Proposal Update:
Function Implementation Hiding

Stage 2 • Michael Ficarra • 71st Meeting of TC39

Considered Use Cases

no information leaks

1. the function *does not* appear in stack traces
2. the function's source text (including parameter list) is not available through `F.p.toString`

appear as a built-in (for polyfilling)

1. the function *does* appear in stack traces
2. the function has no file attribution or position info in stack traces
3. the function is rendered as a `NativeFunction` by `F.p.toString`

Considered Use Cases (cont.)

no unintentional API

1. the function *does* appear in stack traces
2. the function has no file attribution or position info in stack traces
3. the function's source text (including name and parameter list) is not available through `F.p.toString`
4. the function's `length` property is not necessarily based on its parameter list
5. the function's `name` property is not necessarily based on the syntactic binding

call site insensitivity & refactoring helper functions

1. the function *does not* appear in stack traces

Summarisation of Features

A

1. the function *does* appear in stack traces

¬A

2. the function *does not* appear in stack traces

B

3. the function has no file attribution or position info in stack traces
4. the function is rendered as a `NativeFunction` by `F.p.toString`

C

5. the function's `length` property is not necessarily based on its parameter list
6. the function's `name` property is not necessarily based on the syntactic binding

Alignments

no information leaks

¬A and B

appear as a built-in (for polyfilling)

A and B

no unintentional API

A and B and C

call site insensitivity &
refactoring helper functions

¬A

A Proposal?

A

~~1. the function *does* appear in stack traces~~

¬A

2. the function *does not* appear in stack traces

B

3. the function has no file attribution or position info in stack traces
4. the function is rendered as a `NativeFunction` by `F.p.toString`

C

~~5. the function's `length` property is not necessarily based on its parameter list~~
~~6. the function's `name` property is not necessarily based on the syntactic binding~~

A Proposal?

1. A directive, tentatively "sensitive":

   ¬A
   1. the function *does not* appear in stack traces

   B
   2. the function has no file attribution or position info in stack traces
   3. the function is rendered as a `NativeFunction` by `F.p.toString`

2. A second directive for hiding source text info, tentatively "hide implementation":

   B
   1. the function has no file attribution or position info in stack traces
   2. the function is rendered as a `NativeFunction` by `F.p.toString`

Alternative 1: APIs

- (¬A) `Error.hideFromStackTraces(f):` An API for omitting a function from stack traces, globally.
- (B) `Error.hideImplementationDetails(f):` An API for hiding position info from stack traces and source text from `F.p.toString`.

Pros

1. No new directives.

Cons

1. Anyone with a reference implicitly grants capability to omit from stack traces or hide implementation details when intending to grant capability to call/inspect.
2. A combined API for the security-sensitive use case would be much easier.
3. Less convenient than a directive.
4. Harder to statically enforce.

Alternative 2: Schrödinger's Directive

A directive which unconditionally hides source text / stack trace location information (feature B), and makes omission from stack traces (feature A) conditional upon whether the function is declared using one of the export declaration productions (`export function f(){}` or `export default function(){}`).

Pros

1. Usually does what you want.

Cons

1. Could be quite confusing/surprising.
2. Should public class members also be special-cased?

Alternative 3: Additive Directives

"sensitive" is a superset of "hide implementation". We could instead make them additive, e.g. "hide from stack traces" plus "hide implementation".

Pros

1. Orthogonal primitives.

Cons

1. No singular opt-in for "security sensitive" code.
2. No clear use case for "hide from stack traces" independent from "hide implementation".

Alternative 4: Chaining Behaviour

Any function called by a "hide implementation" function is automatically omitted from stack traces ("sensitive" behavior).

Pros

1. Only one directive

Cons

1. Security-sensitive code (especially recursive code) must use a wrapper pattern to be completely hidden.