Cryptographic Findings Report for RavenDB

Prepared for Hibernating Rhinos Ltd

January 12, 2018

EDGESECURITY

www.edgesecurity.com

Abstract

Edge Security LLC was tasked by Hibernating Rhinos Ltd with examining the use of cryptography—specifically the use of libsodium and of X.509 certificates—in RavenDB over the course of 5 days during December 2017. The review of the RavenDB implementation produced several high risk findings. Given that the software is still pre-release, the findings below are not rated, and Edge Security LLC recommends that Hibernating Rhinos Ltd remediate *all* findings prior to publication.

Cryptographic Architecture

RavenDB deploys cryptography essentially on two different fronts: symmetric cryptography of all data on disk, and asymmetric cryptography via X.509 certificates as a means of authentication between clients and servers. This description follows the remediation of the findings in this report.

All symmetric encryption uses Daniel J. Bernstein's XChaCha20Poly1305 algorithm, as implemented in libsodium, with a randomized 192-bit nonce. While opting for XChaCha20 over ChaCha20 means more calls to the RNG and a computation of HChaCha20, it also means that there is no possibility of nonce-reuse, which means that it is considerably more resilient than ad-hoc designs that might make a best-effort attempt to avoid nonce-reuse, without ensuring it. Symmetric encryption covers the database main data store, index definitions, journal, temporary file streams, and secret handling. Such secret handling uses the Windows APIs for protected data, but only for a randomly generated encryption key, which is then used as part of the XChaCha20Poly1305 AEAD, to add a form of authentication. All long-term symmetric secrets are derived from a master key using the Blake2b hash function with a usage-specific context identifier.

At setup time, client and server certificates are generated. Clients trust the server's self-signed certificate, and the server trusts each client based on a fingerprint of each client's certificate. All data is exchanged over TLS, and TLS version failures for certificate failures are handled gracefully, with a webpage being shown indicating the failure status, rather than aborting the TLS handshake. Server certificates are optionally signed by Let's Encrypt using a vendor-specific domain name. Certificates are generated using BouncyCastle and are 4096-bit RSA.

Keys, nonces, and certificate private keys are randomly generated using the operating system's CSPRNG, either through libsodium or through BouncyCastle.

Finding 1: Nonce Reuse in IndexDefinitionBase

Retest Status: As of January 12, 2018, this finding has been remediated.

The `Raven.Server/Documents/Indexes/IndexDefinitionBase.cs` file makes an assumption that nonces are 64 bytes (eight times the size of `long`), when in reality nonces for the `chacha20poly1305` algorithm are 64 bits:

```
private static unsafe void EncryptStream(StorageEnvironmentOptions
 ↪  options, MemoryStream stream)
{
  var data = stream.ToArray();
  var nonce = Sodium.GenerateRandomBuffer(sizeof(long) * 8);
  var encryptedData = new byte[data.Length +
   ↪  Sodium.crypto_aead_chacha20poly1305_ABYTES()];

  fixed (byte* pData = data)
  fixed (byte* pEncryptedData = encryptedData)
  fixed (byte* pNonce = nonce)
  fixed (byte* pKey = options.MasterKey)
  {
    ulong cLen;
    var rc = Sodium.crypto_aead_chacha20poly1305_encrypt(
      pEncryptedData,
      &cLen,
      pData,
      (ulong)data.Length,
      null,
      0,
      null,
      pNonce,
      pKey
    );
```

Unfortunately the encryption function will discard the remaining 448 bits of the nonce, only using 64 bits. As a result, it is possible that the same combination of nonce and key will be used more than once, resulting in catastrophic key reuse.

The solution is to use the `crypto_secretbox_xchacha20poly1305` or `crypto_secretbox` function, which each take a 192-bit nonce, so that there is no concern over misuse, provided the nonces are generated using `randombytes_buf`.

Finding 2: Inaccurate Key Size in Secret Protection

Retest Status: As of January 12, 2018, this finding has been remediated.

The `Raven.Server/ServerWide/SecretProtection.cs` file defines a master key size as:

```
private const int KeySize = 512; // sector size
```

However, the two places it is used—passed to `chacha20poly1305` and to `crypto_kdf`—take 256-bit keys. Additionally, while these two functions *do* take the same size key, it is usually best practice to give different keys to different functions, so as to avoid domain collisions.

Finding 3: Nonce Reuse in Secret Protection

Retest Status: As of January 12, 2018, this finding has been remediated.

The `Raven.Server/ServerWide/SecretProtection.cs` file takes a variable length array parameter, `entropy`, and uses it as a nonce for `chacha20poly1305`, which assumes a fixed-size 64-bit nonce:

```
public byte[] Protect(byte[] secret, byte[] entropy)
{
  // [...]
  var protectedData = new byte[secret.Length +
   ↪  Sodium.crypto_aead_chacha20poly1305_ABYTES()];
  var key = _serverMasterKey.Value;

  if (entropy.Length < 8)
    throw new InvalidOperationException($"The provided entropy is too
     ↪  small. Should be at least 8 bytes but was {entropy.Length}
     ↪  bytes");

  fixed (byte* pSecret = secret)
  fixed (byte* pProtectedData = protectedData)
  fixed (byte* pEntropy = entropy)
  fixed (byte* pKey = key)
  {
    ulong cLen;
    var rc = Sodium.crypto_aead_chacha20poly1305_encrypt(
      pProtectedData,
      &cLen,
      pSecret,
      (ulong)secret.Length,
      null,
      0,
      null,
      pEntropy,
      pKey
    );
```

This code is called from various places, such as `Raven.Server/ServerWide/ServerStore.cs` and `tools/rvn/OfflineOperations.cs`, which both have code similar to:

```
var entropy = Sodium.GenerateRandomBuffer(256);
var protectedData = Secrets.Protect(hash, entropy);
```

Here they assume a 256-bit buffer as the entropy source, when this will be truncated down to 64-bits, resulting in catastrophic key reuse.

The solution is to use the `crypto_secretbox_xchacha20poly1305` or `crypto_secretbox` function, which each take a 192-bit nonce, so that there is no concern over misuse, provided the nonces are generated using `randombytes_buf`.

Finding 4: Non-Constant Time Secret Comparison

Retest Status: As of January 12, 2018, this finding has been remediated.

In `Raven.Server/ServerWide/ServerStore.cs`, secret values are compared using a non-constant time comparison function:

```
fixed (byte* pExistingKey = existingKey)
{
  bool areEqual = Sparrow.Memory.Compare(pKey, pExistingKey, key.Length)
   ↪  == 0;
  Sodium.ZeroMemory(pExistingKey, key.Length);
  if (areEqual)
  {
    Sodium.ZeroMemory(pKey, key.Length);
    return;
  }
}
```

An attacker observing the timing of such an operation and controlling one of the inputs may be able to guess bytes of the secret value.

The solution is to use the `sodium_memcmp` function for all secret-value memory comparisons.

Finding 5: Redundant or Missing Authentication in ServerStore

Retest Status: As of January 12, 2018, this finding has been remediated.

In `Raven.Server/ServerWide/ServerStore.cs`, an additional generic hash is computed and used for checking data integrity. In the case where this is passed to the built-in `chacha20poly1305` implementation, this additional hash is redundant. In the case where it is passed to `System.Security.Cryptography.ProtectedData`, it is insufficient as a secure authenticator—this is construction with known vulnerabilities. The function reads as follows:

```
if (Sodium.crypto_generichash(pHash, (UIntPtr)hashLen, pKey,
 ↪  (ulong)key.Length, null, UIntPtr.Zero) != 0)
  throw new InvalidOperationException("Failed to hash key");

Sparrow.Memory.Copy(pHash + hashLen, pKey, key.Length);

var entropy = Sodium.GenerateRandomBuffer(256);

var protectedData = Secrets.Protect(hash, entropy);

var ms = new MemoryStream();
ms.Write(entropy, 0, entropy.Length);
ms.Write(protectedData, 0, protectedData.Length);
ms.Position = 0;
```

Later, it is decrypted, and the computed hash is checked:

```
var data = Secrets.Unprotect(protectedData, entropy);
// [...]
if (Sodium.crypto_generichash(pHash, (UIntPtr)hashLen, pData + hashLen,
 ↪  (ulong)(data.Length - hashLen), null, UIntPtr.Zero) != 0)
  throw new InvalidOperationException($"Unable to compute hash for
   ↪  {name}");

if (Sodium.sodium_memcmp(pData, pHash, (UIntPtr)hashLen) != 0)
  throw new InvalidOperationException($"Unable to validate hash after
   ↪  decryption for {name}, user store changed?");

var buffer = new byte[data.Length - hashLen];
fixed (byte* pBuffer = buffer)
```

```
{
  Sparrow.Memory.Copy(pBuffer, pData + hashLen, buffer.Length);
}
```

As seen here, the use of `generichash` is either redundant or insecure. If `chacha20poly1305` is in use, no `generichash` should be computed. If `ProtectedData` is in use, the data should most likely be encrypted with `chacha20poly1305` first, and the native `ProtectedData` API can then be used for storing a randomly generated secret key.

Finding 6: Missing Authentication When Encrypting

Retest Status: As of January 12, 2018, this finding has been remediated.

In `Raven.Server/ServerWide/TempCryptoStream.cs`, `xchacha20` is used, which lacks an authenticator:

```
private void EncryptToStream(byte* pInternalBuffer)
{
  if (_bufferValidIndex == 0)
    return;

  fixed (byte* n = _nonce)
  fixed (byte* k = _key)
  {
    _stream.Seek(_startPosition + _blockNumber * _internalBuffer.Length,
     ↪  SeekOrigin.Begin);

    var rc = Sodium.crypto_stream_xchacha20_xor_ic(pInternalBuffer,
     ↪  pInternalBuffer, (ulong)_bufferValidIndex, n,
     ↪  (ulong)_blockNumber, k);
```

This means that an attacker can manipulate the data on disk. Since it appears that this is for ephemeral data, and a random key is generated for each usage, the right function to use is the `chacha20poly1305` function with the nonce set to zero (or the sequential index of each independently encrypted block) and the key randomly generated each time.

Finding 7: Weak Hashing in CryptoPager and Journal

Retest Status: As of January 12, 2018, this finding has been remediated.

The `Voron/Impl/Journal/WriteAheadJournal.cs` and `Voron/Impl/Paging/CryptoPager.cs` files make use of a non-cryptographically secure hash function called XXHash, even when authenticated encryption is in use. In `WriteAheadJournal.cs`, this is mostly wasteful rather than harmful:

```
if (performCompression)
  txHeader->Hash = Hashing.XXHash64.Calculate(txHeaderPtr +
   ↪  sizeof(TransactionHeader), (ulong)compressedLen,
   ↪  (ulong)txHeader->TransactionId);
else
  txHeader->Hash = Hashing.XXHash64.Calculate(txPageInfoPtr,
   ↪  (ulong)totalSizeWritten, (ulong)txHeader->TransactionId);

if (_env.Options.EncryptionEnabled)
  EncryptTransaction(txHeaderPtr);
```

In `CryptoPager.cs`, however, it may wind up being harmful, because the data could have manipulable hash collisions:

```
foreach (var buffer in state.LoadedBuffers)
{
  var checksum = Hashing.XXHash64.Calculate(buffer.Value.Pointer,
   ↪  (ulong)buffer.Value.Size);
  if (checksum == buffer.Value.Checksum)
    continue; // No modification

  // Encrypt the local buffer, then copy the encrypted value to the
   ↪  pager
  var pageHeader = (PageHeader*)buffer.Value.Pointer;
  EncryptPage(pageHeader);
```

It is possible here that the new value fails to be encrypted if the weak hash encounters a collision.

Finding 8: Nonce Reuse in CryptoPager and Journal

Retest Status: As of January 12, 2018, this finding has been remediated.

Both `Voron/Impl/Journal/WriteAheadJournal.cs` and `Voron/Impl/Paging/CryptoPager.cs` have very similar blocks regarding key derivation and nonce handling:

```
fixed (byte* ctx = Sodium.Context)
{
  var num = txHeader->TransactionId;
  if (Sodium.crypto_kdf_derive_from_key(subKey, (UIntPtr)32, (ulong)num,
   ↪  ctx, mk) != 0)
    throw new InvalidOperationException("Unable to generate derived
     ↪  key");
}

var npub = fullTxBuffer + TransactionHeader.SizeOf - macLen -
 ↪  sizeof(long);
if (*(long*)npub == 0)
  Sodium.randombytes_buf(npub, (UIntPtr)sizeof(long));
else
  (*(long*)npub)++;
```

And:

```
if (Sodium.crypto_kdf_derive_from_key(subKey, (UIntPtr)32, (ulong)num,
 ↪  ctx, mk) != 0)
  throw new InvalidOperationException("Unable to generate derived key");

var dataSize = (ulong)GetNumberOfPages(page) *
 ↪  Constants.Storage.PageSize;

var npub = (byte*)page + PageHeader.NonceOffset;
if (*(long*)npub == 0)
  Sodium.randombytes_buf(npub, (UIntPtr)sizeof(long));
else
  *(long*)npub = *(long*)npub + 1;
```

It appears that the intent of these blocks is to derive a subkey from a master key, based on a transaction ID, and then have sequential nonces within each ID. However, this code does not accomplish that. Choosing a random nonce when zero and incrementing it results in non-uniform sized nonce limits, and the possibility for nonce reuse. Further, it does not appear that transaction IDs are *globally unique*. Even between the two bits of code above, the same ID may be used for different data in different settings, resulting in subkey reuse. Finally, it is not immediately clear when `npub` will not be zeroed to start with, and when it is not zero, whether this corresponds to relevant information about previous nonces.

Rather than try to reason about a flawed design, it may be wise to rethink the design of these segments. It might be possible to derive subkeys and attempt to come up with a global ID for deriving the subkeys—where global is relative to the entire lifetime of the master key. However, it appears that this may not be possible given the overall architecture of the database. Therefore, the same conclusion as before applies: use the `crypto_secretbox_xchacha20poly1305` or `crypto_secretbox` function, which each take a 192-bit nonce, so that there is no concern over key/nonce misuse, provided the nonces are generated using `randombytes_buf`.

If it is not possible to store the full 192 bits, given restraints of the on-disk format, it may be permissible to use only 128 bits of random data, filling in the remaining 64 bits with hopefully-unique data implied by the existing state, such as the transaction ID.

Finding 9: Inconsistent Use of KDF and Master Key

Retest Status: As of January 12, 2018, this finding has been remediated.

RavenDB uses a master key, which is then used to encrypt various other things. This master key is passed to several different constructions and contexts. Sometimes it is used directly. Sometimes it is used indirectly, through the `crypto_kdf` variety of functions. It is best practice to have domain separation of key use—using a unique key for each type of cryptographic construction and data type.

For this it is recommended that the `crypto_kdf_derive_from_key` function be used. The `ctx` parameter should be an 8 character string defining the *type* of usage. For example, “indexkey” or “pagerkey” would make good values for `ctx`. For situations in which multiple different keys are desired within the same `ctx`—such as if the number of keys is excessively large or variable—use the 64-bit integer parameter `subkey_id` in an incrementing fashion.

Finding 10: Man in the Middle of Customer Domains

Retest Status: As of January 12, 2018, this finding has been documented.

RavenDB enables users to use a subdomain of `dbs.local.ravendb.net` with Let's Encrypt, by way of server-side assisted ACME dns-01 domain ownership verification. While this is convenient for customers, it does mean that Hibernating Rhinos Ltd has the ability to generate valid certificates, as well, regardless of user intent, which could potentially be used for a man in the middle attack. This means that all users of this feature rely on both the operational security and good will of Hibernating Rhinos Ltd. Depending on the security threat model, this may or may not be considered a vulnerability, though the assessment team does recommend clearly indicating this situation to the user in all cases.

Finding 11: Overly Broad Customer Domain Scope

Retest Status: As of January 12, 2018, this finding has been remediated.

Allowing customers to control subdomains of `dbs.local.ravendb.net` is potentially dangerous, because the top-level domain, `ravendb.net`, is also used for authenticated and potentially sensitive uses, such as `api.ravendb.net`. For that reason, it is advised that a *new* top-level domain—`ravendbusercontent.net` for example—be used instead. It is worth noting that both Google and Github use similar domains—`googleusercontent.com` and `githubusercontent.com`.

Finding 12: Non-high Strength RSA Keys

Retest Status: As of January 12, 2018, this finding has been remediated.

The `Raven.Server/Utils/CertificateUtils.cs` file makes use of 2048-bit RSA keys:

```
const int keyStrength = 2048;
```

While 2048-bit RSA is not currently publicly-known to be broken, and may very well not be for the foreseeable future, the NSA Suite B recommendations recently raised the minimum size to 3072 bits for top secret information. Though the reasons for doing so may be dubious and such government standards are generally not the final word on these issues, it is still advised to be *at least as strong as* the latest Suite B recommendations, by using either 3072-bit or 4096-bit RSA keys.

Finding 13: Collision in Certificate Serial Numbers

Retest Status: As of January 12, 2018, this finding has been remediated.

The `Raven.Server/Utils/CertificateUtils.cs` file generates serial numbers for new certificates with:

```
BigInteger serialNumber = BigIntegers.CreateRandomInRange(BigInteger.One,
 ↪   BigInteger.ValueOf(Int64.MaxValue), random);
certificateGenerator.SetSerialNumber(serialNumber);
```

This gives certificates a total number of $2^{63} - 2$ serial numbers, which is not a large enough range. Instead, a random 20-bytes serial number should be generated; 20 bytes is generally the maximum size of serial numbers that will be readable by all X.509 implementations.

Finding 14: Dubious Random Number Generation for RSA

Retest Status: As of January 12, 2018, this finding has been remediated.

The RSA key generation code in `Raven.Server/Utils/CertificateUtils.cs` makes use of this function for initializing a random number generator:

```
public static SecureRandom GetSeededSecureRandom()
{
  var buffer = new byte[32];
  using (var cryptoRandom = RandomNumberGenerator.Create())
  {
    cryptoRandom.GetBytes(buffer);
  }
  var randomGenerator = new VmpcRandomGenerator();
  randomGenerator.AddSeedMaterial(buffer);
  SecureRandom random = new SecureRandom(randomGenerator);
  return random;
}
```

Rather than using dubious generators, instead always use randomness directly from the operating system, via `System.Security.Cryptography.RandomNumberGenerator`.

Finding 15: Loose Certificate Extension Matching

Retest Status: As of January 12, 2018, this finding has been remediated.

The `Raven.Server/ServerWide/SecretProtection.cs` file verifies that certificates have certain extensions:

```
supported = (extensionString.Contains("Client Authentication") &&
 ↪   extensionString.Contains("Server Authentication"))
|| (extensionString.Contains("1.3.6.1.5.5.7.3.2") &&
 ↪   extensionString.Contains("1.3.6.1.5.5.7.3.1"));
```

Similarly `Raven.Client/Http/RequestExecutor.cs` does the same:

```
foreach (var extension in certificate.Extensions)
{
  if (extension.Oid.Value != "2.5.29.37") //Enhanced Key Usage extension
    continue;

  var extensionsString = new AsnEncodedData(extension.Oid,
   ↪   extension.RawData).Format(false);

  supported = extensionsString.Contains("1.3.6.1.5.5.7.3.2") ||
   ↪   extensionsString.Contains("Client Authentication"); // Client
   ↪   Authentication
}
```

However, by using `Contains`, it is possible that these strings are a substring of a different extension that is not the intended one. Rather than doing this, it is advisable to actually parse the entire list of extensions and check for explicit equality.