little protection against that, especially if the adversary has a privileged account.)

Serialized Ciphertext Representation

The serialized ciphertext representation has already been discussed in [2] and will not be repeated here.

References

[1] OWASP ESAPI For JavaEE 2.0: Design Goals in OWASP ESAPI Cryptography, URL http://owasp-esapi-java.googlecode.com/svn/trunk/documentation/esapi4java-core-2.0-crypto-design-goals.doc

[2] Format of portable serialization of org.owasp.esapi.crypto.CipherText object, URL http://owasp-esapi-java.googlecode.com/svn/trunk/documentation/esapi4java-core-2.0-ciphertext-serialization.pdf