Spam Scanner is a Node.js anti-spam, email filtering, and phishing prevention tool and service. Built for **@ladjs**, **@forwardemail**, **@cabinjs**, **@breejs**, and **@lassjs**.

spamscanner.net

View license

213 stars **28** forks

master

Go to file

titanism 5.1.5 ... on Jun 13, 2022 **154**

View code

README.md

CI passing | code style XO | styled with prettier | made with lass | license not identifiable by github

Spam Scanner is the best anti-spam, email filtering, and phishing prevention service.

Spam Scanner is a drop-in replacement and the best alternative to SpamAssassin, rspamd, SpamTitan, and more.

Table of Contents

Foreword

Spam Scanner is a tool and service created after hitting countless roadblocks with existing spam-detection solutions. In other words, it's our current plan for spam.

Our goal is to build and utilize a scalable, performant, simple, easy to maintain, and powerful API for use in our service at Forward Email to limit spam and provide other measures to prevent attacks on our users.

Initially we tried using SpamAssassin, and later evaluated rspamd – but in the end we learned that all existing solutions (even ones besides these) are overtly complex, missing required features or documentation, incredibly challenging to configure; high-barrier to entry, or have proprietary storage backends (that could store and read your messages without your consent) that limit our scalability.

To us, we value privacy and the security of our data and users – specifically we have a "Zero-Tolerance Policy" on storing logs or metadata of any kind, whatsoever (see our Privacy Policy for more on that). None of these solutions honored this privacy policy (without removing essential spam-detection functionality), so we had to create our own tool – thus "Spam Scanner" was born.

The solution we created provides several Features and is completely configurable to your liking. You can learn more about the actual Algorithm below. Contributors are welcome.

Features

Spam Scanner includes modern, essential, and performant features that to help reduce spam, phishing, and executable attacks.

Naive Bayes Classifier

Our Naive Bayesian classifier is available in this repository, the npm package, and is updated frequently as it gains upstream, anonymous, SHA-256 hashed data from Forward Email.

It was trained with an extremely large dataset of spam, ham, and abuse reporting format ("ARF") data. This dataset was compiled privately from multiple sources.

Spam Content Detection

Provides an out of the box trained Naive Bayesian classifier (uses naivebayes and natural under the hood), which is sourced from hundreds of thousands of spam and ham emails. This classifier relies upon tokenized and stemmed words (with respect to the language of the email as well) into two categories ("spam" and "ham").

Phishing Content Detection

Robust phishing detection approach which prevents domain swapping, IDN homograph attacks, and more.

Executable Link and Attachment Detection

Link and attachment detection techniques that checks links in the message, "Content-Type" headers, file extensions, magic number, and prevents homograph attacks on file names – all against a list of executable file extensions.

Virus Detection

Using ClamAV, it scans email attachments (including embedded CID images) for trojans, viruses, malware, and/or other malicious threats.

NSFW Image Detection

We have plans to add NSFW image detection and opt-in toxicity detection as well.

Algorithm

In a nutshell, here is how the Spam Scanner algorithm works:

1. A message is passed to Spam Scanner, known as the "source".
2. In parallel and asynchronously, the source is passed to functions that detect the following:
   - Classification
   - Phishing
   - Executables
   - Arbitrary
   - Viruses
3. After all functions complete, if any returned a value indicating it is spam, then the source is considered to be spam. A detailed result object is provided for inspection into the reason(s).

We have extensively documented the API which provides insight into how each of these functions work.

Requirements

Note that you can simply use the Spam Scanner API for free at https://spamscanner.net instead of having to independently maintain and self-host your own instance.

Dependency	Description
Node.js	You must install Node.js in order to use this project as it is Node.js based. We recommend using nvm and installing the latest with `nvm install --lts`. If you simply want to use the Spam Scanner API, visit the website at https://spamscanner.net for more information.
Cloudflare	You can optionally set `1.1.1.3` and `1.0.0.3` as your DNS servers as we use DNS over HTTPS to perform a lookup on links, with a fallback to the DNS servers set on the system itself if the DNS over HTTPS request fails. We use Cloudflare for Family for detecting phishing and malware links.
ClamAV	You must install ClamAV on your system as we use it to scan for viruses. See ClamAV Configuration below.

ClamAV Configuration

Ubuntu

1. Install ClamAV:

```
sudo apt-get update
sudo apt-get install build-essential clamav-daemon clamav-freshclam clamav-unofficial-sig
sudo service clamav-daemon start
```

> You may need to run `sudo freshclam -v` if you receive an error when checking `sudo service clamav-daemon status`, but it is unlikely and depends on your distro.

2. Configure ClamAV:

```
sudo vim /etc/clamav/clamd.conf
```

```
-Example
+#Example

-#StreamMaxLength 10M
+StreamMaxLength 50M

+# this file path may be different on your OS (that's OK)

\-#LocalSocket /tmp/clamd.socket
\+LocalSocket /tmp/clamd.socket
```

```
sudo vim /etc/clamav/freshclam.conf
```

```
-Example
+#Example
```

3. Ensure that ClamAV starts on boot:

```
systemctl enable freshclamd
systemctl enable clamd
systemctl start freshclamd
systemctl start clamd
```

macOS

1. Install ClamAV:

```
brew install clamav
```

2. Configure ClamAV:

```
# if you are on Intel macOS
sudo mv /usr/local/etc/clamav/clamd.conf.sample /usr/local/etc/clamav/clamd.conf
```

```
# if you are on M1 macOS (or newer brew which installs to `/opt/homebrew`)
sudo mv /opt/homebrew/etc/clamav/clamd.conf.sample /opt/homebrew/etc/clamav/clamd.conf
```

```
# if you are on Intel macOS
sudo vim /usr/local/etc/clamav/clamd.conf

# if you are on M1 macOS (or newer brew which installs to `/opt/homebrew`)
sudo vim /opt/homebrew/etc/clamav/clamd.conf
```

```
-Example
+#Example

-#StreamMaxLength 10M
+StreamMaxLength 50M

+# this file path may be different on your OS (that's OK)

\-#LocalSocket /tmp/clamd.socket
\+LocalSocket /tmp/clamd.socket
```

```
# if you are on Intel macOS
sudo mv /usr/local/etc/clamav/freshclam.conf.sample /usr/local/etc/clamav/freshclam.conf

# if you are on M1 macOS (or newer brew which installs to `/opt/homebrew`)
sudo mv /opt/homebrew/etc/clamav/freshclam.conf.sample /opt/homebrew/etc/clamav/freshclam
```

```
# if you are on Intel macOS
sudo vim /usr/local/etc/clamav/freshclam.conf

# if you are on M1 macOS (or newer brew which installs to `/opt/homebrew`)
sudo vim /opt/homebrew/etc/clamav/freshclam.conf
```

```
-Example
+#Example
```

```
freshclam
```

3. Ensure that ClamAV starts on boot:

```
sudo vim /Library/LaunchDaemons/org.clamav.clamd.plist
```

> If you are on Intel macOS:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
```

```
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>org.clamav.clamd</string>
  <key>KeepAlive</key>
  <true/>
  <key>Program</key>
  <string>/usr/local/sbin/clamd</string>
  <key>ProgramArguments</key>
  <array>
    <string>clamd</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
</dict>
</plist>
```

> If you are on M1 macOS (or newer brew which installs to `/opt/homebrew`)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/
<plist version="1.0">
<dict>
  <key>Label</key>
  <string>org.clamav.clamd</string>
  <key>KeepAlive</key>
  <true/>
  <key>Program</key>
  <string>/opt/homebrew/sbin/clamd</string>
  <key>ProgramArguments</key>
  <array>
    <string>clamd</string>
  </array>
  <key>RunAtLoad</key>
  <true/>
</dict>
</plist>
```

4. Enable it and start it on boot:

```
sudo launchctl load /Library/LaunchDaemons/org.clamav.clamd.plist
sudo launchctl start /Library/LaunchDaemons/org.clamav.clamd.plist
```

5. You may want to periodically run `freshclam` to update the config, or configure a similar `plist` configuration for `launchctl` .

Install

npm:

```
npm install spamscanner
```

Usage

```
const fs = require('fs');
const path = require('path');

const SpamScanner = require('spamscanner');

const scanner = new SpamScanner();

//
// NOTE: The `source` argument is the full raw email to be scanned
// and you can pass it as String, Buffer, or valid file path
//
const source = fs.readFileSync(
  path.join(__dirname, 'test', 'fixtures', 'spam.eml')
);

// async/await usage
(async () => {
  try {
    const scan = await scanner.scan(source);
    console.log('scan', scan);
  } catch (err) {
    console.error(err);
  }
});

// then/catch usage
scanner
  .scan(source)
  .then(scan => console.log('scan', scan))
  .catch(console.error);

// callback usage
if (err) return console.error(err);
scanner.scan(source, (err, scan) => {
  if (err) return console.error(err);
  console.log('scan', scan);
});
```

API

`const scanner = new SpamScanner(options)`

The `SpamScanner` class accepts an optional `options` Object of options to configure the spam scanner instance being created. It returns a new instance referred to commonly as a `scanner`.

We have configured the scanner defaults to utilize a default classifier, and sensible options for ensuring scanning works properly.

For a list of all options and their defaults, see the index.js file in the root of this repository.

`scanner.scan(source)`

> **NOTE:** This is most useful method of this API as it returns the scanned results of a scanned message.

Accepts a required `source` (String, Buffer, or file path) argument which points to (or is) a complete and raw SMTP message (e.g. it includes headers and the full email). Commonly this is known as an "eml" file type and contains the extension `.eml`, however you can pass a String or Buffer representation instead of a file path.

This method returns a Promise that resolves with a `scan` Object when scanning is completed. You can also use this method with a second callback argument.

The scanned results are returned as an Object with the following properties (descriptions of each property are listed below):

```
{
  is_spam: Boolean,
  message: String,
  results: {
    classification: Object,
    phishing: Array,
    executables: Array,
    arbitrary: Array
  },
  links: Array,
  tokens: Array,
  mail: Object
}
```

Property	Type	Description
`is_spam`	Boolean	A value of `true` is returned if `category` property of the `results.classification` Object was determined to be `"spam"`, `results.phishing` was not empty, or `results.executables` was not empty – otherwise its value is `false`
`message`	String	A human-friendly message indicating why the `source` was classified as spam or ham (e.g. all messages/reasons from `results.classification`, `results.phishing`, and `results.executables` are joined together)
`results`	Object	An Object of properties that provide detailed information about the scan (very useful for debugging)

Property	Type	Description
`results.classification`	Object	An Object with `category` (String) and `probability` (Number) values returned based off the categorization of the `source` from the Naive Bayes classifier
`results.phishing`	Array	An Array of Strings indicating phishing attempts detected on the `source`
`results.executables`	Array	An Array of Strings indicating executable attacks detected on the `source`
`results.arbitrary`	Array	An Array of Strings indicating arbitrary spam-detection mechanisms detected on the `source`
`links`	Array	An Array of Strings that include all of the parsed and normalized links detected on the `source` . This is extremely useful for URL reputation management.
`tokens`	Array	**Debug only:** An Array of tokenized and stemmed words (parsed from the `source` , with respect to determined locale) used internally (for classification against the classifier) and exposed for debugging. This property is only returned when `debug` option in the instance is set to `true` .
`mail`	Object	**Debug only:** A parsed `mailparser.simpleParser` object used internally and exposed for debugging. This property is only returned when `debug` option in the instance is set to `true` .

`scanner.getTokensAndMailFromSource(source)`

Accepts a `source` argument (String, Buffer, or file path) to an email message (e.g. a `.eml` file). This method will automatically call `fs.readFile` internally if the `source` argument is a String and determined to be a valid path.

This method parses the `source` email message using mailparser's `simpleParser` function.

It then tokenizes and stems the message's subject, html, and text parts (with respect to the i18n determined language of the message, e.g. `en` , `es` , `jp` , `ru` , etc). See the `getTokens` method documentation for insight into how language is determined.

Currently Spam Scanner supports the following locales for tokenization, stemming, and stopword removal. Note that we select specific tokenizers, stemmers, and stopwords based off the detected language in the `source` .

Name	Locale
Arabic	`ar`
Danish	`da`

Name	Locale
Dutch	`nl`
English	`en`
Finnish	`fn`
Farsi	`fa`
French	`fr`
German	`de`
Hungarian	`hr`
Indonesian	`in`
Italian	`it`
Japanese	`ja`
Norwegian	`nb` , `nn`
Polish	`po`
Portuguese	`pt`
Spanish	`es`
Swedish	`sv`
Romanian	`ro`
Russian	`ru`
Tamil	`ta`
Turkish	`tr`
Vietnamese	`vi`
Chinese	`zh`

This method returns a Promise that resolves with a `{ tokens, mail }` Object. You can also use this method with a second callback argument.

Note that `tokens` is an Array of parsed tokenized and stemmed words, and `mail` is the `simpleParser` parsed mail Object.

This is the core internal method used for building the Bag-of-words model which is then fed to the classifier for categorization.

See classifier.js for an example implementation of this method (e.g. the one used in generating the default classifier dataset).

`scanner.getClassification(tokens)`

Accepts a `tokens` Array of tokens parsed from the `tokens` property returned in the Object from `scanner.getTokensAndMailFromSource` (see above).

This method returns a Promise that resolves with the classification determined from naivebayes.

In order to defend against gibberish attack vectors, classification is limited to a limited bag of words approach by. The default value is `20000` words per category. In other words the most `20000` common spam words and `20000` common ham words are used to determine the classification of the original source.

We have plans to further refine the classifier to strip all gibberish by testing against Wikimedia (or Google AI) datasets of word dictionaries of every language. This is not an easy feat to pull off, however we have concrete plans for how we will approach this.

`scanner.getPhishingResults(mail)`

Accepts a `mailparser.simpleParser` parsed mail Object.

This method returns a Promise that resolves with an Array of messages (if any) that indicates that links parsed from the message were detected to be phishing attempts. You can also use this method with a second callback argument.

This method also prevents the common IDN homograph attacks. If *any* link is detected to start with the string `xn--` (e.g. after conversion from `punycode.toASCII`) then it is detected as phishing.

A common example of this is a link of `paypal.com` which when converted to ASCII is `xn--aypal-uye.com` – but when rendered it looks almost identical (if not identical) to `paypal.com` .

This method checks against Cloudflare for Families servers for both adult-related content, malware, and phishing. This means we do two separate DNS over HTTPS requests to `1.1.1.2` for malware and `1.1.1.3` for adult-related content. You can parse the messages results Array for messages that contain "adult-related content" if you need to parse whether or not you want to flag for adult-related content or not on your application.

If you are using Cloudflare for Families DNS servers as mentioned in Requirements), then if there are any HTTPS over DNS request errors, it will fallback to use the DNS servers set on the system for lookups, which would in turn use Cloudflare for Family DNS. (using DNS over HTTPS with a fallback of dns.resolve4) – and if it returns `0.0.0.0` then it is considered to be phishing.

We actually helped Cloudflare in August 2020 to update their documentation to note that this result of `0.0.0.0` is returned for maliciously found content on FQDN and IP lookups.

`scanner.getExecutableResults(mail)`

Accepts a `mailparser.simpleParser` parsed mail Object.

Note that this method detects (with respect to executables.json using "Content-Type" header detection, file extension detection, and magic number detection.

This method returns a Promise that resolves with an Array of messages (if any) that indicate that links and/or attachments parsed from the message were dangerous (e.g. contained executable files or links to executable files). You can also use this method with a second callback argument.

This method also takes into consideration that the file extension and name could have a homograph attack by using `punycode.toASCII` on the file name.

It also scans against links in the message itself for links to executables.

`scanner.getTokens(str, locale, isHTML = false)`

Accepts a `str` (String) and optional `locale` (String - valid i18n locale according to i18n-locales) and `isHTML` parameters. If `isHTML` is set to `true`, then that indicates that the String passed as `str` is in HTML format.

Returns an Array of SHA-256 hashed tokenized and stemmed words, with respect to the passed, detected, or default locale. If `config.debug` is `true`, then the values are not returned as hashed values (e.g. this is useful in testing and debugging).

Note that this is "smart" in the sense it will parse the "Content-Language" header of the message, the `content` attribute of the HTML message's `<meta http-equiv="Content-Language" content="en-us">`, or the `lang` attribute of `<html lang="en">`.

After parsing the language of the message, it will then use the package franc to attempt to determine the language of the message (as long as the message has at least 5 characters, which is configurable).

Most importantly the following types of tokens are replaced with cryptographically generated random hashes:

- Emojis (this includes Github-flavored emoji written in Markdown and all Unicode emojis)
- MAC addresses
- Credit cards
- Bitcoin addresses
- Phone numbers
- Hex colors
- Initialisms
- Abbreviations
- Email addresses
- Links
- Integers and floating point values
- Currencies

Note that the replacements for these types of tokens are whitelisted when stemming is performed.

Contractions are also expanded, e.g. "they're" becomes two tokens, "they" and "are", which are then stemmed accordingly.

`scanner.getArbitraryResults(mail)`

Accepts a `mailparser.simpleParser` parsed mail Object.

This method will test the message against arbitrary spam-detection reasons, such as GTUBE.

Returns an Array of messages (if any) that indicate that parts of the message were detected to be spam-related for arbitrary reasons. You can also use this method with a second callback argument.

`scanner.getVirusResults(mail)`

Accepts a `mailparser.simpleParser` parsed mail Object.

This method returns a Promise that resolves with an Array of messages (if any) that indicate that attachments parsed from the message were dangerous (e.g. contained trojans, viruses, malware, and/or other malicious threats). You can also use this method with a second callback argument.

ClamAV is used internally with this method, in order to scan the attachments (in parallel).

`scanner.parseLocale(locale)`

Accepts a `locale` and returns it as a lowercase string with affixed localizations removed (e.g. `en-US` becomes `en` and `en_US` becomes `en` as well).

Caching

By default a `memoize` config option is passed with an infinite limit for adult-content and malware lookups.

You can configure either the `memoize` or `client` options, with `memoize` being an Object of options to pass to memoizee, and `client` being an instance of Redis, such as one created with @ladjs/redis.

Refer to the tests for examples of both implementations. If you go with the approach of `memoize` , then you should set a `size` option such as:

```
const scanner = new SpamScanner({
  // ...
  memoize: {
    // since memoizee doesn't support supplying mb or gb of cache size
    // we can calculate how much the maximum could potentially be
    // the max length of a domain name is 253 characters (bytes)
    // and if we want to store up to 1 GB in memory, that's
    // `Math.floor(bytes('1GB') / 253)` = 4244038 (domains)
    // note that this is per thread, so if you have 4 core server
    // you will have 4 threads, and therefore need 4 GB of free memory
    size: Math.floor(bytes('1GB') / 253)
  }
});

// ...
```

Note that in Forward Email we use the `client` approach as we have multiple threads across multiple servers running, and in-memory caching would not be efficient.

Debugging

Spam Scanner has built-in debug output via `util.debuglog('spamscanner')` . You can also pass `debug: true` to your instance to get more verbose output.

This means you can run your app with `NODE_DEBUG=spamscanner node app.js` to get useful debug output to your console.

Contributors

Name	Website
Nick Baugh	http://niftylettuce.com/
Shaun Warman	http://shaunwarman.com/

References

- CC-CEDICT is licensed under Creative Commons Attribution-ShareAlike 4.0 International License.
- https://www.digitalocean.com/community/tutorials/how-to-setup-exim-spamassassin-clamd-and-dovecot-on-an-arch-linux-vps
- https://medium.com/@wingsuitist/set-up-clamav-for-osx-1-the-open-source-virus-scanner-82a927b60fa3
- http://redgreenrepeat.com/2019/08/09/setting-up-clamav-on-macos/
- https://paulrbts.github.io/blog/software/2017/08/18/clamav/
- https://gist.github.com/zhurui1008/4fdc875e557014c3a34e

License

Releases 52

v5.1.5 Latest
on Jun 13, 2022

+ 51 releases

Packages

No packages published

Used by 17

+ 9

Contributors 4

- **niftylettuce**
- **titanism**
- **marquicodes** Kyriakos Markakis
- **dependabot[bot]**

Languages

● **JavaScript** 98.2% ● **HTML** 1.6% ● **Shell** 0.2%