Hardened JavaScript

Vulnerability Assessment Report

by Agoric and MetaMask

Background **4**

Introduction to hardened JavaScript **5**
- The Specification 5
- Components of the hardened JavaScript shim implementation 7
  - Lockdown 7
  - Compartment 7
  - Proxy Trap and ScopeProxyHandler 7
  - Optimizer 8

Assessment Overview **9**

Scope **10**
- Target Code 10
- Assessment Goals 10
- Criticality Rating 11

Findings **12**
- Summary 12
- Critical Findings 13
- High Severity Findings 13
- Medium Severity Findings 13
- Low 14
  - Finding-01: The has hazard 14
  - Finding-02: Bypass of Compartment global lexicals 16
  - Finding-03: Compartment leaks its observation of the presence of properties on start compartment globalThis 17
  - Finding-04: Scope proxy defense against property descriptor prototype 18
  - Finding-05: Investigate allowlist in relation to getSubPermit implementation 18
- Informational 19
  - Finding-06: Limit ‘lockdown’ calls 19
  - Finding-07: Strict mode optimizer for evaluate 19
  - Finding-08: Allow Map to be named in Module Scope 20
  - Finding-09: Avoid use of receiver object this in scope handler 20
  - Finding-10: Avoid use of receiver object this in scope handler 21
  - Finding-11: Clarify internal terms for culling disallowed intrinsics 21
  - Finding-12: Use eslint-disable notation consistently 22
  - Finding-13: Add an up reference to --proto-- in hardened JavaScript permits 22
  - Finding-14: Simplify makeCompartmentConstructor 23
  - Finding-15: Add tests for backslash behavior 23
  - Finding-16: Add Lint to disallow hardened JavaScript polymorphic calls 24
  - Finding-17: Propose ECMA 262 language invariant for proxy handlers 24
  - Finding-18: Improve documentation around partial hardening scenarios 25
  - Finding-19: Rename localObject to globalLexicals in performEval 25

Finding-20: makeHardener as an arrow function 26
Finding-21: Add assertions for reflexive subsections of the hardened JavaScript permits declaration 26
Finding-22: Verify that AsyncFunctionPrototype is both de jure and de facto standard 27
Finding-23: Freeze proxy handlers for scopes 27
Finding-23: Refactor evaluate function for readability 27

Observations on the hardened JavaScript shim **29**
Considerations for Consumers of the hardened JavaScript Shim 30
Sanitize the globalThis as much as possible 30
Handle the start compartment carefully 30
Consider tools for reading deeply nested code 31
The “partial hardening hazard” 31
Intent and Recommended Areas of Focus for Future Assessments 32
Non-nested containment 32
Compartment initialization could be interleaved by Component creator 33
Obscure spec pitfalls 33

Observations from the MetaMask Red Team on In-Language Confinement APIs **34**

Background

Agoric's hardened JavaScript shim is a library that is designed to provide an object-capability security model to a modern JavaScript environment, enabling the execution of untrusted code within the same synchronous environment. In order to do this, first an environment is locked down to ensure predictable behavior, and second, a Compartment API is provided to confine untrusted scripts.

JavaScript is considered to be notoriously difficult to secure, but as a highly lexical language with a long history of object-capability advocates on the TC-39 Committee, it appears to be on the cusp of achieving in-language security properties that are extremely rare, valuable, and intuitive to use.

Object-capabilities is a security model that exists as a highly flexible alternative to traditional access control strategies (e.g. access control lists), wherein a given entity receives no authority by default, and receives all of its power only by being explicitly passed object references, either through construction or over the course of activity.

Outside of the web stack, runtime isolation usually manifests itself in technologies like application-level sandboxes, containerization, and/or full VM segmentation. If the hardened JavaScript shim can be trusted, and a Compartment API provided, the shim could allow a similar type of isolation at the JavaScript layer, and result in significant improvements for security, lightweight extensibility, and speed of safe development for JavaScript applications with little to no modification to existing code.

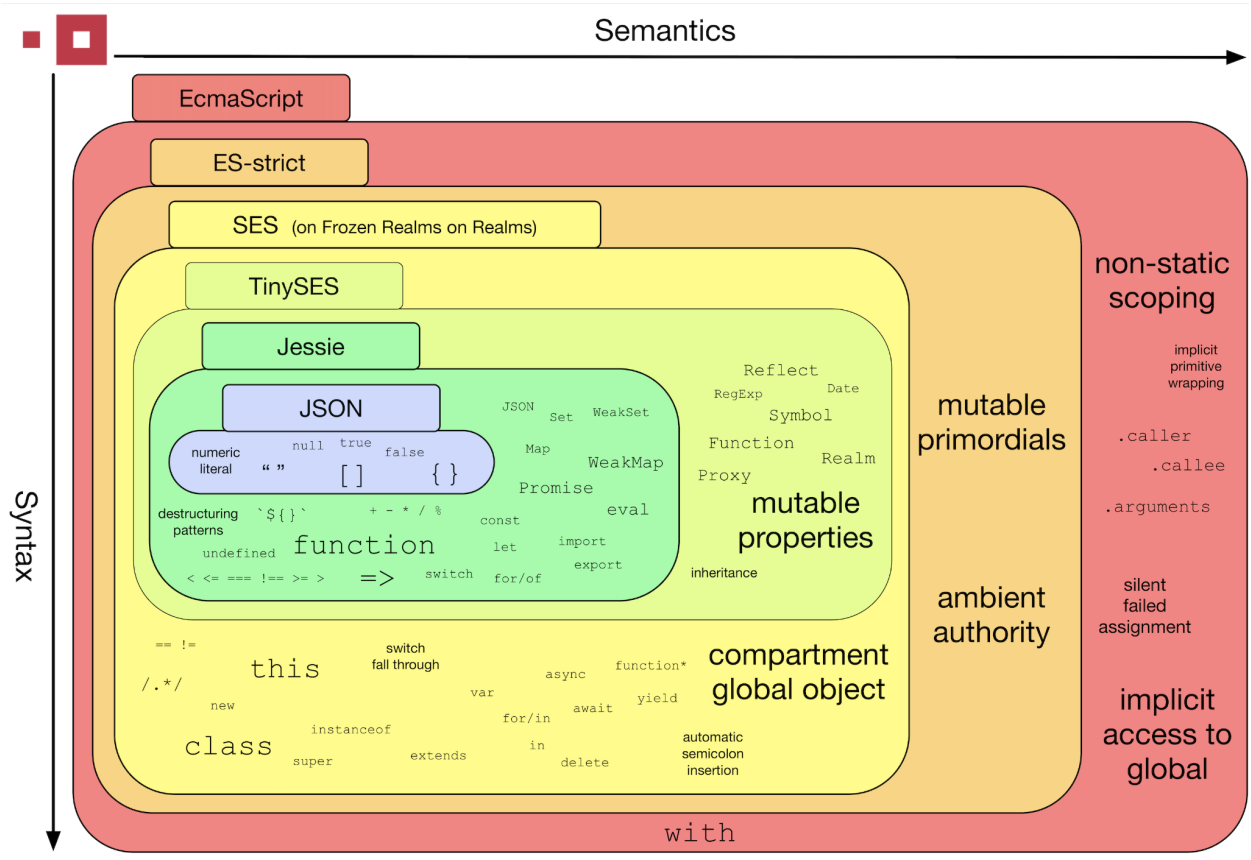
In this report, we will describe the results of a limited-time evaluation of the security of the hardened JavaScript shim. The goal of our review was to evaluate its security properties, and in particular, its ability to confine malicious code. The primary target of this evaluation was not the Compartment specification, but the hardened JavaScript shim which was written to emulate its behavior in modern JavaScript environments.

Introduction to hardened JavaScript

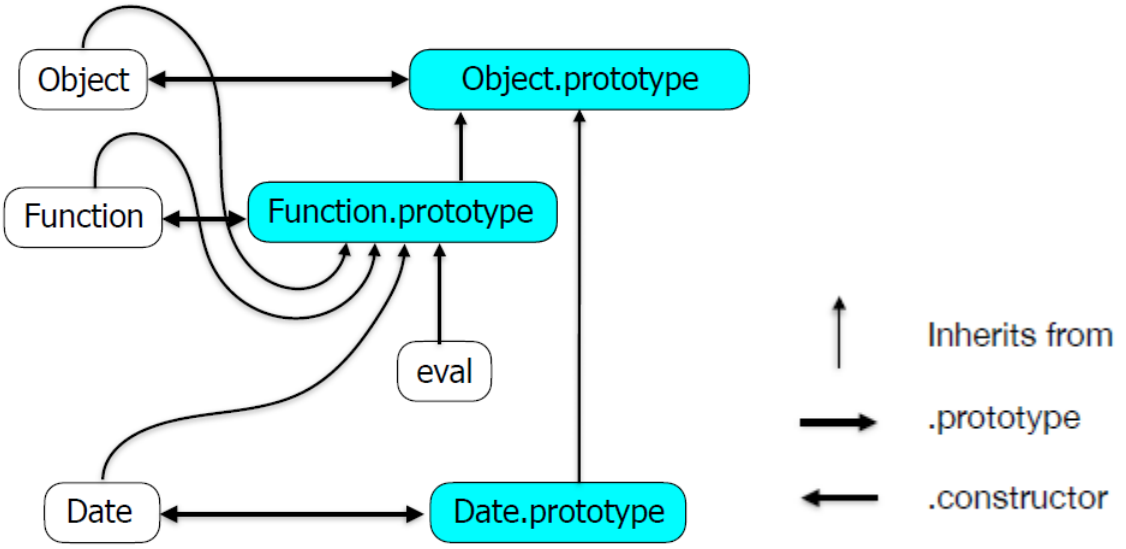
The Specification

The Agoric documentation defines hardened JavaScript as a safe deterministic subset of the strict mode in JavaScript. This means that a number of additional restrictions have been applied to strict mode JavaScript. One of the major changes hardened JavaScript enforces is on the standard globals by removing IO objects that could allow for exfiltration of data out of a hardened JavaScript Compartment. In addition, hardened JavaScript lockdown freezes the shared intrinsics, so attackers cannot replace their methods or otherwise subvert them. The hardened JavaScript guide provides a detailed description of the additions, removal and modifications to the ECMAScript specification.

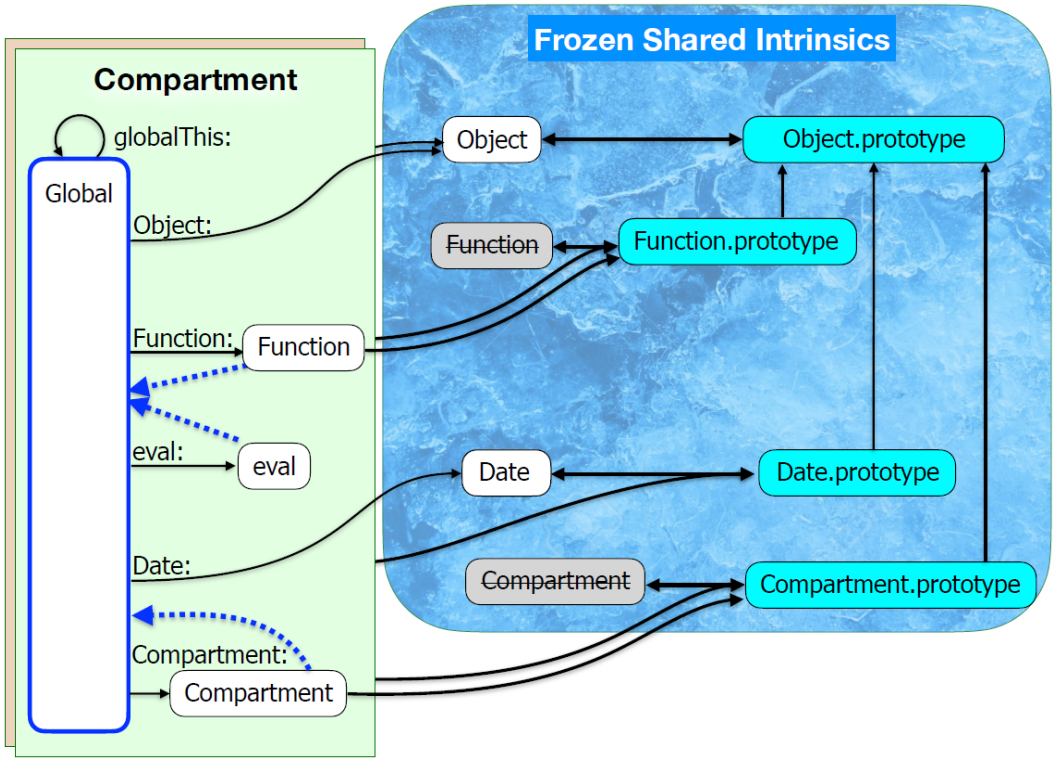
Here is a summary of the language features available in each of these subsets of JavaScript. In this diagram, hardened JavaScript is referred to as SES:



Even under strict mode, access to many intrinsics can result in climbing the prototype/constructor chain to gain access to more powerful prototypes, which can allow polluting the context of other scripts:



During the lockdown phase, one of the goals of the hardened JS shim is to not only freeze these intrinsics, but to replace their constructors with frozen dummy versions so that a malicious actor climbing through the object's properties cannot escape their compartment:



Components of the hardened JavaScript shim implementation

Lockdown

Through an initial `lockdown()` invocation, the hardened JavaScript environment enumerates all known global environment intrinsics and ensures they are deeply (recursively) frozen, and that none of them have `.constructor` or `.prototype` chains that can be used to access the original (mutable) feral objects.

Compartment

Through the so-called “8 magic lines” (which stand in for several more lines of code), the basic isolation of untrusted JavaScript is claimed:

```
return FERAL_FUNCTION(`
 with (this) {
   ${optimizer}
   return function () {
     "use strict";
     return eval(arguments[0]);
   }
 }
`);
```

The `FERAL_FUNCTION` is the original `Function()` constructor, and can be thought of as an indirect `eval`, whose scope is the top level of the executing script.

Proxy Trap and ScopeProxyHandler

The `with` block is used to create a new lexical scope where all property lookups first fall on `this`, which is itself a Proxy trap, designed to catch all lexical lookups on the inner scope, and handle them as specified by the `ScopeProxyHandler`, which is constructed to pass properties

to the Compartment's endowments and shared frozen intrinsics. The `with` block is only allowed in JavaScript "sloppy mode", which is why `use strict` is required within it, but not before it.

Optimizer

The optimizer is intended to simply make some frequently accessed variables available without invoking the more computationally expensive proxy trap.

The strict mode directive is then invoked to provide a variety of environment-level security guarantees that are needed to prevent trivial violation of the object capability guarantees.

Finally, an invocation of `eval()` with the Compartment's intended code to be executed. Note that this invocation of `eval()` will bubble up to the proxy trap, which needs access to a feral `EVAL` only on this first invocation.

See also: https://youtu.be/mSNxsn0pK74?t=801

Assessment Overview

The MetaMask and Agoric teams kicked off their Red Team/Blue Team vulnerability assessment of the shim beginning on July 5, 2021 for a period of 5 days. For the duration of the engagement, the MetaMask team (with the addition of one member of the Agoric team) took on the offensive posture of a Red Team, and Agoric's module maintainers took on the defensive posture of a Blue Team during this collaborative, cross-organization vulnerability assessment.

Red Team	Blue Team
Aaron Davis, MetaMask	Mark S. Miller, Agoric
Dan Finlay, MetaMask	Kris Kowal, Agoric
Erik Marks, MetaMask	Dan Connolly, Agoric
Seth Kaufman, MetaMask	Dean Tribble, Agoric
Cal Leung, MetaMask	
Ryan Lanese, MetaMask	
Mathieu Hofman, Agoric	

During the first two days of the engagement, Agoric's module maintainers presented a walkthrough of the source code and architecture of the module, and highlighted potential areas of risk where issues were likely to be found. On days 3 and 4 of the assessment, the MetaMask team reviewed the list of potential attack vectors, and used a Value/Effort framework to guide review prioritization. The MetaMask team led the analysis of "hot spot" areas using static analysis and fuzzing tools. The final day of the engagement was reserved for authoring a report of findings… and sushi.

Throughout the assessment, both teams presented code walkthroughs, observations, and findings, which are available on YouTube.

- Session #1: https://www.youtube.com/watch?v=SDWEJ9H2xgw
- Session #2: https://www.youtube.com/watch?v=3ngSBIm8rD0
- Session #3: https://youtu.be/MaJSQHgmGmk
- Session #4: https://youtu.be/HMsQzI6L-70

- Session #5: https://youtu.be/liAjywIlp9A

Scope

Target Code

For this review, all teams involved in this assessment reviewed the following code targets with the SHA1 hash of `f91c84cde6cfe82c085ebe316da939d04ca74aa4`.

- https://github.com/endojs/endo/tree/f91c84cde6cfe82c085ebe316da939d04ca74aa4

Assessment Goals

The investigation of this code focused on ensuring, as delineated in the hardened JavaScript documentation, that Compartment confined code should be confined such that it does not have any of the following abilities:

- Access to powerful platform capabilities except where endowed, including:
    - network access,
    - disk access,
    - and access to timers.
- Access to observe the runtime environment, including via Spectre type attacks (without being explicitly granted access to a timer).
- Unauthorized ability to communicate with other compartments

    We considered these objectives in the presence of two threat scenarios:
    - Threat 1: code in the Compartment can break out.
    - Threat 2: creator of a Compartment can break out of a locked down environment, and mutate its realm.

The following conditions for compromising the security properties of the shim were considered out of scope:

- Corruption of the start compartment before it is locked down.

- Running hardened JavaScript in a non-standard host environment that cannot be locked down.
- Endowing a compartment with an intentional sandbox escape, or side-channel like a timer.
- Using features of JavaScript runtimes that are outside of the semantics of the ECMAScript specification, such as Mozilla's Debugger API.
- Passing the lockdown function and Compartment constructors with options that are designed to deliberately weaken their security guarantees as a feature for development.
- Running an infinite loop is acknowledged as a risk of exposing the Compartment API to untrusted parties, and so exhausting system resources alone will not be considered a successful breach of the system's security.

Criticality Rating

Critical	*High Impact + High Likelihood of Exploitation.* A highly exploitable issue that provides an attacker with full control of the program's execution, and violates all security assumptions, e.g. a full sandbox escape with full privileges.
High	*Medium Impact + Medium Likelihood of Exploitation.* An exploitable issue that affects core tenants/assumptions of the program's security assumptions and guarantees, e.g. a sandbox escape with partial privileges.
Medium	*Low Impact + Medium Likelihood of Exploitation*. An exploitable issue that complicates the program's execution without directly resulting in a full compromise of integrity of the program, e.g. tampering with a method's return value.
Low	*Low Impact + Low Likelihood of Exploitation.* An issue that would introduce unexpected behavior to the program without directly resulting in a compromise of security assumptions, e.g. difficult to exploit information disclosure.
Informational	*Little to no Impact + Little to no Likelihood of Exploitation*. An issue that does not present a direct exploitable risk to the program's execution, e.g. documentation, code style, non-exploitable in real world usage findings.

Findings

Summary

This report details the results of an intensive analysis of the hardened JavaScript shim implementation. During this engagement, the MetaMask Red Team was asked to assess the implementation, and to look for implementation bugs that could lead to the compromise of the stated security guarantees.

Ultimately, the Red Team found the shim to be written with extraordinary care and expertise. Some of our biggest concerns were related to maintainability or ensuring correctness in the face of a changing specification. It seems clear that the shim's lockdown alone provides significant improvement to the safety of written JavaScript, and the Compartment API brings a compelling interface for safe confinement of code. To the best of our abilities, we did not find any escapes from this confinement, and are optimistic about its long term potential, but even more so as a possible language feature.

We are happy to have found some improvements and recommendations, and our intention was to contribute to securing the underlying platform. Over the course of this engagement, we gained a deeper appreciation of the depth of mastery of JavaScript at play at Agoric, as well as the subtle complexity of the language as it has evolved over its decades of non-breaking evolution. Due to the number of global APIs and their often inconsistent and unusual mutability, JavaScript as shipped by browsers today is hard to reason about safely. With hardened JavaScript confinement, users have access to a version of the language with an interface that enables safer, more reliable behavior.

Thanks to the language features and safety properties that Mark Miller and other JavaScript advocates have advanced at TC-39 over the last decade, it is looking hopeful that if safe JavaScript is not here yet, it may be soon. The language features that have made the shim possible did not come for free, but required diligent continuous work at the language committee level, both to advocate for new forms of safety and confinement (like strict mode and the Proxy), as well as a strong defense against the all-too enticing language feature that might introduce widespread security compromises.

With more value being entrusted to the web than ever, we look forward to the union of security and simplicity that comes from a language environment like hardened JavaScript.

Critical Issues

No Critical findings were surfaced in the scope of this assessment.

High Severity Issues

No High findings were surfaced in the scope of this assessment.

Medium Severity Issues

No Medium findings were surfaced in the scope of this assessment.

Low Severity Issues

Finding-01: The `has` hazard

Criticality	Low
Classification	Code Change
Github Issue	https://github.com/endojs/endo/pull/820, https://github.com/endojs/endo/pull/898

An undocumented hazard was identified in the scope proxy handler code where an innocuous looking refactor to hardened JavaScript could lead to the exposure of the unsafe/feral `eval` in child compartments, however this is not exploitable in the current implementation. The `has` hazard (as we nicknamed it) stems from interactions between the "8 magic lines", and the current attempt of the shim to replicate `ReferenceError`s when the code evaluates an unknown identifier.

When code is evaluated through the "8 magic lines", `eval` is looked up on the scope proxy, first triggering the `has` trap, then the `get` trap. This lookup is done while the "unsafe eval guard" is down to allow the "8 magic lines" to access the feral `eval`, with the `get` trap in charge of locking the guard back up immediately.

To attempt faithful `ReferenceError` emulation, the `has` trap checks if a property exists on the global object, the lexical object or if it is `eval` to satisfy the logic of the "8 magic lines." At the time of discovery, it was not clear whether the property check on the global object could trigger user code if the global object's prototype was substituted for a proxy. Also noteworthy, the shim has a known limitation that allows scope proxy objects to leak to user code under certain circumstances. This `has` hazard did not introduce a vulnerability thanks to multiple existing behaviors of the shim: the "8 magic lines," while designed to allow the internal `evaluate` to be reused across evaluations, did not do so, and the `has` trap uses a disjoint "`or`" (`||`) that checks for `eval` before looking up the global object.

The first behavior of not reusing existing internal evaluators meant that even if code could be interspersed in the `has` trap, it wouldn't be able to access the unguarded feral `eval` since it couldn't obtain a leaked reference to the fresh scope proxy that was just created for this

new evaluation. However reusing evaluators is an optimization that is planned, so it cannot be relied upon solely as a mitigation.

The second behavior prevents user code from being interspersed through a global object prototype proxy while the guard is down, and it is now understood this must always be guaranteed as we cannot prevent such global object modifications. For other reasons (e.g. start compartment `globalThis` property existence leak), we are also planning on always returning `true` in the `has` trap, which would further prevent this issue.

With the changes to the shim described above, the following code would be sufficient to access the original unsafe `eval`:

```
const c = new Compartment({});
const unsafeEval = c.evaluate(`
  let unsafeEval;
  let inHas = false;
  globalThis.__proto__ = new Proxy({}, {
    has: function (...args) {
      if (args[1] === "eval") {
        if (!inHas) {
          inHas = true;
          unsafeEval = eval;
          inHas = false;
          globalThis.eval = unsafeEval;
        }
        return true;
      }
      return Reflect.has(...args);
    },
  });
  const safeEval = globalThis.eval;
  delete globalThis.eval;
  safeEval("");
  unsafeEval;
`);

globalThis.Date = unsafeEval("Date");
console.log(Date.now());
```

Finding-02: Bypass of Compartment global lexicals

Criticality	Low
Classification	Code Change
Github Issue	https://github.com/endojs/endo/issues/876

The `Compartment` shim has an option for providing global lexical values, which are available in the lexical scope of the compartment, but are not assigned to the `Compartment`'s `globalThis`.

If code within a `Compartment` uses an evaluator (either the `Function` constructor or `eval`), its evaluation will be evaluated without hitting the global lexical configuration, and passing directly to the `globalThis`. This was previously used by Agoric's metering transform to expose the metering function as a global lexical. This issue would have allowed a bypass of that transform.

This is not an escape from the `Compartment`, but a violation of the security properties that would be expected from the global lexical configuration.

The `Compartment` API as proposed to TC-39 does not appear to have a feature for exposing global lexicals (as opposed to values on the global object), and so this issue does not apply to the `Compartment` spec.

Finding-03: Compartment leaks its observation of the presence of properties on start compartment `globalThis`

Criticality	Low
Classification	Code Change
Github Issue	https://github.com/endojs/endo/pull/820

Due to a combination of seeking to allow a `Compartment` to fully virtualize a host environment and the behavior of the Proxy API, the hardened JavaScript shim was forced to make a tough decision between behavior that most resembles a normal host environment, and behavior that maximally obscures information about the host environment. We found the shim to be leaking information about the host environment.

The issue occurs when the guest code performs a `typeof foo` instead of a direct `foo` access for a non-defined variable. In strict mode, `typeof` would return `undefined` and `foo` would throw a `ReferenceError`.

Emulating this behavior in a Compartment through the shim is not possible. If the proxy trap for `has` returns false, then `typeof foo` gets `undefined`, and a lexical use of that identifier would cause a `ReferenceError`, as a normal program would expect. However, if we return false for a value that exists on the start compartment's global scope, it would expose the variable as it exists on the start compartment's `globalThis`.

To mitigate this information leak, it is probably worth violating platform consistency for the sake of minimizing the guest's ability to probe the host environment, by returning `true` from the `has` trap for all variables that exist on the start compartment's `globalThis`. This mitigation would operate the same way, but it would result in a loss of `ReferenceError` in the environment. For the Compartment specification, it is likely worth specifying the more virtualized behavior, where a `ReferenceError` should be thrown for variables that are not in a compartment's scope (which cannot be emulated with a shim today).

Finding-04: Scope proxy defense against property descriptor prototype

Criticality	Low
Classification	Code Change
Github Issue	https://github.com/endojs/endo/pull/834

An improvement was made to the scope handler to protect against prototype pollution of the `value` property on `Object.prototype` before lockdown.

Finding-05: Investigate allowlist in relation to `getSubPermit` implementation

Criticality	Low
Classification	Test Development
Github Issue	https://github.com/endojs/endo/issues/835

An improvement was proposed for test development to validate the `getSubPermit` function in `allowlist`.

Informational

Finding-06: Limit `lockdown` calls

Criticality	Informational
Classification	Code Change
Github Issue	https://github.com/endojs/endo/issues/814

An improvement was made to restrict the calling of `lockdown` to only happen once, to simplify lockdown behavior.

Finding-07: Strict mode optimizer for evaluate

Criticality	Informational
Classification	Code Change
Github Issue	https://github.com/endojs/endo/issues/816

To reduce attack surface, this improvement enforces strict mode on the optimizer instead of utilizing sloppy mode.

Finding-08: Allow Map to be named in Module Scope

Criticality	Informational
Classification	Code Change
Github Issue	https://github.com/endojs/endo/issues/823

An improvement was made to all `Map` to be named in module scope during the hardened JavaScript module initializer.

Finding-09: Avoid use of receiver object `this` in scope handler

Criticality	Informational
Classification	Code Change
Github Issue	https://github.com/endojs/endo/pull/829

An improvement was made to increase the integrity of intrinsics after compartment initialization and before lockdown in the start compartment.

Finding-10: Avoid use of receiver object `this` in scope handler

Criticality	Informational
Classification	Code Change
Github Issue	https://github.com/endojs/endo/pull/833

An improvement was made to use a controller object with explicit named methods instead of the `this` on the scope proxy handler object which create clear actions rather than an abstraction of functionality, which may cause confusion or collide other Proxy objects in the namespace.

Finding-11: Clarify internal terms for culling disallowed intrinsics

Criticality	Informational
Classification	Code Change
Github Issue	https://github.com/endojs/endo/pull/836

A naming refactor was applied to the allowed intrinsics file function names to use the terms `visit` and `allowed`. Minor logic changes were made to the `visitPrototype` function to conform to the terms.

Finding-12: Use `eslint-disable` notation consistently

Criticality	Informational
Classification	Configuration Change
Github Issue	https://github.com/endojs/endo/pull/837

An update was made to use `eslint-disable` notation consistently throughout the code base.

Finding-13: Add an up reference to `--proto--` in hardened JavaScript permits

Criticality	Informational
Classification	Documentation Change
Github Issue	https://github.com/endojs/endo/pull/838

A documentation update was made to reference the difference between [[Proto]] and --proto--.

Finding-14: Simplify `makeCompartmentConstructor`

Criticality	Informational
Classification	Code Change
Github Issue	https://github.com/endojs/endo/issues/815

An improvement was proposed to only have a single `makeCompartmentConstructor`.

Finding-15: Add tests for backslash behavior

Criticality	Informational
Classification	Test Development
Github Issue	https://github.com/endojs/endo/pull/817

A test case was created to evaluate if the platform mishandles the backslash & unicode in the `evaluate` function.

Finding-16: Add Lint to disallow hardened JavaScript polymorphic calls

Criticality	Informational
Classification	Test Development
Github Issue	https://github.com/endojs/endo/issues/818

A lint rule was introduced for the hardened JavaScript repository to guide the use of exports of `common.js` globals that are possibly poisoned.

Finding-17: Propose ECMA 262 language invariant for proxy handlers

Criticality	Informational
Classification	Third Party to Resolve
Github Issue	https://github.com/endojs/endo/issues/819

This is a proposal to ECMA 262 language to maintain the integrity of the `get` invariant for proxy handlers.

Finding-18: Improve documentation around partial hardening scenarios

Criticality	Informational
Classification	Documentation Change
Github Issue	https://github.com/endojs/endo/issues/825

This is a proposal to add more documentation on the safe use of `harden` to avoid partially hardened objects.

Finding-19: Rename `localObject` to `globalLexicals` in `performEval`

Criticality	Informational
Classification	Code Change
Github Issue	https://github.com/endojs/endo/issues/840

An improvement was proposed to rename the `localObject` to `globalLexicals` to help readability in `performEval`.

Finding-20: `makeHardener` as an arrow function

Criticality	Informational
Classification	Code Change
Github Issue	https://github.com/endojs/endo/issues/841

An improvement was proposed to `makeHardener` an arrow function.

Finding-21: Add assertions for reflexive subsections of the hardened JavaScript permits declaration

Criticality	Informational
Classification	Documentation Change
Github Issue	https://github.com/endojs/endo/issues/842

A documentation improvement was proposed to document and add assertions for reflexive subsections in the `whitelist.js`.

Finding-22: Verify that `AsyncFunctionPrototype` is both de jure and de facto standard

Criticality	Informational
Classification	Documentation Change
Github Issue	https://github.com/endojs/endo/issues/843

An improvement was proposed to verify and add a comment in `whitelist.js` for the `AsyncFunctionPrototype` specification versus current implementation.

Finding-23: Freeze proxy handlers for scopes

Criticality	Informational
Classification	Code Change
Github Issue	https://github.com/endojs/endo/pull/846

An improvement was proposed to use `create` instead of `__proto__` to freeze the proxy handlers.

Finding-23: Refactor evaluate function for readability

Criticality	Informational
Classification	Code Change
Github Issue	https://github.com/endojs/endo/issues/907

An improvement was proposed to improve readable organization of a security-critical function.

Observations on the hardened JavaScript shim

During this assessment of the hardened JavaScript shim, the Red Team noted:

1. Performance could be somewhat improved by creating a shared evaluator for all compartments. This change has additional benefits related to avoiding the has hazard (Finding-01) above.
2. The maintainers might consider adding a second `with` statement around the main `with` that only has an “always fail proxy”. This could provide one additional fail-safe in the case that anything that we have not accounted for allows escaping the inner context. It is unclear what the performance implications of this change would be.
3. Consider adjusting the optimizer for code that does not need all globals cached, like short-lived compartments. This is a possible performance improvement, not a security one. This also may not be as useful when using shared evaluators.
4. One recurring theme during this vulnerability assessment was the challenge of understanding the shim, as it’s both very complex and split among many files. Since comprehension is an important component of defensive maintenance, some extra care towards making the code more accessible to third parties could be helpful. An example of this, could be providing an expanded code walkthrough in documentation or a sample project.
5. A refactor of the `permit` code would make it easier to understand.
6. There are benefits to applying a polymorphic call exception to the linter. The shim uses the global object and functions on it liberally, and since it is not frozen, those global functions may have been changed. During setup, it would be a good practice to make a local copy of any functions that will be used during the course of operation. This is a spec fidelity issue, not a vulnerability, since any changes to these objects before lockdown is out of scope. This could especially create trouble when using vetted shims to add additional global objects or functions before lockdown.

Considerations for Consumers of the hardened JavaScript Shim

Sanitize the `globalThis` as much as possible

One way to mitigate the potential damage of successful sandbox escapes would be to remove properties on the start compartment `globalThis`, as a way to limit global authority generally. Unfortunately, the `globalThis` has some sensitive properties that can not be removed, like `document` and `location`. This may be a place where a standards proposal could provide a method of locking down the global environment further. Since this is likely overly destructive to add to the `lockdown()` function, we leave this as a possible recommendation to Compartment consumers. This can look like making a local copy of any globally-available APIs needed, and then removing them from the shared `globalThis` before invoking any third party code, including importing a third party module.

Handle the start compartment carefully

The start compartment after lockdown keeps its original powerful endowments, but is supposed to be left only with “safe” native evaluators (e.g. `eval`, `Function` and `Compartment`). Technically, if the program removed all original endowments from the start compartment after lockdown, it should be no more powerful than any child compartment without these endowments. However, the “tamed”/“safe” `eval` and `Function` constructor have little restrictions compared with the feral kinds or for that matter any further code executing directly in the start compartment. For example, they all have direct access to the start `globalThis` object.

Because there is no way to inject global lexicals or modify the module loader of the start compartment (lack of instance access), the only difference of the “safe” evaluators in the start compartment is that all code executed through them should be in strict mode. We found that host code in the start compartment could accidentally alter the environment such that they could reveal the feral eval to themselves. This was caused by the shim itself executing in the start compartment and relying on a dynamic lookup of `WeakSet` as a free variable in scope after lockdown. If the global `WeakSet` were accidentally replaced, invocations of `add` would be interleaved with shim code while the feral eval was unguarded. This is a re-entrancy hazard. Since this issue didn't confer any more access to start compartment code than it already had, there is no vulnerability, but it was noteworthy to illuminate how exposed the shim is to start

compartment code, and informed the decision to lint against any dynamic usage of start compartment's globals after lockdown.

Consider tools for reading deeply nested code

During the assessment, it was strongly recommended to browse the source code with a modern JavaScript IDE such as Visual Studio Code. In terms of navigation, the IDE's ability to jump around to definitions is helpful to reason with the code's structure. The hardened JavaScript shim is a well developed, complex codebase which requires modern tooling to successfully navigate.

The partial hardening hazard

The `harden` function traverses and freezes objects in place. If the function throws during traversal of the object graph, that object graph will be partially hardened. Partially hardened objects can become attack vectors, e.g. if they are shared between mutually suspicious compartments. This hazard can be avoided through defensive programming, in particular by following the principle of "harden early and often, and dispose of partially hardened objects even sooner".

If creating an `isHardened()` function, since a caught error could cause an object to be partially finalized as hardened when in a hardened state that should be impossible, consider throwing another error on that test. We know that an out of memory error is one case that might be able to cause this, and a developer would only be prone to this problem if they ignored that error. This issue appears in Github as **endo#825**

Intent and Recommended Areas of Focus for Future Assessments

Part of the MetaMask team's interest in the hardened JavaScript approach is that it provides an interface that simplifies the process of writing safer, more secure code. This API increases the likelihood that some executed code is unable to cause unintended consequences. While the implementation focuses on usability, its approach builds on several lesser-used language features, some of which have some surprising behaviors.

During the collaborative codebase review phase of the engagement, the MetaMask team noted a list of areas of the shim that are critical to ensuring the security of the implementation. We ruled out categories of concern when possible, and the rest of the time focused on the most hypothetically-dangerous mistakes that we could imagine.

The Red Team found no immediate concerns or immediately actionable steps beyond the issues presented as findings in this report. In the event that we were asked to reassess this code for vulnerabilities, we would start with the targets listed below.

- The evaluate function, including any of the transforms it performs, like re-inserting comments into trusted scope.
- Exploring ways of acquiring the scope handler from within a Compartment. While this was possible using Firefox debug tools, the use of developer tools was out-of-scope for this assessment, and no other approach was found.
- Experimenting with the `ScopeProxy` (which is reifiable), to try to get its feral `eval` function from it.
- Explore the implications of the Compartment constructor and evaluator tripping `get` and `have` traps on the object it's given, and whether or not a confined script could craft a Proxy to allow re-entrancy and access to feral values.

In addition to this list, we recommend evaluating the areas outlined below.

Non-nested containment

When creating a fully virtualized Linux virtual machine within another fully virtualized virtual machine, a sandbox escape from the second machine should only land you into its creator and not any higher. However in hardened JavaScript, Compartments created by Compartments are "siblings", both children managed equally by the hardened JavaScript system. Since Compartments running untrusted code can create and configure their own child Compartments, The Compartment must be secure against bad constructor parameters and configuration. If there was a way to create a Compartment configured to disable sandboxing, an

attacker could use this to access the start compartment. It may be possible and desirable to implement the shim such that the Compartment implementation is re-defined in the context of the parent compartment, preventing an escape from leading directly to the start compartment.

Compartment initialization could be interleaved by Component creator

The compartment initialization logic in the shim is particularly sensitive as it leverages the powerful `eval`. If an attacker was able to interleave malicious code at the right point during initialization, it may be able to access `eval` and break out to the start compartment. While this was not an active vulnerability at time of review, introducing this vulnerability only required a small code change. This attack was possible after a small change was introduced to the `scopeProxyHandler` implementation.

```
const c = new Compartment()
c.globalThis.__proto__ = new Proxy({}, { has (_, key) { debugger } })
```

Obscure spec pitfalls

Many of the security features of hardened JavaScript depend on core JavaScript functionality, and the ECMAScript specification is so large that even the most capable experts cannot know the spec in its entirety. A fundamental change to JavaScript itself could introduce attack surface or vulnerabilities into hardened JavaScript as uncommon or poorly understood facets of the specification could break assumptions that the security guarantees of hardened JS rely on.

An example of this is present in the hardened JS shim as it relies on the `with` statement. The `with` statement is augmented by a `Symbol.unscopables` property, which is relatively obscure as it only augments the somewhat deprecated `with` statement. This could allow a breakage from the containment if it was not handled. In this case, JS does handle it. But in the `with` statement section of the specification, there are no clear references to the `Symbol.unscopables` property and its effect on the `with` statement behavior. Other undocumented behaviors similar to this are likely exist in the specification, and significant changes to the spec that impact behaviors like this could introduce risk or vulnerability into hardened JavaScript.

Observations from the MetaMask Red Team on In-Language Confinement APIs

As representatives from the MetaMask cryptocurrency wallet, we have found the potential benefits of an in-language confinement API to be at least two-fold:

- As a tool for confining dependencies that we import to have less ambient authority, which can lead to vulnerabilities such as we've seen manifest in real-world attacks such as the event-stream incident. We have built a build-system tool that uses the hardened JavaScript compartment as an added isolation barrier for our dependencies in a tool we call LavaMoat.
- Since cryptocurrency is also a fast-moving industry, the pressure to experiment with new features is dangerously relentless. With runtime confinement, we can also experiment with ways for users to try adding new features to their own wallets while hopefully preserving the principle of least authority to those new scripts. This would minimize the possible damage from programmer error as well as the user risk from installing a malicious extension. We have pursued the concept of a crypto-wallet plugin system in a project we call Snaps.

These two motivations have different security demands, but benefit from the same guarantees. While a LavaMoat failure leaves a project in the same state as a normal JavaScript package with many dependencies, potentially leading to a false sense of security, it relies on the same confinement properties that a plugin system would demand.
A plugin system ideally allows a user to issue only the exact authority a script needs to execute, but if anything less than that guarantee is given, then those permissions' descriptions themselves should be weakened with a disclaimer to the user reflecting any uncertainty in the underlying platform by the host developers.