SIV Encryption Security Assessment

Tim McLean | November 4, 2016

Summary

Connect2id[1] engaged the consultant, Tim McLean, to perform a security assessment of a Synthetic IV encryption library, `siv-mode`[2] .

The assessment identified two medium-severity vulnerabilities, described later in this report. Additionally, this report documents four areas for improvements that present a low or unknown risk, and provides guidelines on using `siv-mode` securely (see "Guidelines for Usage").

About the consultant

Tim McLean is an independent consultant who helps companies deploy cryptography securely. See https://www.chosenplaintext.ca for more information.

1 http://connect2id.com/
2 https://github.com/cryptomator/siv-mode

Vulnerability Details

1. Unsafe number of Associated Data (AD) elements permitted

Severity: Medium. **Impact**: Medium. **Exploitability**: Medium.

Status: Resolved. An exception is thrown if too many Associated Data elements are provided.

Impact: An attacker with sufficient control over the Associated Data inputs may be able to find other AD or plaintext inputs that have the same IV, allowing the attacker to forge new messages (bypassing SIV mode's tamper protection) or decrypt existing messages.

Description: The SIV mode of operation is only secure when there are fewer than $n-2$ Associated Data elements, where n is the block size in bits (see RFC 5297 section 7, final paragraph). This is a limitation of the *S2V* construction, which is only secure for up to $n-1$ input elements[3] . For AES, a maximum of 126 AD elements should be permitted.

3 http://web.cs.ucdavis.edu/~rogaway/papers/keywrap.pdf

Fortunately, exploitability of this issue is limited to applications that use a large (or attacker-controlled) number of AD elements.

Recommendations: Throw an exception if the number of AD elements passed to `encrypt` or `decrypt` exceeds `blockSizeInBits - 2`.

2. Use of table lookups in Bouncy Castle AES implementation

Severity: Medium. **Impact**: High. **Exploitability**: Low.

Status: Resolved. The default block cipher is now the JCE AES implementation instead of Bouncy Castle's AES implementation. JCE uses AES-NI when available, which is immune to these side channel attacks.

Impact: In the right circumstances, an attacker can use a timing attack to gradually learn information about each of the SIV keys, eventually allowing the attacker to reconstruct the keys entirely.

Description: A "cache timing attack" is where an attacker observes how long it takes to load data from memory in order to determine whether that value was stored in the CPU cache. If the operation was fast, then the attacker can conclude that the data was loaded from a cache. Based on the speed of the operation, the attacker can also determine from which level of cache the data was likely retrieved. These measurements can reveal a surprising amount of information about what is going on in the internals of a running program.

When the AES competition was held in the late 90s to choose a standard for symmetric encryption, cache timing attacks were not well known or understood. The AES algorithm that we use today was designed on the incorrect assumption that the time it takes to load data from memory is unrelated to the memory

location being accessed. Unfortunately, there is a relation: memory locations that have been accessed recently are more likely to have been cached.

In 2005, long after the AES competition had completed, Daniel Bernstein published a paper[4] describing how to mount a cache timing attack against AES in order to extract the encryption key from a server. More papers soon followed, improving on his original attack. In response, most libraries that used AES implemented some sort of countermeasure to prevent the attacks.

Unfortunately, all three of Bouncy Castle's AES implementations (AESEngine, AESLightEngine, and AESFastEngine) appear to be vulnerable to cache timing attacks. A bug report was filed in August 2015 to the Bouncy Castle project issue tracker[5] , but the project does not seem to have the resources to resolve the problem properly. Practical exploitation of this vulnerability in Bouncy Castle has been demonstrated[6] .

Recommendations: Replace the Bouncy Castle AES implementation with one that uses the CPU's hardware support for AES. Most modern processors have special instructions for hardware acceleration of AES. AES implementations using these instructions are immune to cache timing attacks.

4 https://cr.yp.to/antiforgery/cachetiming-20050414.pdf
5 http://www.bouncycastle.org/jira/browse/BJA-555
6 https://arxiv.org/pdf/1511.04897.pdf (page 11)

3. PGP private key in public repository

Severity: Unknown.

Status: Resolved. According to the maintainer, the key is protected by "a 30-character generated passphrase", which should be sufficient to make password cracking impossible.

Description: An encrypted PGP private key for "releases@cryptomator.org" was found in the public `siv-mode` repository. The key is password-protected. Because the encrypted key is publicly available, anyone can download a copy and attempt to crack the password while offline.

It is difficult to determine whether or not this presents a real risk without knowing how the password was selected. Attempts to crack the password with John the Ripper using a dictionary of popular passwords were unsuccessful. However, this is not representative of the time and resources available to a real-world attacker.

Recommendations: Avoid making private keys available publicly, even when password-protected. Alternatively, ensure that the password cannot be cracked by generating a password with at least 100 bits of entropy (e.g. a Diceware passphrase of 8 or more words, or a randomly generated 20-character alphanumeric password).

4. Overwriting keys after use is not effective

Severity: Low. **Impact**: Low. **Exploitability**: Low.

Status: Resolved.

Impact: Library users are given the impression that the two keys can be overwritten after use in order to eliminate those keys from memory. However, both keys are copied in memory many times during the encryption and decryption operations. Those copies are not overwritten before being garbage collected.

Description: Both of the encrypt and decrypt methods that accept `SecretKey` objects operate on a copy of the raw bytes inside each `SecretKey` object. Before returning, that copy is overwritten with `0x00` bytes.

This behaviour is documented: the methods that accept `SecretKey` objects claim to destroy the key bytes after use, while the methods that accept key byte arrays directly indicate that the calling method itself must overwrite the keys when done.

Unfortunately, this provides no effective protection. During the encryption and decryption operations many copies of each key are created in memory. For example, the `KeyParameter` constructor makes a copy of the provided key, and AESFastEngine copies the key into `WorkingKey` in the key schedule. Since

these copies are subject to normal garbage collection, erasing the other copies of the keys is not beneficial.

Recommendations: One option is to fork Bouncy Castle and patch all relevant code to ensure that keys are properly erased after use.

Realistically, however, overwriting keys after use is not strictly necessary. In languages like C that are not memory safe, erasing keys can serve as "defense in depth" in case an attacker is able to exploit a vulnerability to read other parts of a program's memory. In Java, barring a vulnerability in the JVM or a native library, this is not as useful.

Since Bouncy Castle cannot provide this kind of protection, the best option may be simply to remove the feature and document this limitation.

5. Undocumented limitation on block size

Severity: Unknown.

Status: Resolved.

Description: When instanciated with block ciphers with a block size other than 128 bits, SivMode will typically throw an exception, but will sometimes output an incorrect result.

Recommendations: Either implement support for other block sizes, or document this limitation. If the limitation is kept, throw an exception immediately if a block cipher implementation does not use the supported block size.

6. Arithmetic without proper checks for overflow

Severity: Informational.

Status: Resolved.

Description: In the implementation of counter mode encryption, a value `numBlocks` is computed based on the length of the plaintext input:

```
final int numBlocks = (plaintext.length + 15) / 16;
// ...
final byte[] x = new byte[numBlocks * 16];
```

An attacker can choose a length for `plaintext` such that `numBlocks` overflows and becomes negative. However, the first usage of `numBlocks` is to compute a length to initialize an array (`x`), at which point a `NegativeArraySizeException` is thrown, making this overflow unexploitable.

A refactor could make this exploitable by allowing an attacker to, for example, create discrepancies between what data is encrypted/decrypted vs what data is processed when computing the IV.

Recommendations: If it is possible for an arithmetic calculation to overflow, validate the inputs to that calculation to prevent overflow from occurring.

Example for `encrypt`:

```
// Validate length before the calculation occurs
if (plaintext.length > (Integer.MAX_VALUE - 15)) {
   throw new IllegalArgumentException("plaintext is too long");
}

final int numBlocks = (plaintext.length + 15) / 16;
```

Guidelines for Usage

- **Key generation**:

  Generate keys using a secure random number generator:

  ```
  byte[] key = new byte[32];
  new SecureRandom().nextBytes(key);
  ```

  Keys can be 16, 24, or 32 bytes each. I recommend 32-byte keys, unless performance is a concern.

- **Using the keys**:

  Unlike most encryption modes of operation, SIV mode requires not one key, but two: an encryption key (`ctrKey`) and an authentication key (`macKey`). It is best to treat these two keys as one inseparable large key. Once a pair of keys has been generated, those keys must always be used together. It is not safe to use either of those keys without the other. Additionally, the two keys must always be used for the same purpose – i.e. the `ctrKey` cannot be swapped with the `macKey`.

- **Additional Associated Data**:

  The SIV mode encrypt and decrypt operations are variable-argument methods, which allows them to be passed zero or more `additionalData` elements. Additional data is data that should not be encrypted, but should still be protected against tampering. This feature is optional and safe to ignore (by not providing any arguments for `additionalData`).

- **Usage limits**:

  A single AES-SIV key pair can only safely encrypt up to about 2^63 unique messages, at which point the key pair must be replaced. Most real world systems, however, will not reach this threshold.

Although reasonable efforts were made to ensure that this analysis was comprehensive, it is not guaranteed that no other vulnerabilities exist. As with any security assessment, it is possible that other security defects will be identified, especially as the state of the art in offensive research advances.