mongoose-encryption

2.1.2 • Public • Published 3 years ago

Readme

Code Beta

7 Dependencies

25 Dependents

28 Versions

mongoose-encryption

Simple encryption and authentication for mongoose documents. Relies on the Node `crypto` module. Encryption and decryption happen transparently during save and find. Rather than encrypting fields individually, this plugin takes advantage of the BSON nature of mongoDB documents to encrypt multiple fields at once.

How it Works

Encryption is performed using `AES-256-CBC` with a random, unique initialization vector for each operation. Authentication is performed using `HMAC-SHA-512`.

To encrypt, the relevant fields are removed from the document, converted to JSON, enciphered in `Buffer` format with the IV and plugin version prepended, and inserted into the `_ct` field of the document. Mongoose converts the `_ct` field to `Binary` when sending to mongo.

To decrypt, the `_ct` field is deciphered, the JSON is parsed, and the individual fields are inserted back into the document as their original data types.

To sign, the relevant fields (which necessarily include `_id` and `_ct`) are stably stringified and signed along with the list of signed fields, the collection name, and the plugin version. This signature is stored in `Buffer` format in the `_ac` field with the plugin version prepended and the list of signed fields appended. Mongoose converts the field to `Binary` when sending to mongo.

To authenticate, a signature is generated in the same fashion as above, and compared to the `_ac` field on the document. If the signatures are equal, authentication succeeds. If they are not, or if `_ac` is missing from the document, authentication fails and an error is passed to the callback.

During `save`, documents are encrypted and then signed. During `find`, documents are authenticated and then decrypted

Before You Get Started

Read the Security Notes below

Installation

```
npm install mongoose-encryption
```

Usage

Generate and store keys separately. They should probably live in environment variables, but be sure not to lose them. You can either use a single `secret` string of any length; or a pair of base64 strings (a 32-byte `encryptionKey` and a 64-byte `signingKey`).

A great way to securely generate this pair of keys is `openssl rand -base64 32; openssl rand -base64 64;`

Basic

By default, all fields are encrypted except for `_id`, `__v`, and fields with indexes

```
var mongoose = require('mongoose');
var encrypt = require('mongoose-encryption');

var userSchema = new mongoose.Schema({
    name: String,
    age: Number
    // whatever else
```

```
});

// Add any other plugins or middleware here. For example, middleware for hashing passwords

var encKey = process.env.SOME_32BYTE_BASE64_STRING;
var sigKey = process.env.SOME_64BYTE_BASE64_STRING;

userSchema.plugin(encrypt, { encryptionKey: encKey, signingKey: sigKey });
// This adds _ct and _ac fields to the schema, as well as pre 'init' and pre 'save' middleware,
// and encrypt, decrypt, sign, and authenticate instance methods

User = mongoose.model('User', userSchema);
```

And you're all set. `find` works transparently (though you cannot query fields that are encrypted) and you can make `New` documents as normal, but you should not use the `lean` option on a `find` if you want the document to be authenticated and decrypted. `findOne`, `findById`, etc..., as well as `save` and `create` also all work as normal. `update` will work fine on unencrypted and unauthenticated fields, but will not work correctly if encrypted or authenticated fields are involved.

Exclude Certain Fields from Encryption

To exclude additional fields (other than _id and indexed fields), pass the `excludeFromEncryption` option

```
// exclude age from encryption, still encrypt name. _id will also remain unencrypted
userSchema.plugin(encrypt, { encryptionKey: encKey, signingKey: sigKey, excludeFromEncryption: ['age'] });
```

Encrypt Only Certain Fields

You can also specify exactly which fields to encrypt with the `encryptedFields` option. This overrides the defaults and all other options.

```
// encrypt age regardless of any other options. name and _id will be left unencrypted
userSchema.plugin(encrypt, { encryptionKey: encKey, signingKey: sigKey, encryptedFields: ['age'] });
```

Authenticate Additional Fields

By default, the encrypted parts of documents are authenticated along with the `_id` to prevent copy/paste attacks by an attacker with database write access. If you use one of the above options such that only part of your document is encrypted, you might want to authenticate the fields kept in cleartext to prevent tampering. In particular, consider authenticating any fields used for authorization, such as `email`, `isAdmin`, or `password` (though password should probably be in the encrypted block). You can do this with the `additionalAuthenticatedFields` option.

```
// keep isAdmin in clear but pass error on find() if tampered with
userSchema.plugin(encrypt, {
    encryptionKey: encKey,
    signingKey: sigKey,
    excludeFromEncryption: ['isAdmin'],
    additionalAuthenticatedFields: ['isAdmin']
});
```

Note that the most secure choice is to include all non-encrypted fields for authentication, as this prevents tampering with any part of the document.

Nested Fields

Nested fields can be addressed in options using dot notation. For example, `encryptedFields: ['nest.secretBird']`

Renaming an Encrypted Collection

To guard against cross-collection attacks, the collection name is included in the signed block. This means that if you simply change the name of a collection in Mongo (and therefore update the model name in Mongoose), authentication would fail. To restore functionality, pass in the `collectionId` option with the old model name.

```
// used to be the `users` collection, now it's `powerusers`
poweruserSchema.plugin(encrypt, {
    encryptionKey: encKey,
    signingKey: sigKey,
    collectionId: `User` // this corresponds to the old model name
});

PowerUser = mongoose.model('PowerUser', poweruserSchema);
```

Encrypt Specific Fields of Sub Docs

You can even encrypt fields of sub-documents, you just need to add the `encrypt` plugin to the subdocument schema. *Subdocuments are not self-authenticated*, so you should consider adding the `encrypt` plugin to the parent schema as well for the authentication it provides, or if you would like to avoid that overhead, add the `encrypt.encryptedChildren` plugin to the parent schema if you will continue to work with documents following saves.

```
var hidingPlaceSchema = new Schema({
  latitude: Number,
  longitude: Number,
  nickname: String
});

hidingPlaceSchema.plugin(encrypt, {
  encryptionKey: encKey,
  signingKey: sigKey,
  excludeFromEncryption: ['nickname']
});

var userSchema = new Schema({
  name: String,
  locationsOfGold: [hidingPlaceSchema]
});

// optional but recommended: authenticate subdocuments from the parent document
userSchema.plugin(encrypt, {
  encryptionKey: encKey,
  signingKey: sigKey,
  additionalAuthenticatedFields: ['locationsOfGold'],
  encryptedFields: []
});

// alternative to the above. needed for continuing to work with document following a save
userSchema.plugin(encrypt.encryptedChildren);
```

The need for `encrypt.encryptedChildren` arises because of the order of middleware hooks in Mongoose 5.x.

Save Behavior

By default, documents are decrypted after they are saved to the database, so that you can continue to work with them transparently.

```
joe = new User ({ name: 'Joe', age: 42 });
joe.save(function(err){ // encrypted when sent to the database
                        // decrypted in the callback
  console.log(joe.name); // Joe
  console.log(joe.age); // 42
  console.log(joe._ct); // undefined
});
```

You can turn off this behavior, and slightly improve performance, using the `decryptPostSave` option.

```
userSchema.plugin(encrypt, { ..., decryptPostSave: false });
...
joe = new User ({ name: 'Joe', age: 42 });
joe.save(function(err){
  console.log(joe.name); // undefined
  console.log(joe.age); // undefined
  console.log(joe._ct); // <Buffer 61 41 55 62 33 ...
});
```

Secret String Instead of Two Keys

For convenience, you can also pass in a single secret string instead of two keys.

```
var secret = process.env.SOME_LONG_UNGUESSABLE_STRING;
userSchema.plugin(encrypt, { secret: secret });
```

Changing Options

For the most part, you can seemlessly update the plugin options. This won't immediately change what is stored in the database, but it will change how documents are saved moving forwards.

However, you cannot change the following options once you've started using them for a collection:

- `secret`
- `encryptionKey`
- `signingKey`
- `collectionId`

Instance Methods

You can also encrypt, decrypt, sign, and authenticate documents at will (as long as the model includes the plugin). `decrypt`, `sign`, and `authenticate` are all idempotent. `encrypt` is not.

```
joe = new User ({ name: 'Joe', age: 42 });
joe.encrypt(function(err){
  if (err) { return handleError(err); }
  console.log(joe.name); // undefined
  console.log(joe.age); // undefined
  console.log(joe._ct); // <Buffer 61 41 55 62 33 ...

  joe.decrypt(function(err){
    if (err) { return handleError(err); }
    console.log(joe.name); // Joe
    console.log(joe.age); // 42
    console.log(joe._ct); // undefined
  });
});



joe.age = 30

joe.sign(function(err){
  if (err) { return handleError(err); }
  console.log(joe.name); // Joe
  console.log(joe.age); // 30
  console.log(joe._ac); // <Buffer 61 fa 63 95 50

  joe.authenticate(function(err){
    if (err) { return handleError(err); }
    console.log(joe.name); // Joe
    console.log(joe.age); // 30
    console.log(joe._ac); // <Buffer 61 fa 63 95 50

    joe.age = 22

    joe.authenticate(function(err){ // authenticate without signing changes, error is passed to callback
        if (err) { return handleError(err); } // this conditional is executed
        console.log(joe.name); // this won't execute
  });
});
```

There are also `decryptSync` and `authenticateSync` functions, which execute synchronously and throw if an error is hit.

Getting Started with an Existing Collection

If you are using mongoose-encryption on an empty collection, you can immediately begin to use it as above. To use it on an existing collection, you'll need to either run a migration or use less secure options.

The Secure Way

To prevent tampering of the documents, each document is required by default to have a signature upon `find`. The class method `migrateToA()` encrypts and signs all documents in the collection. This should go without saying, but **backup your database** before running the migration below.

```
// This should be run in a separate migration script
userSchema.plugin(encrypt.migrations, { .... });
User = mongoose.model('User', userSchema);
User.migrateToA(function(err){
    if (err){ throw err; }
    console.log('Migration successful');
});
```

Following the migration, you can use the plugin as above.

The Quick Way

You can also start using the plugin on an existing collection without a migration, by allowing authentication to succeed on documents unsigned documents. This is less secure, but you can always switch to the more secure options later.

```
userSchema.plugin(encrypt, { requireAuthenticationCode: false, .... });
```

Migrating from Versions ≤ 0.11.0

If you're using an earlier version of mongoose-encryption, it is recommended that you upgrade. This version adds authentication, without which an attacker with write access to your database may be able to decrypt documents they should not otherwise be able to access, depending on the details of your application.

- Resolve breaking changes
  - Rename `key` -> `encryptionKey`
  - Add `signingKey` as 64-byte base64 string (generate with `openssl rand -base64 64`)
  - Run migrations
    - If you have encrypted subdocuments, first run the class method `migrateSubDocsToA()` on the parent collection

      ```
      // Only if there are encrypted subdocuments
      // Prepends plugin version to _ct
      userSchema.plugin(encrypt.migrations, { .... });
      User = mongoose.model('User', userSchema);
      User.migrateSubDocsToA('locationsOfGold', function(err){
          if (err){ throw err; }
          console.log('Subdocument migration successful');
      });
      ```

    - Run the class method `migrateToA()` on any encrypted collections (that are not themselves subdocuments)

      ```
      // Prepends plugin version to _ct and signs all documents
      userSchema.plugin(encrypt.migrations, { .... });
      User = mongoose.model('User', userSchema);
      User.migrateToA(function(err){
          if (err){ throw err; }
          console.log('Migration successful');
      });
      ```

- Suggestions
  - Set `additionalAuthenticatedFields` to include, at minimum, all fields involved in authorizing access to a document in your application
  - If using encrypted subdocuments, note additional recommendations here
- Deprecations
  - Rename `fields` -> `encryptedFields`
  - Rename `exclude` -> `excludeFromEncryption`

Pros & Cons of Encrypting Multiple Fields at Once

Advantages:

- All Mongoose data types supported via a single code path
- Faster encryption/decryption when working with the entire document
- Smaller encrypted documents

Disadvantages:

- Cannot select individual encrypted fields in a query nor unset or rename encrypted fields via an update operation
- Potentially slower in cases where you only want to decrypt a subset of the document

- Transactions including the entire encrypted/authenticated block are effectively enforced. Updating any encrypted or authenticated field forces them all to be marked as modified.

Security Notes

- Always store your keys and secrets outside of version control and separate from your database. An environment variable on your application server works well for this.
- Additionally, store your encryption key offline somewhere safe. If you lose it, there is no way to retrieve your encrypted data.
- Encrypting passwords is no substitute for appropriately hashing them. bcrypt is one great option. Here's one nice implementation. Once you've already hashed the password, you may as well encrypt it too. Defense in depth, as they say. Just add the mongoose-encryption plugin to the schema after any hashing middleware.
- If an attacker gains access to your application server, they likely have access to both the database and the key. At that point, neither encryption nor authentication do you any good.

How to Run Unit Tests

0. Install dependencies with `npm install` and **install mongo** if you don't have it yet
1. Start mongo with `mongod` (or `brew services start mongodb-community`)
2. Run tests with `npm test`

Security Issue Reporting / Disclaimer

None of the authors are security experts. We relied on accepted tools and practices, and tried hard to make this tool solid and well-tested, but nobody's perfect. Please look over the code carefully before using it (and note the legal disclaimer below). **If you find or suspect any security-related issues, please email us at security@cinchfinancial.com** and we will get right on it. For non-security-related issues, please open a Github issue or pull request.

Acknowledgements

Huge thanks goes out to **Cinch Financial** for supporting this plugin through version 1.0, as well as **@stash** for pointing out the limitations of earlier versions which lacked authentication and providing invaluable guidance and review on version 0.12.0.

Feel like contributing with different kinds of bits? Eth: 0xb53b70d5BE66a03E85F6502d1D060871a79a47f7

License

Keywords

mongoose mongo encrypt encryption sign authenticate authentication mongodb HMAC

Install

```
npm i mongoose-encryption
```

Repository

github.com/joegoldbeck/mongoose-encryption

Homepage

github.com/joegoldbeck/mongoose-encryption