SES Documentation

Release 0.4.0

Agoric

Feb 21, 2020

Contents:

1 Getting Started **1**

1.1 Installing SES 1

1.2 Try it out 1

1.3 Bundlers 2

1.4 Building from scratch 2

1.5 Webworkers 2

2 SES API **3**

3 Draft Spec for Standalone SES **9**

3.1 Omissions and Simplifications 9

3.2 Additions 10

3.3 Work in Progress 11

3.4 Stage Separated SES 11

CHAPTER 1

Getting Started

SES is a JavaScript package that allows you to run third-party code safely. It runs in Node.js and in the browser.

1.1 Installing SES

In Node.js:

```
npm install ses
```

In the browser:

```
<script src="https://unpkg.com/ses"></script>
```

1.2 Try it out

In a Node.js repl, after running `npm install ses`:

```
const SES = require('ses');
const s = SES.makeSESRootRealm({consoleMode: 'allow', errorStackMode: 'allow'});
// NOTE: errorStackMode enables confinement breach, do not leave on in production
s.evaluate('1+2'); // returns 3
s.evaluate('1+a', {a: 3}); // returns 4
function double(a) {
  return a*2;
}
const doubler = s.evaluate(`(${double})`);
doubler(3); // returns 6
```

In the browser after loading SES:

```
const s = SES.makeSESRootRealm(...);
s.evaluate(...);
```

1.3 Bundlers

SES works with the main bundlers such as Webpack, Browserify, Rollup, and Parcel. Simply install SES using npm and `require` or `import` it.

1.4 Building from scratch

Clone the SES repo and run `npm run-script build`, which will create a variety of files under `dist/`. `ses.cjs.js` is the CommonJS version of SES, `ses.esm.js` is the ES6 module version, and `ses.umd.js` is the UMD version intended for the browser.

1.5 Webworkers

Note that the Realm shim currently requires either the Node.js `vm` module, or a browser's `<iframe>` element (it does `document.createElement('iframe')`), so it won't work in a DOM-less `WebWorker`, `SharedWorker`, or `ServiceWorker`. If/when the Realms proposal becomes a standard part of Javascript, these environments ought to have a native `Realm` object available, and SES should work in all of them.

CHAPTER 2

SES API

The main entry point to SES is `const s = SES.makeSESRootRealm(options)`. This creates a new SES "root" Realm, in which all primordials are frozen, all sources of non-determinism are disabled, and all means of escape are blocked off.

The main utility of this new `s` object (which we might call the "realm controller") is the `s.evaluate()` method. This currently takes two arguments: `s.evaluate(code, endowments)`. `code` is a string that defines a javascript expression, and `endowments` is an object whose properties will be made available in the global lexical scope of that expression.

The code is evaluated in a new global object, so assigning anything to it is not generally useful: `s.evaluate('let a = 3')` is legal, but useless, since a subsequent `s.evaluate('a+1')` won't see the same `a`. On the other hand, `s.evaluate('let a = 3; a+1')` will yield 4, because the same `a` is in scope throughout both parts of the same evaluation.

The code is evaluated as an expression, so to get a function object back out, you must wrap it in parenthesis, or evaluate an arrow function:

```
let d1 = s.evaluate('(function double(a) { return a+a; })');
d1(1); // 2
let d2 = s.evaluate('(function(a) { return a+a; })');
d2(1); // 2, function is anonymous
let d3 = s.evaluate('a => a+a');
d3(1); // 2
```

`s.evaluate()` takes a string, but usually you write javascript in files with an editor, so there are some tricks to make it comfortable to edit the code you're going to submit to be evaluated. Most (but not all!) javascript engines will remember the source code of the functions you define, and stringifying the function object (e.g. with the backtick "template literal" operator) will retrieve its source code. We can use this to write a function as usual, then pass its stringified form into evaluate:

```
// we define inner but never invoke it in the outer realm, it exists only to be
// stringified
function inner(a) {
  return a+a;
```

(continues on next page)

(continued from previous page)

```
}

let d4 = s.evaluate(`(${inner})`);
```

caveats: `inner` must not reference anything outside its curly brackets, as that won't be available during its evaluation (i.e. it cannot "close over" variables from outside, as cool as that'd be). On the other hand, you can use "endowments" to provide names that this will look up:

```
// note: 'b' is not defined anywhere in the outer realm. Within the definition of
↪'inner', it is
// an unbound/free variable. Many linters and editors will notice this and complain.
function inner(a) {
  return a+b;
}

let d5 = s.evaluate(`(${inner})`, { b: 2 });
d5(1); // 3
```

You can evaluate an entire file, with multiple function definitions that reference each other, but only the final expression will be returned.

You may want to use `rollup` or some bundling tool with an API to turn multiple files into a single string. These source files can `require()` (CommonJS-mode) or `import` (ES6-module mode) each other, but the final string must not have any require/import statements, because `s.evaluate()` only evaluates a string and does not do module lookup (but see `s.makeRequire()` for a helper function that can provide something similar to `require()`).

Endowments are provided in the global lexical scope of the code being evaluated, where they can be referenced with free variables. The "global lexical scope" is not the same thing as the "global object". Using `this` at the top level of the evaluated code references a sort of global object (which is frozen), which has properties like `Number` and `Array` but not the endowments.

TODO: an example that shows the differences between `this.a=3` (which fails because the sort-of-global object is frozen), `let a = 3` (which modifies the other kind of global object, which is not frozen, but which goes out of scope at the end of evaluation), `let a = 3` with `endowments=x={}` (which I think shadows the endowment in the ephemeral global object and does *not* set `x.a=3` in the outer realm), and `a+1` (which probably looks at the ephemeral global object first, then falls back to the endowment)

Another common way to pass endowments is as arguments to a generated function.

```
function makeAdder(b) {
  return a => a+b;
}

let d6 = s.evaluate(`(${makeAdder})`)(4);
d6(1); // 5
```

Both approaches let the generated function close over the endowment, but using the `endowments` argument makes them available globally everywhere inside the evaluated code, whereas passing them as an argument makes them only available to the function that the code yields, which might enable finer-grained POLA.

The most common use for endowments (of either sort) is to safely allow in-Realm code access facilities from outside the realm. For example, the Realm's `consoleMode: 'allow'` feature is implemented with something like:

```
console.log('this is the real console object');
function makeConsole() {
  return {
    log(...args) {consoleEndowment.log(...args);}
```

(continues on next page)

(continued from previous page)

```
  }
}

const newConsole = s.evaluate(`(${makeConsole})()`, {consoleEndowment: console});
s.evaluate('console.log(4)', { console: newConsole });
```

Wrapping endowments like this is critical for security, because the simple approach would reveal an outer-realm object to the confined code, which it could use to escape confinement by modifying the parent Realm's primordials like the `toString()` method on `Objects`:

```
function evil() {
  const outerObjectPrototype = consoleEndowment.log.__proto__.__proto__;
  outerObjectPrototype.toString = obj => 'haha';
}

s.evaluate(`(${evil})()`, { consoleEndowment: console });
({}).toString(); // prints 'haha'
```

The key is that we evaluate trusted code to generate the safe endowment, and only pass the safe endowment to the untrusted code. Every object in the system should be examined to identify which realm it is coming from (outer or inner), and never ever reveal outer-realm objects to untrusted code. Even passing a collection of safe inner-realm objects to untrusted code enables a confinement breach:

```
const safeConsole = ...;
const safeAdder = ...;
s.evaluate(`(${untrustedCode})()`, { collection: { safeConsole, safeAddres } });
// the 'collection' object is outer-realm, and enables a breach
```

The safest approach is to build a bunch of outer-realm helper functions, bundle your entire application into a single string that defines a bootstrap function which accepts those helpers as an argument, then invoke the bootstrap function. Other patterns are in development, specifically ones that use `require` or `import` and a manifest of authorities to implement safe module loading.

The `SES.makeSESRootRealm()` call takes an options bundle. This affects what features of the realm are enabled or disabled. The default is to provide full confinement, which means that calling `s.evaluate(code)` (with no endowments, and discarding the return value) will never affect the outer realm, no matter what 'code' might contain. (This is clearly useless, like asking whether a tree falling in the woods makes a sound if there's nobody around to hear it). The default is also fully deterministic: no aspects of the platform will affect the execution of the code.

The options bundle can accept some keys which weaken these properties in exchange for other useful behavior.

- `SES.makeSESRootRealm({consoleMode: 'allow'})`: the default setting removes the `console` from the global scope, but setting this to `allow` brings it back. The in-realm `console` is not as fully-featured as the usual one that browsers or Node.js provides, but the most common methods are present.
- `errorStackMode: 'allow': To prevent confinement breaches, several platform-specific properties of Error objects are removed. Unfortunately this breaks the display of line numbers and file names, stack traces, and frequently the Error string itself. Exceptions that are not caught normally cause Node to exit with a stack trace: the SES default setting causes Node to print`undefined' and exit with no other explanation, which is particularly annoying. We currently recommend turning this on only temporarily while debugging an uncaught exception. Do not turn it on outside of debugging, because we believe it causes a confinement breach. Hopefully we'll find a way to fix this and enable sensible Error reporting without enabling a breach, at which point we'll change the default value.

- `mathRandomMode: 'allow'`: Since SES is supposed to be deterministic, `Math.random()` is a problem. By default it is disabled, and calling it throws an exception. When this mode is `allow`, Math.random is enabled. This introduces non-determinism, but if the platform's PRNG is sound, it should not enable the confined code to sense a covert channel, nor should it enable communication between otherwise isolated objects.
- `dateNowMode: 'allow'`: Allowing `Date.now()` to return the current time would both cause non-determinism *and* allow the reading of covert channels (enabling communication between isolated objects), and most applications don't need it, so this is disabled by default too. This affects both the static `Date.now()` call and the zero-argument `new Date()` constructor.
- `intlMode: 'allow'`: The platform normally supplies a default locale, for use in `Intl.DateTimeFormat` and `Intl.NumberFormat` calls that don't supply a specific locale to use. This platform locale introduces nondeterminism, so these must be disabled. The default setting is to delete the entire `Intl` object, but setting this to `allow` brings everything back. We may be able to bring back most of `Intl` by default, but platforms currently appear to supply the platform-default locale even to calls that supply a specific one, if the requested locale is not available (e.g. `Intl.NumberFormat('es')` will return the default locale's formatter function if it doesn't have a Spanish one available), which will take more work to tame.
- `rexexpMode: 'allow'`: several platforms provide non-standard properties on regexps that would enable communication between otherwise isolated objects. These are removed by default, but 'allow' would let them remain (enabling a confinement breach).

The realm controller object returned by `SES.makeSESRootRealm()` has basically three useful properties:

- `s.evaluate(code, endowments)`: described above
- `s.global`: this is the (frozen) global object inside the new Realm. Not actually very useful.
- `s.makeRequire(config)`

`makeRequire` is a helper function to construct an in-realm `require` object, so that the same code can be run outside of SES (where it uses Node.js's normal `require()` feature), or inside SES (where it uses the helper's version). It takes a `config` object that names the modules that can be imported, and describes what they should get when they do the import. The configuration syntax is intended to protect outer-realm objects against accidental exposure (which would enable a confinement breach).

```
const Nat = require('@agoric/nat');
const SES = require('ses');
const s = SES.makeSESRootRealm();
function mymod(x) {
  return x+x;
}

const req = s.makeRequire({'@agoric/nat': Nat, double: mymod})
function inner(y) {
  const double = require('double');
  const Nat = require('@agoric/nat');
  return double(Nat(y));
}
const inner = s.evaluate(`(${inner})`, {require: req});
inner(1); // 2
inner(-1); // Error since -1 is not a natural number
```

If the value of a config object element is a function, that function will be stringified, then evaluated inside the realm, then hardened, and the result is used as the module value (i.e. it is returned by any `require(modname)` done while that `require` endowment is in scope). This works for simple standalone functions that are designed to be stringified this way, like the `Nat` from `@agoric/nat` and the `mymod` function above. This won't work for functions that depend upon external references.

Note that `makeRequire` has an internal cache of modules, so any module that creates some mutable state (and makes it possible for callers to interact with it) may enable communication between otherwise isolated clients. A future version of makeRequire might help with the creation of "pure" modules that do not enable this unauthorized communication.

If the value of a configuration element is an object, `makeRequire` evaluates its `.attenuatorSource` property to get a function, then invokes that function with the rest of the configuration value. The result is hardened and used as the new module. This is intended to help build attenuating wrappers around external authorities.

We expect to change this API a lot. Eventually it should grow into a safe module loader, to enable some new variant of `s.evaluate` that looks more like a module load (with a corresponding manifest of acceptable authorities).

Javascript's `eval()` is a one-argument evaluator: it takes source code and evaluates it, producing a value or a function. The native `eval()` allows that source code to access the same lexical scope as the `eval` itself, which makes it unsafe for use on untrusted code.

Instead, SES offers a "safe two-argument evaluator". The "safe" property means that it doesn't give access to the scope of the invoker, making it safe to use with untrusted code. The second argument is a set of endowments to provide in place of that unsafe caller's scope.

From outside a Realm, you use `s.evaluate(code, endowments)` to invoke this safe two-argument evaluator. From *inside* a Realm, you instead of `SES.confine(code, endowments)`. This does the same thing, but acts "in-place" (from inside a realm).

If the code provided to `s.evaluate()` throws an error, the error object is mapped into an outer-Realm `Error` type before being exposed, to avoid accidents. The error object thrown by `SES.confine` is from the same realm as the `SES` object.

We also have a `confineExpr` variant. TODO: how exactly does this differ, when would you use it?

TODO: ambient `SES` within a realm is likely to go away, in favor of `require('ses')` and a special `s.makeRequire()` mode (just like `require('@agoric/harden')` is special). Not sure if that's good enough, or if the safe two-argument `eval` is important enough to expose in some easier way.

CHAPTER 3

Draft Spec for Standalone SES

In the Realms, Frozen Realms, Realms shim, and SES shim work, we've generally worked towards standardizing the APIs for dynamically *creating* a SES world from within a standard EcmaScript world. For IoT or blockchain purposes, the more relevant question is: What is the resulting standard SES world, independent of whether it was created from within a standard EcmaScript world, or whether it was implemented directly by a standalone SES engine that supports only SES?

(We use "blockchain" here as shorthand for the more general category of deterministically replicated SES computation, whether on a blockchain, permissioned BFT system, or whatever.)

3.1 Omissions and Simplifications

Since the primary purpose of the existing Realms/SES APIs and shims are to dynamically suppress parts of standard EcmaScript, a standalone SES engine would simply omit these elements, resulting in a simpler and smaller engine. Starting from standard EcmaScript, the simplification or omissions for the default configuration of SES are

- Omit all support for sloppy mode
- Aside from `BigInt`, omit everything else outside the EcmaScript 2018 spec.
- In particular, omit the `import()` and `import.meta` expressions.
- Omit annex B (except those our whitelist allows)
- In particular, omit the `RegExp` static properties that provide a global communications channel.
- Omit `Math.random()`
- **Omit ambient access to current date/time:**
    - `Date.now()` returns `NaN`
    - `new Date()` return equivalent of `new Date(NaN)`
- By default, omit `Intl`, the internationalization APIs
- If some of `Intl` is included, it must suppress ambient authority and non-determinism.

- **For all forms of function expressible by syntax (function, generator, async-function, async-generator)**
  - *func*`.[[Prototype]].constructor` is a function constructor that always throws. Because these function constructors always throw, we do not consider them to be evaluators.

We define the *shared globals* as all the standard shared global variable bindings defined by the above, i.e., without `Intl` by default, with `Realm` (see below), without `eval`, without `Function`, without anything outside the EcmaScript 2018 spec, and with `BigInt`. We define the *shared primordials* as all the objects transitively reachable from the shared globals. Note that no global objects or evaluators are reachable from the shared primordials.

3.2 Additions

Some IoT and blockchain configurations may omit all runtime evaluators. For standalone SES configurations that include runtime evaluators, they would appear as follows.

1. Include the portion of the Realm API for creating compartments, and for evaluating script code in a compartment with endowments:
   - `Realm.makeCompartment(options={})` -> aRealm instance representing a new compartment
   - `Realm.prototype.global` —> global object of compartment. This is a getter-only accessor.
   - `Realm.prototype.evaluateProgram(programSrcString, endowments={})` –> completion value
     - The own properties of the endowments which are legal variable names become the const variable bindings of the global lexical scope in which the program is evaluated. Unlike standard EcmaScript, there is no shared global lexical scope. Each global lexical scope comes *only* from the endowments.
   - `Realm.prototype.evaluateExpr(exprSrcString, endowments={})` –> value of expression
     - Given that `exprSrcString` parses as an expression, `js aRealm.evaluateExpr(exprSrcString, endowments)` is equivalent to `js aRealm.evaluateProgram(`(${exprSrcString});`, endowments)‘

   The additional element from the proposed Realm API is `Realm.makeRootRealm(options={})`. SES allows but does not require this static method. IoT and blockchain uses of SES generally have no need for multiple root realms. However, browser-based and Node-based use of SES will often be coupled with creating multiple confined root realms. On platforms that do not support `Realm.makeRootRealm`, the property must be absent so that SES code can feature-test for it.

2. Freeze all shared primordials. With the above omissions, there is no hidden state or ambient authority among the shared primordials, so transitive freezing means that the shared primordials are immutable and rom-able. Since no global objects or evaluators are reachable from the shared primordials. They can be placed in ROM without the bookkeeping needed for them to point at any objects not in ROM.

3. For each compartment, create a new global populated by:
   - The shared globals with their standard global property names.
   - An `eval` function and `Function` constructor that evaluates code in the scope of that global
     - Both this `eval` function and `Function` constructor inherit from the shared %FunctionPrototype% primordial.
     - Each of these `eval` functions is considered an initial eval function for purposes of determining whether a an expression in direct-eval syntax is indeed a direct-eval. (The direct-eval feature is impossible to shim and rarely needed anyway, and so is low priority. When omitted, the direct-eval syntax should also be statically rejected with an early error.)

    - `Function.prototype` is initialized to point at the same shared %FunctionPrototype% primordial.
  - All of these global properties are made non-configurable non-writable data properties. The new per-global objects (the eval function and Function constructor) are frozen. Since they have no hidden state, they are immutable and rom-able.
  - This new global object is *not* frozen. It remains extensible. However, the global's [[Prototype]] slot cannot be altered.

4. The host creates a start-compartment whose start-global is populated as above.
5. To that start-global object, the host adds global bindings to those host objects that provide initial access to the program's outside world, e.g., the I/O environment of the device.
6. The program's start scripts are then evaluated as program code in that start-compartment.

Each compartment scope has its own `Function`, which does evaluate. All compartment scopes share the same `Function.prototype` and therefore the same `Function.prototype.constructor` which is a function that only throws. Thus, in all compartment scopes,

```
Function !== Function.prototype.constructor
```

TBD: * What portion of the additions above are relevant to a standalone SES without runtime evaluators? * Should `eval` and `Function` actually be on a compartment's global object, or should we include them in the compartment's global lexical scope?

3.3 Work in Progress

We are still working towards specifying how SES supports modules. Indeed, this is the main topic of the SES-strategy sessions. Somehow, whether by import, require, or otherwise, a SES environment must provide access to the exports of the packages currently named '@agoric/nat' and '@agoric/harden', which will normally be bound to const variable named `Nat` and `harden`. We'll revisit all this is a separate document.

TBD: * `System` * error stacks * weak references * loader? * Should SES provide support for `require` and core CommonJS Modules? * Where should `Nat` and `harden` come from? * `SES` * `SES.confine`

3.4 Stage Separated SES

Full SES, as embedded into EcmaScript, supports running vetted customization code in a freezable realm prior to freezing it into a SES realm. Such vetted customization code runs in an environment like that described above except: * The shared primordials are not yet frozen * No host objects have been added to the global. Thus the vetted customizations run fully confined, without access to any external world.

Although the custoimizations run confined, because they can arbitrarily mutate the shared primordial state before other code runs, all later code is fully vulnerable to these custiomizations. This is why we refer to them as *vetted customization code*. Once the shared primordial state is transitively frozen, then we can support the standalone SES environment described above, where compartments are units of protection between subgraphs of mutually suspicious objects.

A analogy is that vetted customizations are what a shopkeeper does to their shop in preparation for opening for business. Freezing the primordials is the last step before opening the doors and allowing in untrusted customers.

In an IoT context, we should associate these two stages with build-time and runtime. The build-time environment should support more of the Realms and SES APIs for creating a SES world, that would be absent from within the standalone SES world they are creating. The freezing of the primordials is the snapshotting of the post-constomization primordial state for transfer to ROM.

SES is a JavaScript package that allows you to run third-party code safely. It runs in Node.js and in the browser.