

Designing a secure authentication package

(and the dangers of "there is ~~an app~~ a package for that")

Mattis Turin-Zelenko, chax.at

Outline

- Designing an authentication package
 - Design Goals
 - Lessons learned
- The TOTP "conspiracy"
 - Security in Open Source

Motivation

- Multiple projects all requiring authentication
- Code changes slightly per project, based on requirements
 - OpenID SSO via Google, Facebook, Microsoft, ...
 - SAML SSO
 - Two-factor authentication
- Let's standardize everything into a single package!

Goal 1 - Secure by default

News Analysis Sectors Themes Insights Events Reports Premium Insights Newsletters

Themes Artificial Intelligence Cloud Corporate Governance Cybersecurity Environmental Sustainability Internet of Things Robotics Social Responsibility

Unsecured MongoDB database leaves details from 200 million CVs exposed

Elien Daniel | January 11, 2019

Share 



Go deeper with GlobalData

Reports Business Process Outsourcing Market Size and Forecast (by Country, IT Solution area, Size Band and... GlobalData)

Reports IT Services Contracts - Quarterly Review Q1 2022 GlobalData

- Package is secure by default
- Throw error if secrets are not provided or insecure
- Provide options to make it less secure (e.g. for test/dev environments)

Goal 2 - Hard to use wrong

- Use scary names for scary things
- Provide documentation for *why* they are scary
- e.g. `UnsafePbkdf2HashingAlgorithm`,
`@UnsafeDisableCsrfCheckForNonGetMethod('We can disable
the CSRF check here, because...')`
- Code reviews immediately show potential problems
- Forbid unsafe settings when `NODE_ENV=production`

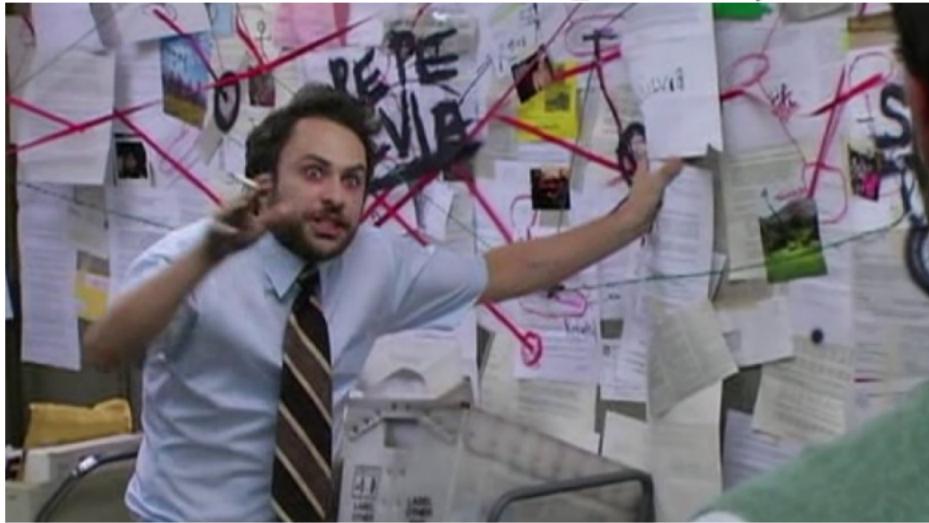
Goal 3 - Quick and easy to use

- Internal package allows us to tailor it to our common project architecture
- Package provides a NestJS module that can be imported and configured in <1h
- Extensive documentation - package will be used in *every* project!
- Proper releases, changelog, ...

Development Strategy

- Pair Programming
- Research existing open source packages...
- ...use them "under the hood" if feasible
- Use generic solutions if possible (e.g. one OpenID connector instead of Google/Microsoft/...)
 - ...if they are spec compliant
 - (looking at you, LinkedIn!)

The TOTP "conspiracy"



Background

- Frequent advice: Use "battle-tested" package
 - many common bugs already fixed
 - many people verified the code
 - ...but did they really?
- We wanted to use a totp package...
- ...but then we checked the code

Speakeasy

speakeasy 

2.0.0 • Public • Published 8 years ago

 Readme

 Code 

 1 Dependency

 363 Dependents

 7 Versions



Speakeeasy

two-factor authentication for node.js

 build unknown

 downloads 17M

 coverage 100%

 npm v2.0.0

Jump to — [Install](#) · [Demo](#) · [Two-Factor Usage](#) · [General Usage](#) · [Documentation](#) · [Contributing](#) ·

[License](#)

Speakeeasy is a one-time passcode generator, ideal for use in two-factor authentication, that supports Google Authenticator and other two-factor devices.

It is well-tested and includes robust support for custom token lengths, authentication windows, hash algorithms like SHA256 and SHA512, and other features, and includes helpers like a secret key generator.

Install

```
> npm i speakeeasy
```



Repository

 github.com/speakeasyjs/speakeeasy

Homepage

 github.com/speakeasyjs/speakeeasy

Weekly Downloads

253 008



Version

2.0.0

License

MIT

Can you spot the bug?

```
553     /**
554      * Generates a key of a certain length (default 32) from A-Z, a-z, 0-9, and
555      * symbols (if requested).
556      *
557      * @param {Integer} [length=32] The length of the key.
558      * @param {Boolean} [symbols=false] Whether to include symbols in the key.
559      * @return {String} The generated key.
560     */
561   exports.generateSecretASCII = function generateSecretASCII (length, symbols) {
562     var bytes = crypto.randomBytes(length || 32);
563     var set = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz';
564     if (symbols) {
565       set += '!@#$%^&*()<>?/[]{},.::;';
566     }
567
568     var output = '';
569     for (var i = 0, l = bytes.length; i < l; i++) {
570       output += set[Math.floor(bytes[i] / 255.0 * (set.length - 1))];
571     }
572     return output;
573   };
574
```

There are actually 3 bugs!

```
553     /**
554      * Generates a key of a certain length (default 32) from A-Z, a-z, 0-9, and
555      * symbols (if requested).
556      *
557      * @param {Integer} [length=32] The length of the key.
558      * @param {Boolean} [symbols=false] Whether to include symbols in the key.
559      * @return {String} The generated key.
560     */
561   exports.generateSecretASCII = function generateSecretASCII (length, symbols) {
562     var bytes = crypto.randomBytes(length || 32);
563     var set = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz';
564     if (symbols) {
565       set += '!@#$%^&*()<>?/[]{},:.;';
566     }
567
568     var output = '';
569     for (var i = 0, l = bytes.length; i < l; i++) {
570       output += set[Math.floor(bytes[i] / 255.0 * (set.length - 1))];
571     }
572     return output;
573   };
574 }
```

otplib

otplib 

12.0.1 • Public • Published 4 years ago

 Readme

 Code 

 3 Dependencies

 347 Dependents

 53 Versions

otplib

Time-based (TOTP) and HMAC-based (HOTP) One-Time Password library

 v12.0.1  passing  100%  26M  d.ts

- [About](#)
- [Features](#)
- [Quick Start](#)

Install

 npm i otplib



Repository

 github.com/yeojz/otplib

Homepage

 yeojz.otplib.dev

Weekly Downloads

475 975



Version

12.0.1

License

MIT

Unpacked Size

25.6 kB

Total Files

9

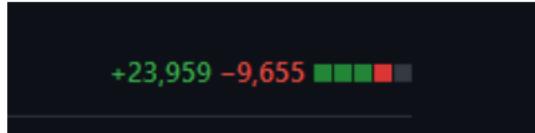
TOTP Algorithm Requirements

- RFC 4226

R6 - The algorithm MUST use a strong shared secret. The length of the shared secret MUST be at least 128 bits. This document RECOMMENDs a shared secret length of 160 bits.

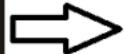
- 16 bytes minimum, 20 bytes recommended
- Secrets are base32 encoded (8 characters = 5 byte)
- Minimum: ~25 characters, recommended: 32 characters

otplib - PR#182



- A small change in those >30k changed lines...

```
 82      /**
 83      * Generates and encodes a secret key
 84      *
 85      * @param {number} length - secret key length (not encoded key length)
 86      * @return {string}
 87      * @see {@link module:impl/authenticator/secretKey}
 88      * @see {@link module:impl/authenticator/encodeKey}
 89      */
 90      generateSecret(len = 20) {
 91          if (!len) {
 92              return '';
 93          }
 94          const secret = secretKey(len, this.optionsAll);
 95          return encodeKey(secret);
 96      }
```



```
 /**
   * Reference: [[authenticatorGenerateSecret]]
   */
  public generateSecret(numberOfBytes = 10): Base32SecretKey {
      return authenticatorGenerateSecret<T>(numberOfBytes, this.allOptions());
  }
```

- ...no mention in the changelog...
- ...and no response from the maintainer since May 2022

passport-totp

passport-totp

0.0.2 • Public • Published 9 years ago

Readme

Code Beta

3 Dependencies

14 Dependents

2 Versions

Passport-TOTP

Passport strategy for two-factor authentication using a **TOTP** value.

This module lets you authenticate using a TOTP value in your Node.js applications. By plugging into Passport, TOTP two-factor authentication can be easily and unobtrusively integrated into any application or framework that supports **Connect**-style middleware, including **Express**. TOTP values can be generated by hardware devices or software applications, including **Google Authenticator**.

Note that in contrast to most Passport strategies, TOTP authentication requires that a user already be authenticated using an initial factor. Requirements regarding when to require a second factor are a matter of application-level policy, and outside the scope of both Passport and this strategy.

Install

```
> npm i passport-totp
```



Repository

🔗 github.com/jaredhanson/passport-totp

Homepage

🔗 github.com/jaredhanson/passport-totp

Weekly Downloads

15 451



Version

License

passport-totp "complete, working example"

- Can you spot the bugs? (Usage: `utils.randomKey(10);`)

```
1  exports.randomKey = function(len) {
2      var buf = []
3          , chars = 'abcdefghijklmnopqrstuvwxyz0123456789'
4          , charlen = chars.length;
5
6      for (var i = 0; i < len; ++i) {
7          buf.push(chars[getRandomInt(0, charlen - 1)]);
8      }
9
10     return buf.join('');
11 };
12
13     function getRandomInt(min, max) {
14         return Math.floor(Math.random() * (max - min + 1)) + min;
15     }
```

passport-totp "complete, working example"

- There are a few...

```
1 (exports.randomKey = function(len) {
2     var buf = []
3         , chars = 'abcdefghijklmnopqrstuvwxyz0123456789'
4         , charlen = chars.length;
5
6     for (var i = 0; i < len; ++i) {
7         buf.push(chars[getRandomInt(0, charlen - 1)]);
8     }
9
10    return buf.join('');
11};
12
13function getRandomInt(min, max) {
14    return Math.floor(Math.random() * (max - min + 1)) + min;
15}
```

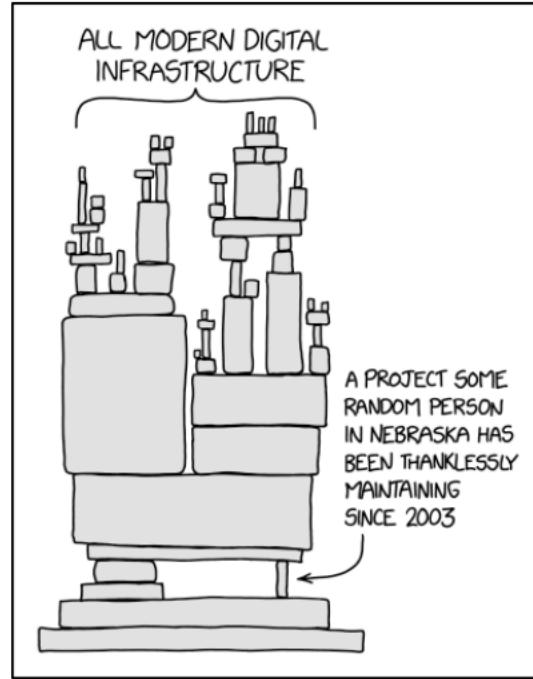
Debunking the "conspiracy" - why did this happen?

- Initial Google Authenticator libpam implementation used random 80 bit secrets
 - 80 bits in base32 result in 16 characters
 - Saving 16 characters in UTF-8 takes 16 bytes = 128 bit
 - ...but that's not the required 128 bits of randomness!
- Other packages copied this implementation (not reading RFC 4226)
- Google Authenticator fixed this in 2016

What can we do to prevent this?

- Verify dependencies yourself (especially if security-critical)
 - feasibility with 1205 dependencies?
 - reduce dependencies?
- Use paid services (auth0,...) from specialized companies
 - Vendor lock-in?
 - closed source?
- Other ideas?

How many projects look like this?



Summary

- Re-usable packages are great, especially for security-critical code
 - even if "just" for specific internal use cases
 - Design goal 1: Secure by default
 - Design goal 2: Hard to use wrong
 - Design goal 3: Quick & easy to use
- TOTP "conspiracy"
 - "Battle tested" does not equal bug-free
 - Verify dependencies, especially if security-critical