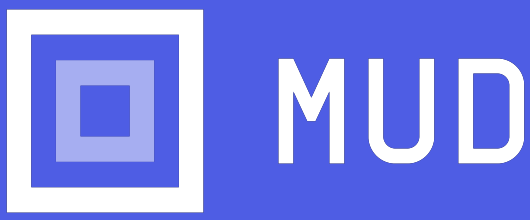
OpenZeppelin | security

MUD Code Generation Audit



February 9, 2024

Table of Contents

Summary

Type	Library
Timeline	From 2023-12-05 To 2023-12-22
Languages	TypeScript, Solidity

Total Issues	14 (7 resolved, 4 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	2 (1 resolved, 1 partially resolved)
Low Severity Issues	5 (1 resolved, 2 partially resolved)
Notes & Additional Information	7 (5 resolved, 1 partially resolved)

Scope

We audited the latticexyz/mud repository at commit f613359.

In scope were the following contracts:

```
packages
├── common/src/codegen
│   ├── render-solidity
│   │   ├── common.ts
│   │   ├── renderEnums.ts
│   │   └── renderTypeHelpers.ts
│   └── utils
│       ├── contractToInterface.ts
│       ├── extractUserTypes.ts
│       ├── format.ts
│       ├── formatAndWrite.ts
│       └── loadUserTypesFile.ts
├── world/ts/node/render-solidity
│   ├── renderSystemInterface.ts
│   ├── renderWorld.ts
│   └── worldgen.ts
└── store/ts/codegen
    ├── field.ts
    ├── record.ts
    ├── renderFieldLayout.ts
    ├── renderTable.ts
    ├── renderTableIndex.ts
    ├── renderTypesFromConfig.ts
    ├── tableOptions.ts
    ├── tablegen.ts
    ├── tightcoder
    │   ├── renderDecodeSlice.ts
    │   ├── renderEncodeArray.ts
    │   └── renderFunctions.ts
    └── userType.ts
```

System Overview

In our previous audit report, we described the `Store` and `World` components of the MUD system. In summary, they provide a low-level storage mechanism and a framework for additional standard functionality such as access control. A central feature of the system is the "table" abstraction which represents a structured way to interact with the storage. Developers would typically introduce app-specific business logic by designing and populating their own tables.

As a convenience, the Lattice team also provides utility functions for developers to convert a specification of the desired table structure into a corresponding collection of Solidity libraries. In fact, this is how the `World` tables are generated. This is the main purpose of the code under review. It also provides a mechanism to generate interfaces for the `World` and its associated systems.

Security Model and Trust Assumptions

Since the code under review simply generates convenience libraries, it does not introduce any new security assumptions. All access control and data integrity requirements should be enforced by the core `Store` and `World` contracts. As such, this audit primarily focuses on code quality and correctness. Specifically, we are only interested in scenarios where the table configuration is accurately specified. Although the code includes several consistency checks, it is still possible for invalid configurations to produce poorly specified libraries. This outcome is both reasonable and expected.

Medium Severity

M-01 Insufficient Testing

The codebase has several potential gaps in testing which may pose a risk to the robustness and security of the system. The following list is a collection of identified issues and proposed recommendations aimed at improving the overall quality of the testing suite:

- Overall, the testing coverage of the code under review is notably low. This can lead to various consequences such as maintainability problems, functionality issues, and security concerns. Consider thoroughly testing the codebase to enhance system maintainability and fortify security measures.
- Consider adding more tests to ensure a complete coverage of all possible branches. For example, `renderFieldLayout.ts` is one of the few audited files with corresponding tests. However, the conditions on lines 17, 18, and 22 are not covered by the test suite.
- Consider adding tests to check if the values in `constants.ts` are equivalent to the values in `constants.sol`, similar to the test in `storeEvents.test.ts` and `storeEventsAbi.test.ts`.
- In both `storeEvents.test.ts` and `storeEventsAbi.test.ts`, only the `helloStoreEvent` event is being tested. Consider adding more tests to cover all the existing events.
- The tests in `storeEvents.test.ts` and `storeEventsAbi.test.ts` are functionally equivalent but have been implemented with slight code variations. Consider merging logically identical tests into a single test in order to increase the maintainability of the test suite.

Update: *Partially resolved in pull request #2176. The Lattice Labs team stated:*

> *We have addressed the specific examples listed above and added some tests for various low-level helpers. For the higher-level stuff (rendering functions, etc.), we rely on the codegen output generated throughout the codebase (packages, examples, templates, etc.) and check into git to test/review the output we expect.*
>
> *We will plan to expand test coverage for specific edge cases as we find things that break or do not output what we expect. We are also planning a much deeper refactor of codegen. Before this, we may add a test suite that generates tables, etc. for a specific*

> *set of MUD config, and then use that to compare the refactor output to keep things compatible.*

M-02 Static Arrays Are Extendable

Static arrays are treated as dynamic for storage purposes and are cast to and from dynamic arrays as required. However, this means the corresponding dynamic field methods (`length`, `getItem`, `push`, `pop` and `update`) are rendered. Since `push` and `pop` change the array length, calling them could introduce unexpected behavior. In particular, when retrieving the contents of an array that is too short, an empty array is returned (and the remaining values are ignored). On the other hand, if the array is too long, it is truncated to the expected size.

Consider skipping the `push` and `pop` functions for static arrays. In addition, consider implementing a simpler `length` function that directly returns the known length. Lastly, in the interests of predictability, when converting dynamic arrays to static ones, consider reverting whenever the lengths are inconsistent.

Update: *Resolved in pull request #2175.*

Low Severity

L-01 Encapsulation Recommendation

There are several examples throughout the codebase where rendering functions make strong assumptions about how and when they are called. For example, the `renderDecodeDynamicFieldPartial` function assumes that the `_blob`, `_start` and `_end` variables exist, and that `SliceLib` will be available. While the example is intended to illustrate the claim, this pattern is a broad feature of the entire codebase, which is error-prone and makes local reasoning difficult.

The rest of this report includes suggestions for specific trivial simplifications, but we also believe that the codebase could benefit from a more structured approach. Our core recommendation is to make extensive use of TypeScript objects instead of strings to accumulate and synthesize business logic. The final rendering should focus entirely on describing the object in Solidity syntax.

For example, there could be a `SolidityFunction` object that individually tracks comments, input arguments, return values, local variables, visibility, etc. The arguments would also be objects that track type, location and name. Possible advantages include:

- The body of the function could reference specific named parameters or local variables. If the variable did not exist, it would raise an error.
- Functions could only invoke other functions if they exist in the higher `SolidityContract` object.
- Instead of using configurable callbacks to create similar functions, the object could simply be copied and modified to add new parameters or return values.
- The objects could use partial (or `Pick`) types to clearly indicate partially complete structures.

Consider restructuring the rendering code to focus on manipulating TypeScript objects instead of strings.

Update: *Acknowledged, not resolved. The Lattice Labs team stated:*

> *We are going to punt on this because we believe that an object-oriented approach may be meaningful for a standalone library designed for generating arbitrary solidity code. On the other hand, MUD codegen has a very narrow purpose and most of its functions are not meant for external use. Overly generalising it will complicate its development and maintenance with little benefit to the MUD codebase.*
>
> *A `SolidityFunction` object could be a full AST which requires unparsing it (using slang, which is in alpha, or writing our own unparser). This removes any context assumptions but adds a lot of code. This also does not create typescript compile-time typechecks. An object with some structured properties (like arguments, name, comment, etc.) and an unstructured body. The body has to allow many possible operations besides assignment.*
>
> *This slightly isolates context assumptions, but at the cost of replacing simple strings with complicated objects, and additional helpers that work on said objects. A `SolidityContract` object would only be meaningful for `renderTable`, which is redundant - it would verify imports and variables which are mostly static and verified by the Solidity compiler.*

L-02 Missing Function Comments

Most of the functions in the codebase do not have explanatory comments. Also, note that the `extractUserTypes` function is missing a `@param` statement for the `fromPath` parameter.

To improve the readability of the codebase, consider documenting all functions and their parameters.

Update: *Resolved in pull request #2185. The Lattice Labs team stated:*

> *We pulled out some return values into TS interfaces to make it easier to document params and added comment headers to the rest of the functions.*

L-03 Missing Import of Events, Structs, and Enums from Contract to ABI

The `renderSystemInterface` function automatically generates an interface from the corresponding contract. The function takes the imports, the contract name, functions, function prefix, and errors into account. However, it does not include events, structs, and enums. It is worth noting that since the interface is generated from the System file, the System cannot inherit its own interface. Thus, the duplicate definition does not lead to a conflict. One advantage of generating all the interfaces is that the `IWorld` interface will fail to compile if there are conflicting function signatures or errors across the different systems. This feature can be extended to the other structures as well.

Consider also including the events, structs, and enums to ensure the generation of a complete interface.

Update: *Partially resolved in pull request #2194. The Lattice Labs team stated:*

> *We played around with this but decided against these changes:*
>
> - *It does not make sense for us to copy over structs/enums into system interfaces because they end up being treated as different types in Solidity. It's best to define+import them from a common place outside of systems.*
> - *We do not want to encourage defining events in systems because they create side effects with potentially unexpected results depending on the context in which the system was called (`call` vs `delegatecall`). We did make a small adjustment to only include errors in interfaces when they are actually used (rather than including them if they are found in the file).*

L-04 Restricting Solidity Version

In `renderedSolidityHeader`, the Solidity version has been hardcoded to `>= 0.8.21`. This was chosen to match the manually created part of the codebase. However, this can cause restrictions when developing contracts on other EVM-compatible chains that often do not support certain opcodes (e.g., the `PUSH0` opcode that was introduced in Solidity version `0.8.20`).

Consider making the Solidity version a configurable parameter to increase the possible applicability of the codebase.

Update: *Acknowledged, not resolved. The Lattice Labs team stated:*

> *We are going to punt on this because we want to make use of some recent Solidity features and many of our contracts, libraries, etc. are meant to be used together as a whole (i.e., a framework) rather than in isolation.*
>
> *While it might be better to set each Solidity file to its minimum viable Solidity version, this is not something we have seen users ask for and probably does not make sense for us to maintain.*

L-05 TypeScript Inconsistency

The codebase makes extensive use of TypeScript types to validate consistency and improve code clarity. Here are some instances that could benefit from more consistent types:

- `renderCommonData` and `fieldPortionData` could return a named type instead of an arbitrary object.
- The argument type for `renderCommonData` and `renderTypeHelpers` could `Pick` from `RenderTableOptions`.
- `renderTightCoderDecode` and `renderTightCoderEncode` could `Pick` from `RenderType`.

Update: *Partially resolved in pull request #2121 and pull request #2185. The Lattice Labs team stated:*

> *`RenderTableOptions` is a store package concept and importing that into the common package for reuse would create a circular dependency and the codegen utils in the common package have no real knowledge of tables. Going to punt on this change.*

> *Also going to punt on notes for `renderCommonData` and `fieldPortionData` as they are more stylistic suggestions and do not seem entirely necessary. Will save any cleanup of this for a larger refactor of the codegen utils.*

Notes & Additional Information

N-01 Missing Explicit Function Visibility

The generated functions in `renderWrapperStaticArray` and `renderUnwrapperStaticArray` are implicitly using the default `public` visibility.

To clarify the intent and favor readability, consider explicitly declaring the visibility of the aforementioned functions.

Update: *Resolved. This is not an issue. The Lattice Labs team stated:*

> *This issue seems to be incorrect. These particular renderers are only ever used to render free functions which cannot have visibility (and they are not meant to be used outside of the table file). Perhaps an alternative issue could be to either render private library functions instead of free functions, or to make the use of these TypeScript renderers less ambiguous.*

N-02 Code Cleanup

We identified the following examples of code that can be simplified for better readability:

- In `record.ts`, the `if` statements on lines 210 to 227 and 279 to 291 contain repeated code in both branches. This code can be moved outside the conditional structure.
- In `formatAndWrite.ts`, lines 9 to 12 and lines 22 to 25 can be refactored into a single function.
- This `name` parameter could reuse the corresponding constant.
- Using an Immediately Invoked Function Expression to set `staticResourceData` seems unnecessary.
- The regular expression conformity checks present in these functions could use `test` (on the regex itself) instead of `match`. This is because the matched result is not used.

- This expression unnecessarily wraps a string inside another string.
- `Object.assign` modifies the target object. Hence, assigning the result to itself is unnecessary.
- `renderTableIndex` could use the `tableIdName` in `staticResourceData` instead of recomputing it.

Update: *Partially resolved in pull request #2110. The Lattice Labs team stated:*

> - *Punting on `record.ts` file changes as we think it is clearer to repeat the code snippets instead of trying to reduce repeated code (and it is unclear how to not repeat without increasing complexity).*
> - *Punting on DRY-ing `formatAndWrite.ts` as it feels unnecessary here and would require a later refactor if we want to introduce any Solidity-specific or TS-specific code paths.*

N-03 Imprecise Error Message

In the `encodeFieldLayout` function, both error messages on lines 17 and 18 render the same message without differentiating whether the dynamic fields or total fields caused the error.

Consider changing both error messages to show the exact reason for failure.

Update: *Resolved in pull request #2114. The Lattice Labs team stated:*

> *We found the same imprecise errors in Solidity so we improved those too.*

N-04 Naming Suggestions

Throughout the codebase, some functions and variables can benefit from being renamed:

- The `length` function should be renamed to `getLength`.
- The `renderEncodedLengths` function should be renamed to `renderEncodeLengths`.
- The `_internal` parameter does not describe its behavior and should be renamed to `_useExplicitFieldLayout` or something similar.
- The `renderWorld` function should be renamed to `renderWorldInterface`.

Update: *Resolved in pull request #2115. The Lattice Labs team stated:*

> *Going to punt on the `length` -> `getLength` suggestion because it would conflict with our pattern of `get{FieldName}`. For each array field, a corresponding `length` method is added (in addition to `get`, `set`, etc.).*

N-05 Bloated Codebase

Listed below are instances where files are generated upon configuration changes or the addition of system contracts but not removed when they become unused:

- When adding a new system contract, the contract's interface is extracted and written to the `worldgenBaseDirectory`. If the system contract is removed or the `worldgenBaseDirectory` changes, the interface remains.
- When adding tables to the configuration or when `outputBaseDirectory` changes, table files are generated and written to the `outputBaseDirectory`. In this case, only when a table is removed and `outputBaseDirectory` remains the same, the corresponding generated table file is also removed.
- When changing the `codegenIndexFilename` or `outputBaseDirectory` a new table index file is written without removing the old one.
- When changing the `userTypesFilename` or `outputBaseDirectory`, a new types file is written without removing the old one.

Consider clearing obsolete files so that the configuration and `systems` directory always match the current codebase.

Update: *Acknowledged, not resolved. The Lattice Labs team stated:*

> *We do not keep any sort of manifest to determine what files have been generated between usages of `tablegen` or `worldgen` (either run independently as regular functions or via `dev-contracts`, or other commands). Therefore, in the context of these functions, we have no "previous output directory" or "previous index filename" to know what files to clear.*
>
> *We work around this by:*
>
> 1. *Defaulting to putting codegen files into a consistent directory (i.e., `codegen`) so they can be removed with one command.*
> 2. *Having each template include a `clean` script in the `package.json` file for manual cleanup of this directory. We would like to move towards combining codegen functions/commands into a single util and consolidate some of this behavior and may wait for deeper changes (like a file manifest) until that time.*

N-06 Unused Variables

`methodNameSuffix` is defined and used in an object to convert a `RenderType` into a `RenderField`. However, it does not exist on either type and is otherwise unused.

Similarly, the following values are never used:

- The `UserTypeInfo` type
- The `_tableId` and `_keyArgs` return values from renderCommonData
- The `name` parameter of `StaticResourceData`
- The `worldContractName` parameter of `zWorldConfig`
- The `_untypedStore` parameter to the `renderWithStore` callback

Consider removing any unused variables from the codebase.

Update: *Resolved in pull request #2103. The Lattice Labs team stated:*

> *`StaticResourceData`'s `name` is used by `renderTableId`. `worldContractName` was in use but was an oversight when rewriting our deploy pipeline. We intend to add this back in and this is outside the scope of codegen anyway.*

N-07 Typographical Errors

The following typographical errors were identified in the codebase:

- dyanmic should be "dynamic".
- registed should be "register".
- system should be "world".

Consider fixing the aforementioned typographical errors in order to improve the readability of the codebase.

Update: *Resolved in pull request #2101.*

Conclusion

The code generation system provides a convenient mechanism to map a desired database configuration onto a corresponding collection of table libraries. This improves developer experience as the generated code ensures consistency and reliability of the interface, which allows developers to focus on the higher-level architectural design.

The main recommendations include improving the clarity and reliability of the generation code itself. This can be achieved by expanding the test suite and refactoring the codebase to use TypeScript objects instead of strings to structure the business logic. The Lattice team was highly responsive and helpful throughout the audit period.