cure53 / **DOMPurify** Public

Code | Issues | Pull requests 1 | Actions | Projects | Wiki | Security and quality 8 | Insights

main | 5 Branches | 136 Tags | Go to file | Go to file | Code

dependabot[bot] build(deps-dev): bump serialize-javascript from 7.0.4 to 7.0.5 (#1223)

883ac15 · 3 days ago

.github	fix: Removed Node 16 test target as it breaks	last week
.settings	Fix #489	6 years ago
demos	See #931	2 years ago
dist	fix: Fixed a problem with the type defition p…	3 weeks ago
scripts	fix: Fixed a problem with the type defition p…	3 weeks ago
src	fix: Expanded the regex ever so slightly to …	last month
test	test: Testing whether the Browser Stack "lat…	last week
typescript	Improved display of verification results.	last year
website	chore: Preparing 3.3.3 release	3 weeks ago
.babelrc	bump browsers versions to support for…of …	3 years ago
.editorconfig	Refactor to ES201* syntax and fix UMD reli…	9 years ago
.gitattributes	fix: prettier windows end of line not lf (#1186)	3 months ago
.gitignore	Update .gitignore	5 years ago
.nvmrc	chore: updated nvmrc	5 years ago
.prettierignore	chore: add prettierrc and *ignore	9 years ago
.prettierrc	chore: add prettierrc and *ignore	9 years ago
LICENSE	Update LICENSE	2 years ago
README.md	Update README.md (#1222)	last week
SECURITY.md	chore: update the link to bug bounty page	4 years ago
bower.json	chore: Preparing 3.3.3 release	3 weeks ago
package-lock.json	build(deps-dev): bump serialize-javascript f…	3 days ago
package.json	build(deps): bump serialize-javascript and …	3 weeks ago
rollup.config.js	Revert "chore: update dependencies (#118…	3 months ago
tsconfig.json	Fix sourcemaps.	last year

README | License | Security

DOMPurify

npm package 3.3.3 | Build and Test passing | downloads 140M/month | gzip 8.9 kB | dependents 604.4K | Cloudback succeeded

DOMPurify is a DOM-only, super-fast, uber-tolerant XSS sanitizer for HTML, MathML and SVG.

It's also very simple to use and get started with. DOMPurify was started in February 2014 and, meanwhile, has reached version **v3.3.3**.

DOMPurify runs as JavaScript and works in all modern browsers (Safari (10+), Opera (15+), Edge, Firefox and Chrome - as well as almost anything else using Blink, Gecko or WebKit). It doesn't break on MSIE or other legacy browsers. It simply does nothing.

Note that DOMPurify v2.5.9 is the latest version supporting MSIE. For important security updates compatible with MSIE, please use the 2.x branch.

Our automated tests cover 39 different browsers right now, more to come. We also cover Node.js v20.x, v22.x, 24.x and v25.x, running DOMPurify on jsdom. Older Node versions are known to work as well, but hey... no guarantees.

DOMPurify is written by security people who have vast background in web attacks and XSS. Fear not. For more details please also read about our Security Goals & Threat Model. Please, read it. Like, really.

What does it do?

DOMPurify sanitizes HTML and prevents XSS attacks. You can feed DOMPurify with string full of dirty HTML and it will return a string (unless configured otherwise) with clean HTML. DOMPurify will strip out everything that contains dangerous HTML and thereby prevent XSS attacks and other nastiness. It's also damn bloody fast. We use the technologies the browser provides and turn them into an XSS filter. The faster you browser, the faster DOMPurify will be.

How do I use it?

It's easy. Just include DOMPurify on your website.

Using the unminified version (source-map available)

```
<script type="text/javascript" src="dist/purify.js"></script>
```

Using the minified and tested production version (source-map available)

```
<script type="text/javascript" src="dist/purify.min.js"></script>
```

Afterwards you can sanitize strings by executing the following code:

```
const clean = DOMPurify.sanitize(dirty);
```

Or maybe this, if you love working with Angular or alike:

```
import DOMPurify from 'dompurify';

const clean = DOMPurify.sanitize('<b>hello there</b>');
```

The resulting HTML can be written into a DOM element using `innerHTML` or the DOM using `document.write()`. That is fully up to you. Note that by default, we permit HTML, SVG **and** MathML. If you only need HTML, which might be a very common use-case, you can easily set that up as well:

```
const clean = DOMPurify.sanitize(dirty, { USE_PROFILES: { html: true } });
```

Is there any foot-gun potential?

Well, please note, if you *first* sanitize HTML and then modify it *afterwards*, you might easily **void the effects of sanitization**. If you feed the sanitized markup to another library *after* sanitization, please be certain that the library doesn't mess around with the HTML on its own.

Okay, makes sense, let's move on

After sanitizing your markup, you can also have a look at the property `DOMPurify.removed` and find out, what elements and attributes were thrown out. Please **do not use** this property for making any security critical decisions. This is just a little helper for curious minds.

Running DOMPurify on the server

DOMPurify technically also works server-side with Node.js. Our support strives to follow the Node.js release cycle.

Running DOMPurify on the server requires a DOM to be present, which is probably no surprise. Usually, jsdom is the tool of choice and we **strongly recommend** to use the latest version of *jsdom*.

Why? Because older versions of *jsdom* are known to be buggy in ways that result in XSS *even if* DOMPurify does everything 100% correctly. There are **known attack vectors** in, e.g. *jsdom v19.0.0* that are fixed in *jsdom v20.0.0* - and we really recommend to keep *jsdom* up to date because of that.

Please also be aware that tools like happy-dom exist but **are not considered safe** at this point. Combining DOMPurify with *happy-dom* is currently not recommended and will likely lead to XSS.

Other than that, you are fine to use DOMPurify on the server. Probably. This really depends on *jsdom* or whatever DOM you utilize server-side If you can live with that, this is how you get it to work:

```
npm install dompurify
npm install jsdom
```

For *jsdom* (please use an up-to-date version), this should do the trick:

```
const createDOMPurify = require('dompurify');
const { JSDOM } = require('jsdom');

const window = new JSDOM('').window;
const DOMPurify = createDOMPurify(window);
const clean = DOMPurify.sanitize('<b>hello there</b>');
```

Or even this, if you prefer working with imports:

```
import { JSDOM } from 'jsdom';
import DOMPurify from 'dompurify';

const window = new JSDOM('').window;
const purify = DOMPurify(window);
const clean = purify.sanitize('<b>hello there</b>');
```

If you have problems making it work in your specific setup, consider looking at the amazing isomorphic-dompurify project which solves lots of problems people might run into.

```
npm install isomorphic-dompurify
```

```
import DOMPurify from 'isomorphic-dompurify';

const clean = DOMPurify.sanitize('<s>hello</s>');
```

Is there a demo?

Of course there is a demo! Play with DOMPurify

What if I find a *security* bug?

First of all, please immediately contact us via email so we can work on a fix. PGP key

Also, you probably qualify for a bug bounty! The fine folks over at Fastmail use DOMPurify for their services and added our library to their bug bounty scope. So, if you find a way to bypass or weaken DOMPurify, please also have a look at their website and the bug bounty info.

Some purification samples please?

How does purified markup look like? Well, the demo shows it for a big bunch of nasty elements. But let's also show some smaller examples!

```
DOMPurify.sanitize('<img src=x onerror=alert(1)//>'); // becomes <img src="x">
DOMPurify.sanitize('<svg><g/onload=alert(2)//<p>'); // becomes <svg><g></g></svg>
DOMPurify.sanitize('<p>abc<iframe//src=jAva&Tab;script:alert(3)>def</p>'); // becomes <p>abc</p>
DOMPurify.sanitize('<math><mi//xlink:href="data:x,<script>alert(4)</script>">'); // becomes <math><mi></mi></math>
DOMPurify.sanitize('<TABLE><tr><td>HELLO</tr></TABL>'); // becomes <table><tbody><tr><td>HELLO</td></tr></tbody></tab
DOMPurify.sanitize('<UL><li><A HREF=//google.com>click</UL>'); // becomes <ul><li><a href="//google.com">click</a></l
```

What is supported?

DOMPurify currently supports HTML5, SVG and MathML. DOMPurify per default allows CSS, HTML custom data attributes. DOMPurify also supports the Shadow DOM - and sanitizes DOM templates recursively. DOMPurify also allows you to sanitize HTML for being used with the jQuery `$()` and `elm.html()` API without any known problems.

What about legacy browsers like Internet Explorer?

DOMPurify does nothing at all. It simply returns exactly the string that you fed it. DOMPurify exposes a property called `isSupported`, which tells you whether it will be able to do its job, so you can come up with your own backup plan.

What about DOMPurify and Trusted Types?

In version 1.0.9, support for Trusted Types API was added to DOMPurify. In version 2.0.0, a config flag was added to control DOMPurify's behavior regarding this.

When `DOMPurify.sanitize` is used in an environment where the Trusted Types API is available and `RETURN_TRUSTED_TYPE` is set to `true`, i tries to return a `TrustedHTML` value instead of a string (the behavior for `RETURN_DOM` and `RETURN_DOM_FRAGMENT` config options does not change).

Note that in order to create a policy in `trustedTypes` using DOMPurify, `RETURN_TRUSTED_TYPE: false` is required, as `createHTML` expects a normal string, not `TrustedHTML`. The example below shows this.

```
window.trustedTypes!.createPolicy('default', {
  createHTML: (to_escape) =>
    DOMPurify.sanitize(to_escape, { RETURN_TRUSTED_TYPE: false }),
});
```

Can I configure DOMPurify?

Yes. The included default configuration values are pretty good already - but you can of course override them. Check out the `/demos` folder to see a bunch of examples on how you can customize DOMPurify.

General settings

```
// strip {{ ... }}, ${ ... } and <% ... %> to make output safe for template systems
// be careful please, this mode is not recommended for production usage.
// allowing template parsing in user-controlled HTML is not advised at all.
// only use this mode if there is really no alternative.
const clean = DOMPurify.sanitize(dirty, {SAFE_FOR_TEMPLATES: true});


// change how e.g. comments containing risky HTML characters are treated.
// be very careful, this setting should only be set to `false` if you really only handle
// HTML and nothing else, no SVG, MathML or the like.
// Otherwise, changing from `true` to `false` will lead to XSS in this or some other way.
const clean = DOMPurify.sanitize(dirty, {SAFE_FOR_XML: false});
```

Control our allow-lists and block-lists

```
// allow only <b> elements, very strict
const clean = DOMPurify.sanitize(dirty, {ALLOWED_TAGS: ['b']});

// allow only <b> and <q> with style attributes
const clean = DOMPurify.sanitize(dirty, {ALLOWED_TAGS: ['b', 'q'], ALLOWED_ATTR: ['style']});

// allow all safe HTML elements but neither SVG nor MathML
// note that the USE_PROFILES setting will override the ALLOWED_TAGS setting
// so don't use them together
const clean = DOMPurify.sanitize(dirty, {USE_PROFILES: {html: true}});

// allow all safe SVG elements and SVG Filters, no HTML or MathML
const clean = DOMPurify.sanitize(dirty, {USE_PROFILES: {svg: true, svgFilters: true}});

// allow all safe MathML elements and SVG, but no SVG Filters
const clean = DOMPurify.sanitize(dirty, {USE_PROFILES: {mathMl: true, svg: true}});

// change the default namespace from HTML to something different
const clean = DOMPurify.sanitize(dirty, {NAMESPACE: 'http://www.w3.org/2000/svg'});

// leave all safe HTML as it is and add <style> elements to block-list
const clean = DOMPurify.sanitize(dirty, {FORBID_TAGS: ['style']});

// leave all safe HTML as it is and add style attributes to block-list
const clean = DOMPurify.sanitize(dirty, {FORBID_ATTR: ['style']});

// extend the existing array of allowed tags and add <my-tag> to allow-list
const clean = DOMPurify.sanitize(dirty, {ADD_TAGS: ['my-tag']});

// extend the existing array of allowed attributes and add my-attr to allow-list
const clean = DOMPurify.sanitize(dirty, {ADD_ATTR: ['my-attr']});

// use functions to control which additional tags and attributes are allowed
const allowlist = {
    'one': ['attribute-one'],
    'two': ['attribute-two']
};
const clean = DOMPurify.sanitize(
    '<one attribute-one="1" attribute-two="2"></one><two attribute-one="1" attribute-two="2"></two>',
    {
        ADD_TAGS: (tagName) => {
            return Object.keys(allowlist).includes(tagName);
        },
        ADD_ATTR: (attributeName, tagName) => {

            return allowlist[tagName]?.includes(attributeName) || false;
        }
    }
); // <one attribute-one="1"></one><two attribute-two="2"></two>

// prohibit ARIA attributes, leave other safe HTML as is (default is true)
const clean = DOMPurify.sanitize(dirty, {ALLOW_ARIA_ATTR: false});

// prohibit HTML5 data attributes, leave other safe HTML as is (default is true)
const clean = DOMPurify.sanitize(dirty, {ALLOW_DATA_ATTR: false});
```

Control behavior relating to Custom Elements

```
// DOMPurify allows to define rules for Custom Elements. When using the CUSTOM_ELEMENT_HANDLING
// literal, it is possible to define exactly what elements you wish to allow (by default, none are allowed).
//
// The same goes for their attributes. By default, the built-in or configured allow.list is used.
//
// You can use a RegExp literal to specify what is allowed or a predicate, examples for both can be seen below.
// When using a predicate function for attributeNameCheck, it can optionally receive the tagName as a second paramete
// for more granular control over which attributes are allowed for specific elements.
// The default values are very restrictive to prevent accidental XSS bypasses. Handle with great care!

const clean = DOMPurify.sanitize(
    '<foo-bar baz="foobar" forbidden="true"></foo-bar><div is="foo-baz"></div>',
```

```
    {
        CUSTOM_ELEMENT_HANDLING: {
            tagNameCheck: null, // no custom elements are allowed
            attributeNameCheck: null, // default / standard attribute allow-list is used
            allowCustomizedBuiltInElements: false, // no customized built-ins allowed
        },
    }
); // <div is=""></div>

const clean = DOMPurify.sanitize(
    '<foo-bar baz="foobar" forbidden="true"></foo-bar><div is="foo-baz"></div>',
    {
        CUSTOM_ELEMENT_HANDLING: {
            tagNameCheck: /^foo-/, // allow all tags starting with "foo-"
            attributeNameCheck: /baz/, // allow all attributes containing "baz"
            allowCustomizedBuiltInElements: true, // customized built-ins are allowed
        },
    }
); // <foo-bar baz="foobar"></foo-bar><div is="foo-baz"></div>

const clean = DOMPurify.sanitize(
    '<foo-bar baz="foobar" forbidden="true"></foo-bar><div is="foo-baz"></div>',
    {
        CUSTOM_ELEMENT_HANDLING: {
            tagNameCheck: (tagName) => tagName.match(/^foo-/), // allow all tags starting with "foo-"
            attributeNameCheck: (attr) => attr.match(/baz/), // allow all containing "baz"
            allowCustomizedBuiltInElements: true, // allow customized built-ins
        },
    }
); // <foo-bar baz="foobar"></foo-bar><div is="foo-baz"></div>

// Example with attributeNameCheck receiving tagName as a second parameter
const clean = DOMPurify.sanitize(
    '<element-one attribute-one="1" attribute-two="2"></element-one><element-two attribute-one="1" attribute-two="2">
    {
        CUSTOM_ELEMENT_HANDLING: {
            tagNameCheck: (tagName) => tagName.match(/^element-(one|two)$/),
            attributeNameCheck: (attr, tagName) => {
                if (tagName === 'element-one') {
                    return ['attribute-one'].includes(attr);
                } else if (tagName === 'element-two') {
                    return ['attribute-two'].includes(attr);
                } else {
                    return false;
                }
            },
            allowCustomizedBuiltInElements: false,
        },
    }
); // <element-one attribute-one="1"></element-one><element-two attribute-two="2"></element-two>
```

Control behavior relating to URI values

```
// extend the existing array of elements that can use Data URIs
const clean = DOMPurify.sanitize(dirty, {ADD_DATA_URI_TAGS: ['a', 'area']});

// extend the existing array of elements that are safe for URI-like values (be careful, XSS risk)
const clean = DOMPurify.sanitize(dirty, {ADD_URI_SAFE_ATTR: ['my-attr']});
```

Control permitted attribute values

```
// allow external protocol handlers in URL attributes (default is false, be careful, XSS risk)
// by default only http, https, ftp, ftps, tel, mailto, callto, sms, cid and xmpp are allowed.
const clean = DOMPurify.sanitize(dirty, {ALLOW_UNKNOWN_PROTOCOLS: true});

// allow specific protocols handlers in URL attributes via regex (default is false, be careful, XSS risk)
// by default only (protocol-)relative URLs, http, https, ftp, ftps, tel, mailto, callto, sms, cid, xmpp and matrix a
```

```
// Default RegExp: /^(?:(?:(?:f|ht)tps?|mailto|tel|callto|sms|cid|xmpp):|[^a-z]|[a-z+.\-]+(?:[^a-z+.\-:]|$))/i;
const clean = DOMPurify.sanitize(dirty, {ALLOWED_URI_REGEXP: /^(?:(?:(?:f|ht)tps?|mailto|tel|callto|sms|cid|xmpp|matr
```

Influence the return-type

```
// return a DOM HTMLBodyElement instead of an HTML string (default is false)
const clean = DOMPurify.sanitize(dirty, {RETURN_DOM: true});

// return a DOM DocumentFragment instead of an HTML string (default is false)
const clean = DOMPurify.sanitize(dirty, {RETURN_DOM_FRAGMENT: true});

// use the RETURN_TRUSTED_TYPE flag to turn on Trusted Types support if available
const clean = DOMPurify.sanitize(dirty, {RETURN_TRUSTED_TYPE: true}); // will return a TrustedHTML object instead of

// use a provided Trusted Types policy
const clean = DOMPurify.sanitize(dirty, {
    // supplied policy must define createHTML and createScriptURL
    TRUSTED_TYPES_POLICY: trustedTypes.createPolicy({
        createHTML(s) { return s},
        createScriptURL(s) { return s},
    })
});
```

Influence how we sanitize

```
// return entire document including <html> tags (default is false)
const clean = DOMPurify.sanitize(dirty, {WHOLE_DOCUMENT: true});

// disable DOM Clobbering protection on output (default is true, handle with care, minor XSS risks here)
const clean = DOMPurify.sanitize(dirty, {SANITIZE_DOM: false});

// enforce strict DOM Clobbering protection via namespace isolation (default is false)
// when enabled, isolates the namespace of named properties (i.e., `id` and `name` attributes)
// from JS variables by prefixing them with the string `user-content-`
const clean = DOMPurify.sanitize(dirty, {SANITIZE_NAMED_PROPS: true});

// keep an element's content when the element is removed (default is true)
const clean = DOMPurify.sanitize(dirty, {KEEP_CONTENT: false});

// glue elements like style, script or others to document.body and prevent unintuitive browser behavior in several ec
const clean = DOMPurify.sanitize(dirty, {FORCE_BODY: true});

// remove all <a> elements under <p> elements that are removed
const clean = DOMPurify.sanitize(dirty, {FORBID_CONTENTS: ['a'], FORBID_TAGS: ['p']});

// extend the default FORBID_CONTENTS list to also remove <a> elements under <p> elements
const clean = DOMPurify.sanitize(dirty, {ADD_FORBID_CONTENTS: ['a'], FORBID_TAGS: ['p']});

// change the parser type so sanitized data is treated as XML and not as HTML, which is the default
const clean = DOMPurify.sanitize(dirty, {PARSER_MEDIA_TYPE: 'application/xhtml+xml'});
```

Influence where we sanitize

```
// use the IN_PLACE mode to sanitize a node "in place", which is much faster depending on how you use DOMPurify
const dirty = document.createElement('a');
dirty.setAttribute('href', 'javascript:alert(1)');

const clean = DOMPurify.sanitize(dirty, {IN_PLACE: true}); // see https://github.com/cure53/DOMPurify/issues/288 for
```

There is even more examples here, showing how you can run, customize and configure DOMPurify to fit your needs.

Persistent Configuration

Instead of repeatedly passing the same configuration to `DOMPurify.sanitize`, you can use the `DOMPurify.setConfig` method. Your configuration will persist until your next call to `DOMPurify.setConfig`, or until you invoke `DOMPurify.clearConfig` to reset it. Remember that there is only one active configuration, which means once it is set, all extra configuration parameters passed to `DOMPurify.sanitize` are ignored.

Hooks

DOMPurify allows you to augment its functionality by attaching one or more functions with the `DOMPurify.addHook` method to one of the following hooks:

- `beforeSanitizeElements`
- `uponSanitizeElement` (No 's' - called for every element)
- `afterSanitizeElements`
- `beforeSanitizeAttributes`
- `uponSanitizeAttribute`
- `afterSanitizeAttributes`
- `beforeSanitizeShadowDOM`
- `uponSanitizeShadowNode`
- `afterSanitizeShadowDOM`

It passes the currently processed DOM node, when needed a literal with verified node and attribute data and the DOMPurify configuration to the callback. Check out the MentalJS hook demo to see how the API can be used nicely.

Example:

```
DOMPurify.addHook(
  'uponSanitizeAttribute',
  function (currentNode, hookEvent, config) {
    // Do something with the current node
    // You can also mutate hookEvent for current node (i.e. set hookEvent.forceKeepAttr = true)
    // For other than 'uponSanitizeAttribute' hook types hookEvent equals to null
  }
);
```

Removed Configuration

Option	Since	Note
SAFE_FOR_JQUERY	2.1.0	No replacement required.

Continuous Integration

We are currently using Github Actions in combination with BrowserStack. This gives us the possibility to confirm for each and every commit that all is going according to plan in all supported browsers. Check out the build logs here: https://github.com/cure53/DOMPurify/actions

You can further run local tests by executing `npm run test`.

All relevant commits will be signed with the key `0x24BB6BF4` for additional security (since 8th of April 2016).

Development and contributing

Installation (`npm i`)

We support `npm` officially. GitHub Actions workflow is configured to install dependencies using `npm`. When using deprecated version of `npm` we can not fully ensure the versions of installed dependencies which might lead to unanticipated problems.

Scripts

We use ESLint as a pre-commit hook to ensure code consistency. Moreover, to ease formatting we use prettier while building the `/dist` assets happens through `rollup`.

These are our npm scripts:

- `npm run dev` to start building while watching sources for changes
- `npm run test` to run our test suite via jsdom and karma
  - `test:jsdom` to only run tests through jsdom
  - `test:karma` to only run tests through karma
- `npm run lint` to lint the sources using ESLint (via xo)
- `npm run format` to format our sources using prettier to ease to pass ESLint
- `npm run build` to build our distribution assets minified and unminified as a UMD module
  - `npm run build:umd` to only build an unminified UMD module
  - `npm run build:umd:min` to only build a minified UMD module

Note: all run scripts triggered via `npm run <script>`.

There are more npm scripts but they are mainly to integrate with CI or are meant to be "private" for instance to amend build distribution files with every commit.

Security Mailing List

We maintain a mailing list that notifies whenever a security-critical release of DOMPurify was published. This means, if someone found a bypass and we fixed it with a release (which always happens when a bypass was found) a mail will go out to that list. This usually happens within minutes or few hours after learning about a bypass. The list can be subscribed to here:

https://lists.ruhr-uni-bochum.de/mailman/listinfo/dompurify-security

Feature releases will not be announced to this list.

Who contributed?

Many people helped and help DOMPurify become what it is and need to be acknowledged here!

Cybozu 💛💸, hata6502 💸, intra-mart-dh 💸, nelstrom ❤️, hash_kitten ❤️, kevin_mizu ❤️, icesfont ❤️, reduckted ❤️, dcramer 💸, JGraph 💸, baekilda 💸, Healthchecks 💸, Sentry 💸, jarrodldavis 💸, CynegeticIO, ssi02014 ❤️, GrantGryczan, Lowdefy, granlem, oreoshake, tdeekens ❤️, peernohell ❤️, is2ei, SoheilKhodayari, franktopel, NateScarlet, neilj, fhemberger, Joris-van-der-Wel, ydaniv, terjanq, filedescriptor, ConradIrwin, gibson042, choumx, 0xSobky, styfle, koto, tlau88, strugee, oparoz, mathiasbynens, edg2s, dnkolegov, dhardtke, wirehead, thorn0, styu, mozfreddyb, mikesamuel, jorangreef, jimmyhchan, jameydeorio, jameskraus, hyderali, hansottowirtz, hackvertor, freddyb, flavorjones, djfarrelly, devd, camerondunford, buu700, buildog, alabiaga, Vector919, Robbert, GreLI, FuzzySockets, ArtemBernatskyy @garethheyes, @shafigullin, @mmrupp, @irsdl,ShikariSenpai, ansjdnakjdnajkd, @asutherland, @mathias, @cgvwzq, @robbertatwork, @giutro, @CmdEngineer_, @avr4mit, davecardwell and especially @securitymb ❤️ & @masatokinugawa ❤️

Testing powered by

Releases 133

DOMPurify 3.3.3 Latest
3 weeks ago

+ 132 releases

Sponsor this project

cure53 Cure53

Sponsor

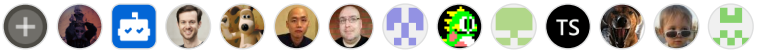
Learn more about GitHub Sponsors

Packages

No packages published

Used by 603k

Contributors 116



+ 102 contributors

Languages

JavaScript 52.3% **TypeScript** 44.9% **HTML** 2.7% **Shell** 0.1%