Policies

> *Stability: 1 - Experimental*

Node.js contains experimental support for creating policies on loading code.

Policies are a security feature intended to allow guarantees about what code Node.js is able to load. The use of policies assumes safe practices for the policy files such as ensuring that policy files cannot be overwritten by the Node.js application by using file permissions.

A best practice would be to ensure that the policy manifest is read-only for the running Node.js application and that the file cannot be changed by the running Node.js application in any way. A typical setup would be to create the policy file as a different user id than the one running Node.js and granting read permissions to the user id running Node.js.

Enabling

The `--experimental-policy` flag can be used to enable features for policies when loading modules.

Once this has been set, all modules must conform to a policy manifest file passed to the flag:

```
node --experimental-policy=policy.json app.js
```

The policy manifest will be used to enforce constraints on code loaded by Node.js.

To mitigate tampering with policy files on disk, an integrity for the policy file itself may be provided via `--policy-integrity`. This allows running `node` and asserting the policy file contents even if the file is changed on disk.

```
node --experimental-policy=policy.json --policy-integrity="sha384-SggXRQHwCG8g+DktYYzxkXRIkTiEYWBHqev0xnpCxYlqMBufKZHAHQM3/boDaI/0" app.js
```

Features

Error behavior

When a policy check fails, Node.js by default will throw an error. It is possible to change the error behavior to one of a few possibilities by defining an "onerror" field in a policy manifest. The following values are available to change the behavior:

- `"exit"` : will exit the process immediately. No cleanup code will be allowed to run.
- `"log"` : will log the error at the site of the failure.
- `"throw"` : will throw a JS error at the site of the failure. This is the default.

```
{
  "onerror": "log",
  "resources": {
    "./app/checked.js": {
      "integrity": "sha384-SggXRQHwCG8g+DktYYzxkXRIkTiEYWBHqev0xnpCxYlqMBufKZHAHQM3/boDaI/0"
    }
  }
}
```

Integrity checks

Policy files must use integrity checks with Subresource Integrity strings compatible with the browser integrity attribute associated with absolute URLs.

When using `require()` or `import` all resources involved in loading are checked for integrity if a policy manifest has been specified. If a resource does not match the integrity listed in the manifest, an error will be thrown.

An example policy file that would allow loading a file `checked.js`:

```
{
  "resources": {
    "./app/checked.js": {
      "integrity": "sha384-SggXRQHwCG8g+DktYYzxkXRIkTiEYWBHqev0xnpCxYlqMBufKZHAHQM3/boDaI/0"
    }
  }
}
```

Each resource listed in the policy manifest can be of one the following formats to determine its location:

1. A relative-URL string to a resource from the manifest such as `./resource.js`, `../resource.js`, or `/resource.js`.
2. A complete URL string to a resource such as `file:///resource.js`.

When loading resources the entire URL must match including search parameters and hash fragment. `./a.js?b` will not be used when attempting to load `./a.js` and vice versa.

To generate integrity strings, a script such as `node -e 'process.stdout.write("sha256-");process.stdin.pipe(crypto.createHash("sha256").setEncoding("base64")).pipe(process.stdout)' < FILE` can be used.

Integrity can be specified as the boolean value `true` to accept any body for the resource which can be useful for local development. It is not recommended in production since it would allow unexpected alteration of resources to be considered valid.

Dependency redirection

An application may need to ship patched versions of modules or to prevent modules from allowing all modules access to all other modules. Redirection can be used by intercepting attempts to load the modules wishing to be replaced.

```
{
  "resources": {
    "./app/checked.js": {
      "dependencies": {
        "fs": true,
        "os": "./app/node_modules/alt-os",
        "http": { "import": true }
      }
    }
  }
}
```

The dependencies are keyed by the requested specifier string and have values of either `true`, `null`, a string pointing to a module to be resolved, or a conditions object.

The specifier string does not perform any searching and must match exactly what is provided to the `require()` or `import` except for a canonicalization step. Therefore, multiple specifiers may be needed in the policy if it uses multiple different strings to point to the same module (such as excluding the extension).

Specifier strings are canonicalized but not resolved prior to be used for matching in order to have some compatibility with import maps, for example if a resource `file:///C:/app/server.js` was given the following redirection from a policy located at `file:///C:/app/policy.json` :

```
{
  "resources": {
    "file:///C:/app/utils.js": {
      "dependencies": {
        "./utils.js": "./utils-v2.js"
      }
    }
  }
}
```

Any specifier used to load `file:///C:/app/utils.js` would then be intercepted and redirected to `file:///C:/app/utils-v2.js` instead regardless of using an absolute or relative specifier. However, if a specifier that is not an absolute or relative URL string is used, it would not be intercepted. So, if an import such as `import('#utils')` was used, it would not be intercepted.

If the value of the redirection is `true` , a "dependencies" field at the top of the policy file will be used. If that field at the top of the policy file is `true` the default node searching algorithms are used to find the module.

If the value of the redirection is a string, it is resolved relative to the manifest and then immediately used without searching.

Any specifier string for which resolution is attempted and that is not listed in the dependencies results in an error according to the policy.

Redirection does not prevent access to APIs through means such as direct access to `require.cache` or through `module.constructor` which allow access to loading modules. Policy redirection only affects specifiers to `require()` and `import` . Other means, such as to prevent undesired access to APIs through variables, are necessary to lock down that path of loading modules.

A boolean value of `true` for the dependencies map can be specified to allow a module to load any specifier without redirection. This can be useful for local development and may have some valid usage in production, but should be used only with care after auditing a module to ensure its behavior is valid.

Similar to `"exports"` in `package.json` , dependencies can also be specified to be objects containing conditions which branch how dependencies are loaded. In the preceding example, `"http"` is allowed when the `"import"` condition is part of loading it.

A value of `null` for the resolved value causes the resolution to fail. This can be used to ensure some kinds of dynamic access are explicitly prevented.

Unknown values for the resolved module location cause failures but are not guaranteed to be forward compatible.

Example: Patched dependency

Redirected dependencies can provide attenuated or modified functionality as fits the application. For example, log data about timing of function durations by wrapping the original:

```
const original = require('fn');
module.exports = function fn(...args) {
  console.time();
  try {
    return new.target ?
      Reflect.construct(original, args) :
      Reflect.apply(original, this, args);
  } finally {
    console.timeEnd();
  }
};
```

Scopes

Use the `"scopes"` field of a manifest to set configuration for many resources at once. The `"scopes"` field works by matching resources by their segments. If a scope or resource includes `"cascade": true`, unknown specifiers will be searched for in their containing scope. The containing scope for cascading is found by recursively reducing the resource URL by removing segments for special schemes, keeping trailing `"/"` suffixes, and removing the query and hash fragment. This leads to the eventual reduction of the URL to its origin. If the URL is non-special the scope will be located by the URL's origin. If no scope is found for the origin or in the case of opaque origins, a protocol string can be used as a scope. If no scope is found for the URL's protocol, a final empty string `""` scope will be used.

Note, `blob:` URLs adopt their origin from the path they contain, and so a scope of `"blob:https://nodejs.org"` will have no effect since no URL can have an origin of `blob:https://nodejs.org`; URLs starting with `blob:https://nodejs.org/` will use `https://nodejs.org` for its origin and thus `https:` for its protocol scope. For opaque origin `blob:` URLs they will have `blob:` for their protocol scope since they do not adopt origins.

Example

```
{
  "scopes": {
    "file:///C:/app/": {},
    "file:": {},
    "": {}
  }
}
```

Given a file located at `file:///C:/app/bin/main.js`, the following scopes would be checked in order:

1. `"file:///C:/app/bin/"`

This determines the policy for all file based resources within `"file:///C:/app/bin/"`. This is not in the `"scopes"` field of the policy and would be skipped. Adding this scope to the policy would cause it to be used prior to the `"file:///C:/app/"` scope.

2. `"file:///C:/app/"`

This determines the policy for all file based resources within `"file:///C:/app/"`. This is in the `"scopes"` field of the policy and it would determine the policy for the resource at `file:///C:/app/bin/main.js`. If the scope has `"cascade": true`, any unsatisfied queries about the resource would delegate to the next relevant scope for `file:///C:/app/bin/main.js`, `"file:"`.

3. `"file:///C:/"`

This determines the policy for all file based resources within `"file:///C:/"`. This is not in the `"scopes"` field of the policy and would be skipped. It would not be used for `file:///C:/app/bin/main.js` unless `"file:///"` is set to cascade or is not in the `"scopes"` of the policy.

4. `"file:///"`

This determines the policy for all file based resources on the `localhost`. This is not in the `"scopes"` field of the policy and would be skipped. It would not be used for `file:///C:/app/bin/main.js` unless `"file:///"` is set to cascade or is not in the `"scopes"` of the policy.

5. `"file:"`

This determines the policy for all file based resources. It would not be used for `file:///C:/app/bin/main.js` unless `"file:///"` is set to cascade or is not in the `"scopes"` of the policy.

6. `""`

This determines the policy for all resources. It would not be used for `file:///C:/app/bin/main.js` unless `"file:"` is set to cascade.

Integrity using scopes

Setting an integrity to `true` on a scope will set the integrity for any resource not found in the manifest to `true`.

Setting an integrity to `null` on a scope will set the integrity for any resource not found in the manifest to fail matching.

Not including an integrity is the same as setting the integrity to `null`.

`"cascade"` for integrity checks will be ignored if `"integrity"` is explicitly set.

The following example allows loading any file:

```json
{
  "scopes": {
    "file:": {
      "integrity": true
    }
  }
}
```

Dependency redirection using scopes

The following example, would allow access to `fs` for all resources within `./app/`:

```json
{
  "resources": {
    "./app/checked.js": {
      "cascade": true,
      "integrity": true
    }
  },
  "scopes": {
    "./app/": {
      "dependencies": {
        "fs": true
      }
    }
  }
}
```

The following example, would allow access to `fs` for all `data:` resources:

```
{
  "resources": {
    "data:text/javascript,import('fs');": {
      "cascade": true,
      "integrity": true
    }
  },
  "scopes": {
    "data:": {
      "dependencies": {
        "fs": true
      }
    }
  }
}
```

Example: import maps emulation

Given an import map:

```
{
  "imports": {
    "react": "./app/node_modules/react/index.js"
  },
  "scopes": {
    "./ssr/": {
      "react": "./app/node_modules/server-side-react/index.js"
    }
  }
}
```

```
{
  "dependencies": true,
  "scopes": {
    "": {
      "cascade": true,
      "dependencies": {
        "react": "./app/node_modules/react/index.js"
      }
    },
    "./ssr/": {
      "cascade": true,
      "dependencies": {
        "react": "./app/node_modules/server-side-react/index.js"
      }
    }
  }
}
```

Import maps assume you can get any resource by default. This means `"dependencies"` at the top level of the policy should be set to `true`. Policies require this to be opt-in since it enables all resources of the application cross linkage which doesn't make sense for many scenarios. They also assume any given scope has access to any scope above its allowed dependencies; all scopes emulating import maps must set `"cascade": true`.

Import maps only have a single top level scope for their "imports". So for emulating `"imports"` use the `""` scope. For emulating `"scopes"` use the `"scopes"` in a similar manner to how `"scopes"` works in import maps.

Caveats: Policies do not use string matching for various finding of scope. They do URL traversals. This means things like `blob:` and `data:` URLs might not be entirely interoperable between the two systems. For example import maps can partially match a `data:` or `blob:` URL by partitioning the URL on a `/` character, policies intentionally cannot. For `blob:` URLs import map scopes do not adopt the origin of the `blob:` URL.

Additionally, import maps only work on `import` so it may be desirable to add a `"import"` condition to all dependency mappings.