
# LavaMoat Webpack Plugin

Security Assessment

October 12th, 2024 — Prepared by OtterSec

Bruno Halltari bruno@osec.io

Caue Obici caue@osec.io

# Table of Contents

# 01 — Executive Summary

## Overview

Metamask engaged OtterSec to assess the `@lavamoat/webpack package`. This assessment was conducted between October 1st and October 10th, 2024. For more information on our auditing methodology, refer to Appendix B.

## Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we identified a vulnerability inside ContextModules, since those are excluded from the lavamoat runtime wrapper, allowing compromised modules to import restricted modules (OS-LMT-ADV-00). Furthermore, we discovered that is possible to exploit Webpack's handling of assets when utilizing the `URL()` constructor to import malicious files. (OS-LMT-ADV-01).

We also made a recommendation to incorporate more stringent validation to prevent arbitrary code execution outside the sandbox (OS-LMT-SUG-00) and advised verifying all chunk names during the build process to ensure they do not contain any path traversal sequences (OS-LMT-SUG-01).

# 02 — Scope

The source code was delivered to us in a Git repository at https://github.com/LavaMoat/LavaMoat. This audit was performed against a859f9f.

**A brief description of the program is as follows:**

| Name | Description |
| --- | --- |
| @lavamoat/webpack package | LavaMoat Webpack Plugin wraps each module in the bundle in a Compartment and enforces LavaMoat Policies independently per package. |

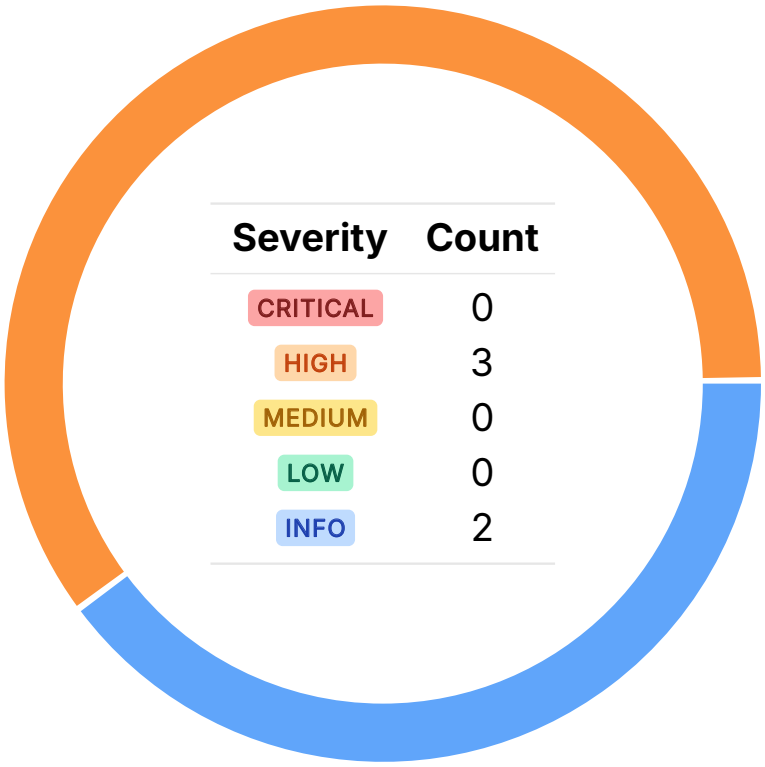
Below is the list of attack vectors we covered as part of the audit:

1. Abuses of runtime `__webpack_require__` exposed functions (including type confusions such as passing `objecs` and `proxies`).
2. Bugs in policy parsing and matching.
3. `aa` algorithm inconsistencies.
4. `ContextModule` bypasses, as it is excluded from wrapping.
5. Webpack build-time functions abuses (`require`, `import`, `require.ensure`...).
6. Code executions in other modules to gain global permissions.
7. Bypasses utilizing module types other than `javascript/esm`, `javascript/auto` and `javascript/dynamic` (checked all default rules and module types associated with these rules).
8. Lockdown bypasses, including methods to prevent execution or trigger errors during execution.
9. Techniques to gain access to the original `globalThis`.
10. Methods to access and globally alter other module objects by reference.
11. Magic comments in `import`.

# 03 — Findings

Overall, we reported 5 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



| Severity | Count |
| --- | --- |
| CRITICAL | 0 |
| HIGH | 3 |
| MEDIUM | 0 |
| LOW | 0 |
| INFO | 2 |

# 04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

| ID | Severity | Status | Description |
|---|---|---|---|
| OS-LMT-ADV-00 | HIGH | RESOLVED | `ContextModules` are excluded from lavamoat runtime wrapper, allowing compromised modules to import restricted modules. |
| OS-LMT-ADV-01 | HIGH | RESOLVED | Webpack's `asset/resource` module type may be exploited by utilizing the `URL()` constructor to import malicious files. |
| OS-LMT-ADV-02 | HIGH | RESOLVED | `wrapRequireWithPolicy` does not validate whether the provided `specifier` is a string literal, allowing to exploit this by utilizing objects as specifiers to overwrite `toString`. |

## Package Enforcement Bypass via ContextModules HIGH OS-LMT-ADV-00

### Description

There is a package enforcement bypass issue related to `ContextModules`, a Webpack feature that handles imports that contain expressions. Since these modules are not wrapped by LavaMoat's runtime protection, they can bypass the security restrictions defined in LavaMoat's policy. Generated during Webpack bundling, `ContextModules` manage expression imports. Consequently, it allows a compromised module to request and load restricted modules that should otherwise be inaccessible, weakening the security model provided by LavaMoat.

### Proof of Concept

>_ *poc* JAVASCRIPT

```javascript
// node_modules/<compromised_module>/index.js
(async () => {
    const x = "index";

    const y = await import(`@ethereumjs/util/dist/${x}.js`);
    console.log(y);
})();
```

In the provided Proof-of-Concept, a compromised module imports an expression (which translates to `@ethereumjs/util/dist/index.js`) that is not permitted in its policy. The expression import mechanism bypasses LavaMoat's restrictions because the runtime enforcement is not applied to the generated function utilized for loading the module.

### Remediation

Ensure that `ContextModules` are wrapped to apply LavaMoat's runtime protection. Also, special care should be taken to handle both lazy loading (`import()`) and static loading (`require()`), taking into account that each one of them utilizes different `__webpack_require__` runtime functions.

### Patch

Fixed in 0d69202.

## Malicious Imports via Webpack Assets Module HIGH OS-LMT-ADV-01

### Description

It is possible to exploit Webpack's handling of assets when utilizing the `URL()` constructor to import files. Webpack treats certain imported assets, such as HTML files when included via the `URL()` constructor, as `asset/resource` types, which implies they are bundled into the final `dist/` directory of a web application or browser extension. Consequently, a malicious NPM package may be introduced into the project's dependency tree. In the below example, the package contains malicious HTML content, specifically utilizing the `URL()` constructor to import an HTML file ( `a.html` ).

>_ *poc* JAVASCRIPT

```
// node_modules/<compromised_module>/index.js
const x = new URL('./a.html', import.meta.url);

// node_modules/<compromised_module>/a.html
<html>
  <body>
    <script>alert(document.cookie)</script>
  </body>
</html>
```

Since `a.html` is considered an asset/resource module type, Webpack will copy it into the final bundle. Once bundled into the application, it becomes accessible through the web application or browser extension. In this case, opening the HTML file in a browser will execute the `<script>` tag containing `alert(document.cookie)`.

### Remediation

Filter the types of files that can be imported via the `URL()` constructor. Specifically, for modules designated as asset/resource, limit imports to only trusted file types.

### Patch

Fixed in PR#1451.

## Nodejs Package Policy Bypass via Builtins HIGH OS-LMT-ADV-02

### Description

There is a flaw in `wrapRequireWithPolicy`, where it fails to validate the type of the `specifier` argument passed to `__webpack_require__`. It does not validate if the specifier provided by the module is a string literal, so it is possible to exploit it utilizing objects as the specifier and overwriting `toString`. This implies that even if a module is restricted by policy from accessing certain built-ins, this exploit may be utilized to load the unauthorized module, compromising Lavamoat's security policy.

### Proof of Concept

>_ *poc.js* JAVASCRIPT

```javascript
let count = 0;
const toStringExploit = () => {
    count++;
    if (count == 1){
        return "crypto"
    }

    return "./node_modules/buffer/index.js"
}
const x = {toString: toStringExploit}
const r = __webpack_require__(x)

console.log(r)
```

The Proof-of-Concept given above illustrates this issue, simulating the bypass by compromising a package that has access to `crypto` builtin module and does not have access to the `buffer` module, by creating an object ( `x` ) with a custom method ( `toStringExploit` ), which returns the string: `"crypto"`, the first time it is called. Once the access to `crypto` is granted, the `toStringExploit` returns the path to the `buffer` module ( `"./node_modules/buffer/index.js"` ), which is not authorized, but it still gets loaded as a result of the bypass. This POC shows how it would be possible bypass the policy enforcement if the module has access to the crypto builtin module.

### Remediation

Ensure that the `specifier` is a string:

>_ *example.js* JAVASCRIPT

```javascript
const specifier = `${specifier}`
```

### Patch

Fixed in 501d14d.

# 05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

| ID | Description |
| --- | --- |
| OS-LMT-SUG-00 | The current approach to validating JavaScript source (relying on the integrity of the `sesCompatibleSource`), may be manipulated through vulnerabilities, allowing arbitrary code execution outside the sandbox. |
| OS-LMT-SUG-01 | It may be possible to utilize path traversal in Webpack's import magic comments to write chunk files outside the intended directory. |

## Inadequate Source Validation OS-LMT-SUG-00

### Description

`validateSource` checks the entire function (`before + sesCompatibleSource + after`). However, if `sesCompatibleSource` is manipulated, it may be possible to inject malicious code, which may break out of the sandbox if it is crafted in a specific way (such as closing `with()` statements or escaping contexts).

### Remediation

Validate `sesCompatibleSource` separately to ensure it contains only valid code and does not include malicious injections before it gets combined with the rest of the function. Additionally, enable the `runChecks` flag by default, ensuring that these validation checks are always performed.

## Path Traversal in Webpack Chunks OS-LMT-SUG-01

### Description

It is possible to misuse webpack's import magic comments, specifically the `webpackChunkName` comment, which allows defining a custom name for the chunk created for dynamically imported modules. If the chunk name contains a path traversal sequence, it may result in the chunk to be written to directories outside the intended `dist/` folder, such as parent directories, resulting in unintended file creation.

>_ *example.js* JAVASCRIPT

```
// node_modules/<compromised_module>/index.js
import(
    /* webpackChunkName: "../a" */
    './x.js'
);

// x.js must exist
);
```

In this example, the chunk `"../a"` would be generated and written outside the `dist/` directory. Although the content is wrapped with LavaMoat's runtime sandbox and ends with `.js`, this behavior still seems unintended.

### Remediation

Validate all chunk names during the build process to ensure they do not contain any path traversal sequences.

# A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

**CRITICAL**

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

**HIGH**

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

**MEDIUM**

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

**LOW**

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

**INFO**

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

# B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.