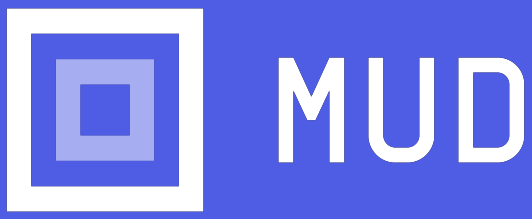
OpenZeppelin | security

MUD Audit



February 15, 2024

Table of Contents

Summary

Type	Library
Timeline	From 2023-10-03 To 2023-11-14
Languages	Solidity

Total Issues	32 (32 resolved)
Critical Severity Issues	2 (2 resolved)
High Severity Issues	0 (0 resolved)
Medium Severity Issues	6 (6 resolved)
Low Severity Issues	12 (12 resolved)
Notes & Additional Information	11 (11 resolved)
Client Reported Issues	1 (1 resolved)

Scope

We audited the latticexyz/mud repository at commit 12a9eb1.

In scope were the following contracts:

```
packages
├── store
│   └── src
│       ├── Bytes.sol
│       ├── FieldLayout.sol
│       ├── Hook.sol
│       ├── Memory.sol
│       ├── PackedCounter.sol
│       ├── ResourceId.sol
│       ├── Schema.sol
│       ├── Slice.sol
│       ├── Storage.sol
│       ├── StoreCore.sol
│       ├── StoreData.sol
│       ├── StoreHook.sol
│       ├── StoreRead.sol
│       ├── StoreSwitch.sol
│       ├── constants.sol
│       ├── leftMask.sol
│       ├── storeHookTypes.sol
│       ├── storeResourceTypes.sol
│       └── tightcoder
│           └── TightCoder.sol
└── world
    └── src
        ├── AccessControl.sol
        ├── Create2.sol
        ├── Create2Factory.sol
        ├── Delegation.sol
        ├── DelegationControl.sol
        ├── Module.sol
        ├── SystemCall.sol
        ├── SystemHook.sol
        ├── World.sol
        ├── WorldContext.sol
        ├── WorldFactory.sol
        ├── WorldResourceId.sol
        ├── constants.sol
        ├── modules
        │   └── core
        │       ├── CoreModule.sol
        │       ├── constants.sol
        │       └── implementations
```

```
│           ├── AccessManagementSystem.sol
│           ├── BalanceTransferSystem.sol
│           ├── BatchCallSystem.sol
│           ├── ModuleInstallationSystem.sol
│           ├── StoreRegistrationSystem.sol
│           └── WorldRegistrationSystem.sol
├── requireInterface.sol
├── revertWithBytes.sol
├── systemHookTypes.sol
├── version.sol
└── worldResourceTypes.sol
```

System Overview

The MUD system provides a cohesive range of standard and extensible functionality to quickly develop blockchain applications.

Store

The central component of the MUD system is the `Store`, which is a contract that behaves like a database. All persistent storage is presented as a set of tables, and the `Store` contract takes care of mapping this structure to the linear EVM storage layout. Importantly, the structure is not fixed at deployment time, which means developers can easily add new tables or any other functionality (described below) as their application evolves.

In addition, the `Store` contract allows multiple applications to provide different overlapping views of the same underlying data. This is achieved by standardizing the events that describe all state changes, so off-chain tooling can represent the storage in an application-agnostic way. In fact, some tables can be designated as *off-chain tables*, where they will emit the standard event stream on storage changes but the actual data will not be saved on-chain and must be reconstructed from these events.

Tables can also include hooks. These are arbitrary code snippets that can be executed whenever a record is added, removed or modified. This would typically be used for data validation, or to ensure consistency within a database record.

World

Although the `Store` contract provides the basic database functionality, it will typically be extended to provide higher-level abstractions for users and developers. One such extension is the `World` contract (in conjunction with the `CoreSystem` contract) which provides generic and flexible mechanisms for many standard features.

For example, the `World` contract registers tables in the `Store` contract to track access control relationships for all the other tables. Specifically, the database is segmented into different namespaces each of which has an owner address. The namespace owner can create new tables and manage which addresses can modify which tables in the namespace.

In addition, the namespace owner can register systems which are arbitrary contracts that are granted access to the namespace. These would typically be used for application-specific logic, to modify multiple records and tables in the same transaction, and to do so in a consistent way. They can even register system hooks (arbitrary code snippets) to run before or after the system is invoked. Moreover, there are some convenience functions which can be used to invoke systems more easily or to make multiple calls in a single transaction. The namespace owner also controls which addresses can invoke which systems.

The `World` contract also provides a delegation mechanism whereby any address can allow other addresses to perform actions on its behalf. If desired, the delegation can be managed by an arbitrary contract, so approval can be granted or rejected based on the details of the action and the particular delegatee. The namespace owner can also provide a fallback namespace delegation contract that manages all delegations for systems within the namespace.

The ETH held by the `World` contract is partitioned among the namespaces. This provides the systems in the namespace with a convenient way to share the same balance. Naturally, each individual system contract can also maintain a private ETH balance if that is desired.

Lastly, there is a root namespace which can be used to register privileged systems with complete control over the storage. In this way, the behavior of any deployed `World` contract is upgradeable and customizable.

Security Model and Trust Assumptions

Privileged Roles

The codebase is intended to be deployed by application developers. Depending on the use case, an application might require its own `World` or it could use an existing `World` instance, possibly with its own namespace for encapsulation. In either case, all `World`s are independent and the Lattice team does not retain any privileged access over them.

In the standard use case, the `World` deployer will initialize the system to become the owner of the root namespace, the world namespace, and the store namespace. They will then create the tables and records required to implement the features described above.

Anyone can register a new namespace and become its owner. This will let them manage the access control requirements (including transferring and renouncing ownership) and create new tables, systems, and hooks within the namespace.

However, it is worth noting that the root namespace owner can install new privileged systems or replace the `CoreSystem` functionality which implicitly gives them control over all the resources. Therefore, unless they renounce their power, they should be trusted by all the `World` users.

Possible Data Corruption

In the interest of efficiency, the system intentionally bypasses many data integrity checks. For example, users can set a table field to a value that is larger than what the type supports, which would overflow into subsequent fields, possibly beyond the allocated space. Similarly, users do not need to follow the expected key format when referencing records which could lead to them storing values at "invalid" pseudorandom locations. Users who have been granted write access to a table are implicitly trusted not to corrupt the data in this way.

It is worth noting that the system already relies on the assumption that pseudorandom locations (for all records in all tables in all namespaces) are sufficiently separated to prevent data collisions between different records. The ability for users to overwrite locations near their legitimate records, or at some other pseudorandom locations, does not undermine the basic assumption. Although users can corrupt their own tables, it should still be impossible to manipulate records in other tables and namespaces.

Critical Severity

C-01 Namespace Access Can Be Backdoored

The `WorldRegistrationSystem` contract takes care of registering namespaces, among other things. A namespace sets the context for a user to deploy systems and register tables. Ownership of or access to this namespace allows for the performance of critical operations such as setting `Store` values or transferring balance. Access to a namespace also gives access to all resource-specific systems or tables.

However, there is a problem when it comes to the arbitrary options of registering namespaces. For instance, consider the following attack. It assumes that the victim initially registers something for a new namespace (so the namespace does not yet exist):

1. The user attempts to register a system or a table, which would implicitly register the namespace.
2. The attacker front-runs the victim to:
    1. Register the namespace directly themselves
    2. Grant access to an account they control
    3. Transfer ownership of the namespace to the victim
3. The victim proceeds with (1). The call succeeds because while the namespace already exists, the victim is the legitimate owner.
4. The attacker has full control over anything that requires access such as balance transfer or `Store` writes.

Consider making the design more self-contained by removing the automatic namespace registration from the aforementioned functions. Instead, make it mandatory for a user to register the namespace beforehand such that a front-run would make the user's call fail due to the existence check.

Update: *Resolved in pull request #2007.*

C-02 Core System Can Be Disabled

The `CoreSystem` contract is registered in the `ROOT_NAMESPACE` in the `World` contract. This means that it is called using `delegatecall`.

As a pure implementation contract, there are no access control protections when its functions are invoked directly (bypassing the `World`). In particular, an attacker can register a system for the `CORE_SYSTEM_ID` and pass in any `_msgSender` value in the calldata to bypass the access control check. This is not an attack in itself because the contract storage of the `CoreSystem` contract is not relevant to the `World` contract. However, when the attacker invokes the `registerTable` function directly on the `CoreSystem`, it will make a `delegatecall` to the attacker-controlled core system, which could execute the `selfdestruct` opcode. This will remove the `CoreSystem`, effectively disabling it for the `World` contract.

Consider removing the namespace registration branch so that the `CoreSystem` will not execute a `delegatecall`. Alternatively (or in addition), consider protecting all `CoreSystem` functions with an `onlyProxy` modifier.

Update: *Resolved in pull request #2111, pull request #2007, and pull request #2180.*

Medium Severity

M-01 Incorrect Hook Parameter

Any storage hook that implements the `onBeforeSpliceDynamicData` function should receive the state of the record before the update. However, it will receive the updated encoded lengths instead, which may cause the hook to respond incorrectly.

Consider passing the original encoded lengths.

Update: *Resolved in pull request #2020.*

M-02 `requireInterface` Is Incorrectly Specified

The `requireInterface` function is incorrectly specified in two ways.

Firstly, it uses a `try-catch` block to revert with a meaningful error if the contract does not implement the expected interface. However, this will only catch errors that occur in the context of the target contract. If the target contract has no code, or returns a value that cannot be decoded as a boolean, the error will occur in the caller's context and will revert outside the `try-catch` block.

Secondly, it discards some requirements of the EIP specification. To be compliant, it should confirm that the `supportsInterface` function returns `false` for the invalid interface `0xffffffff`, and that it does not consume more than 30000 gas. This ensures that unrelated contracts that happen to have a fallback function that returns at least 32 bytes will not mistakenly pass the validation.

Consider using the OpenZeppelin `ERC165Checker` contract, or correcting these deviations.

Update: *Resolved in pull request #2016.*

M-03 Sliced Bytes Are Cut Off

In the `Bytes` library, the `slice4` function intends to return a `bytes4` value from a `bytes32` value at a given index. While the output is correctly returned as a `bytes4` value, the actual output variable is declared as a `bytes2` value. This leads to only two correct bytes being returned from the index position followed by two zero bytes. The wrong data could escalate into more severe security issues depending on the context.

Consider correcting the output variable type to `bytes4`.

Update: *Resolved in pull request #2031.*

M-04 Memory Corruption on Load From Storage

In the `Storage` library, the `load` function can load data from a storage location into a memory pointer location. The loaded data can further be specified by an offset in the storage and a length to enable loading data that spans over multiple slots.

However, there is an edge case in the parameter input set that causes memory corruption. This is when data is loaded from an offset position, but the length is less than the remainder of the slot:

```
storage slot: [ 0 ---------- 10 ---------- 20 ----------- 31 ]
                            ^ offset
                            <-- length -->
                            <------- word remainder ------->
```

The problem occurs because the bitmask is based on the word remainder instead of accounting for the length. This leads to extra bytes being written into memory (e.g., bytes 20 to 31 in the example above).

There are no significant consequences in the current version of the codebase. This is because all memory loads either use a zero offset or invoke the three-parameter version which reserves sufficient space to cover and ignore the unwanted bytes. Nevertheless, the library is intended to support external codebases and the inconsistency may lead to arbitrarily severe memory corruption.

Consider respecting the length when constructing the bitmask as seen in the `store` function.

Update: *Resolved in pull request #1978.*

M-05 `registerFunctionSelector` Can Be Front-Run and DoS'ed

The `registerFunctionSelector` function is used to register system function selectors for the `World` context. This unique `bytes4 worldFunctionSelector` is based on the user-provided `namespaceString`, `nameString`, and `systemFunctionSignature`. In the `fallback` function of the `World` contract, the `worldFunctionSelector` is taken as a look-up table value to forward the call to the right system and function. This mechanism exists in parallel to the `call` and `callFrom` functions that allow the user to specify the system to address through a function parameter.

The problem is that function selectors are only four bytes, so collisions (technically, second-preimages) can be realistically generated. This is particularly true if the `worldFunctionSelector` is known ahead of time (e.g., in the context of DAO voting). If a `worldFunctionSelector` is already taken, subsequent registrations will revert. This allows a malicious actor to brute-force a `worldFunctionSelector` that is identical to that of the victim and front-run the transaction. Hence, the legitimate victim transaction would fail.

Further, the `worldFunctionSignature` is constructed by concatenating the namespace, name, and system function signature with underscores. If the function name contains underscores itself, a selector collision can be trivially constructed. For example, the `worldFunctionSignature` of `MyNS_MySystem_do_things()` can be broken down into:

actor	namespace	name	function signature
victim	`MyNS`	`MySystem`	`do_things()`
attacker	`MyNS_MySystem`	`do`	`things()`

The impact of this attack is Denial of Service with the intention of griefing. The victim would have to re-write their code, redeploy it, and try to register it again. Alternatively, they will be unable to use this selector mapping feature.

Another collision that may occur is between registered `worldFunctionSignature`s and any of the external/public functions in the `World` contract. Since the external/public function is prioritized over the fallback function, the intended system call could lead to two problems:

- The call reverts due to a parameter decoding mismatch or access control check.
- The call succeeds and performs an unexpected action.

Consider changing the world function selector delimiter from an underscore to a character that is invalid for function names (e.g., a colon). Also, consider registering the `World`'s functions so that they can't be registered for the fallback function mechanism. However, the front-running issue will persist without a larger redesign. Hence, consider reducing the attack surface and code complexity by removing this feature entirely, or possibly restricting it to the root namespace and enforcing the usage of the `call` and `callFrom` functions that address systems specifically.

Update: *Resolved in pull request #2160, pull request #2169, and pull request #2182. The Lattice Labs team stated:*

> *The fixes address the potential conflicts between function selectors of different namespaces. It does not address the possibility of front-running namespace/function selector registration which we've decided to punt on for now and later address with an optional module. This module will allow "committing" to a hashed namespace and then "revealing/registering" the namespace in a second step. Since C-01 is fixed, this is not a security issue anymore but just a potential griefing vector.*
>
> *We have also decided to punt on addressing the possibility of brute-force calculating a conflicting function selector - since the value space is only 4 bytes it seems impossible to prevent this - but the feature is too useful to remove it altogether. The only issue this could lead to is a griefing attack if a function selector is known long enough before it is registered. This seems like a relatively small issue compared to the value custom function selectors provide in the vast majority of situations.*

M-06 Misleading Documentation

Throughout the codebase, there are multiple instances of misleading documentation:

- In the `loadField` function of the `Storage` library, the documentation suggests that the bytes beyond the `length` parameter are zero. However, considering a storage slot as `[ aa..aa bb..bb cc..cc ]` and `bb..bb` being the desired field, then despite its respective `length`, the result would return `cc..cc` as part of the bytes response. As such, the documentation is misleading and can cause consecutive memory corruption. Consider cleaning up the memory space by applying a bitmask or updating the documentation.
- In the `zero` function of the `Storage` library, the `length` parameter's documentation suggests it should be specified in words. However, in the code, the value is expected to be in bytes. This can lead to a shortcoming in overwrites to zero and therefore in preserving old data, which can lead to additional security issues. Furthermore, the length is rounded up to the nearest 32 bytes multiple which is not clear either. Consider clarifying the `length` documentation.
- The comments in the `Storage.load` function refer to the masking as "middle part" and "surrounding parts" whereas it is actually the left and right parts.
- The code comments for all the `pack` functions of the `PackedCounterLib` library read "Packs a single value into a PackedCounter", but most of them pack multiple values.
- The documentation of the `leftPaddingBits` parameter in the `TightCoder.encode` function is ambiguous. It says, "The number of bits to pad on the left for each element", but the code actually shifts the values by that count to the left. For instance, this means when encoding a `uint120` value, the `leftPaddingBits` count would be 136 as seen in the `EncodeArray` library.

Consider correcting the documentation to align with the code's behavior. This will help improve the clarity and readability of the codebase.

Update: *Resolved in pull request #2100.*

Low Severity

L-01 Missing Table Registration

This issue was independently identified and fixed by the Lattice team during the audit.

The Core module registers the tables it uses, but it excludes the `FunctionSignatures` table. This prevents the indexers from decoding the `FunctionSignatures` events. Consider registering this table as well.

Update: *Resolved in pull request #1841 at commit f96d8b3.*

L-02 Off-Chain Indexers Can Lose Track of On-Chain State

The `StoreCore` contract allows for the manipulation of static and dynamic data in the tables. This includes setting a record of table elements, splicing static and dynamic data, and deleting a record. Hooks before and after the respective action enable reactivity to these storage changes. Whenever one of these functions is called, an event is emitted to notify off-chain indexers about the latest storage changes. It is expected that the indexers can recreate the on-chain storage from these events.

A problem can arise due to the order of events and calls within any of the aforementioned functions which can cause an indexer to perceive a different state outcome than what happened on-chain. The flow for those functions is generally the following:

1. Emit an event.
2. Call the "before" hooks.
3. Do the storage change.
4. Call the "after" hooks.

However, a malicious or mistaken namespace owner can set up a hook that can reenter any of those functions to change the flow (simplified) to the following:

1. Emit event `A`.
2. Reenter through the "before" hook.
    1. Emit event `B`.
    2. Do storage change `B`.
3. Do storage change `A`.

As such, although event `B` was emitted last, storage change `A` is in fact true. For instance, this could cause major confusion when the same record is first deleted and then created which would be perceived in a reversed order off-chain.

Consider placing the event emissions together with the actual state change to prevent this type of confusion for off-chain indexers.

Update: *Resolved in pull request #2017.*

L-03 Namespace Balance Transfer Value Can Be Lost

The `BalanceTransferSystem` is used to send a namespace's accounted ETH value to either another namespace or to an address. However, balances can be sent to a non-existent namespace. This could occur due to human error and would likely lead to a loss of funds. Although the non-existent namespace could be registered afterwards, any other user could front-run this registration to get control over the funds.

Consider checking whether the recipient namespace exists before proceeding with the transaction.

Update: *Resolved in pull request #2095.*

L-04 Delegation Can Be Misconfigured

The `WorldRegistrationSystem` allows users to register a delegation. This will enable the delegatee to make a call on behalf of the delegator, provided it meets the delegator's criteria.

When setting up delegation control with a system, it is possible to register an invalid delegation control without it being immediately noticeable. This would be the case when `initCallData` is not required (has length zero), which causes the registration function to skip the interface conformance check of the delegation control system. An attacker can take advantage of this mistake, if the delegation is not account-specific, by registering this non-existent system ID to thereby gain control over the system.

Consider performing the interface checks on the delegation control system in all cases, whether or not `initCallData` is required. It should be noted that this would prevent clearing the delegation (i.e., resetting the `delegationControlId` to zero). Thus, consider introducing a new function to unregister delegations.

Update: *Resolved in pull request #2096.*

L-05 Deployment Edge Case

The `World` contract's `initialize` function sets up the `CoreSystem` functionality. To make it only callable once, it checks if there is no existing `"core"` module installed.

However, the core module is installed under the name that the module responds with. If it responds with a different name, `initialize` will still be callable. This should not happen if the `World` creator is honest. A malicious owner can utilize this function as a backdoor through the delegate call. As the `initialize` function's access is bound to the creator role, this attack vector is active despite potentially revoking any root namespace ownership. Also, it allows overwriting any other record in the table of installed modules.

It is worth noting that the initial validation reads a record from the `InstalledModules` table before it is registered, and the subsequent update assumes the core module registered the table. This still works whether or not the core module actually registers the `InstalledModules` table because both operations directly access the contract storage location where the relevant table record *would be* stored. Nevertheless, it undermines the expected table abstraction.

Consider explicitly installing the core module under the `CORE_MODULE_NAME` to ensure the `initialize` function is single-use. Alternatively, to avoid accessing an unregistered table, consider introducing a new initialization flag.

Update: *Resolved in pull request #2170. There is a new `CoreModuleAddress` table to save the address.*

L-06 Incorrect ERC-165 Interface

The following interface IDs include the `ERC165_INTERFACE_ID` constant:

- `STORE_HOOK_INTERFACE_ID`
- `MODULE_INTERFACE_ID`
- `SYSTEM_HOOK_INTERFACE_ID`
- `WORLD_CONTEXT_CONSUMER_INTERFACE_ID`

This is inconsistent with the EIP (as can be seen by the Simpson example). Consider excluding the `ERC165_INTERFACE_ID` constant from the custom interface IDs.

Update: *Resolved in pull request #2014.*

L-07 Incomplete Table Validation

There are several consistency checks when registering a new table, but they are incomplete. In particular:

- The field layout validation does not confirm whether the total static field length is the sum of all the individual static field lengths, or that the unused length fields (if the maximum number of fields is not reached) are all zero.
- The schema validations for the key and value schemas do not confirm whether the total static field length is consistent with the static schema types.
- The field layout and value schema are not confirmed to be consistent with each other, except for having the same total number of fields. This means that the number of static fields is not necessarily the same, and the length of a static field in the field layout does not necessarily match the type of the static field in the value schema.

Inconsistent table specifications may interfere with saving, retrieving and interpreting the database records. Consider including these additional validations.

Update: *Resolved in pull request #2046.*

L-08 Incomplete Module Access Control

The `ModuleInstallationSystem` allows anyone to install any module. Since this makes an external call to the module, it does not provide any additional functionality. Instead, it is simply a convenient way to execute several related operations atomically. However, it can overwrite any record in the `InstalledModules` table, which doesn't have any subsequent effects in the current codebase.

Consider whether the `InstalledModules` table should use the module address (along with the arguments hash) instead of the module name as the record key. This is so that it identifies the action that was taken more directly and is more resistant to being overwritten. Alternatively, consider whether the `InstalledModules` table should be removed entirely.

Update: *Resolved in pull request #2168.*

L-09 Incomplete Resource ID Validations

Resource IDs in the `World` are expected to encode three components (the type, namespace and name) that are used to ensure consistency between different table records. However, there are several instances of incomplete consistency checks:

- It is possible to call `transferOwnership` on any resource ID, provided the caller owns the corresponding namespace. This includes a table, system, an unknown resource ID, or an incorrectly specified resource ID.
- It is possible to call `transferBalanceToNamespace` with a `toNamespaceId` that has a non-zero "name" component. This will effectively transfer the funds to a new account in the destination namespace, which needs to be spent individually (i.e., it cannot be consumed by systems within the namespace unless they are explicitly designed to handle it).
- It is possible to call `registerFunctionSelector` with any resource ID, provided the caller owns the corresponding namespace. In practice, if it does not correspond to a valid system, it will be unusable.
- It is possible to `grantAccess` to an incorrectly specified resource ID, or a resource that does not exist.
- It is possible to call `registerNamespace` or `registerNamespaceDelegation` with a `namespaceId` that has a non-zero "name" component.
- It is possible to call `registerSystemHook` with any resource ID, provided the caller owns the corresponding namespace.
- It is possible to call `registerStoreHook` with a non-existent `tableId`, provided the caller owns the corresponding namespace.

In the interest of predictability and limiting the attack surface, consider validating all three components of user-provided resource IDs and validating the existence of a resource wherever relevant throughout the codebase.

Update: *Resolved in pull request #2142 and pull request #2195.*

L-10 Inexplicit Revert

In the `getDynamicFieldSlice` function of the `StoreCore` library, a bytes range specified by `start` and `end` can be read from a dynamic field.

While the range is checked to be within the field's length, the order of `start` and `end` is not checked. If `start` were to be greater than `end`, the subtraction for the length would

underflow and the transaction would revert implicitly without providing any contextual information.

Consider adding a range check to fail more explicitly.

Update: *Resolved in pull request #2034.*

L-11 World Resource ID ROOT String Has Unexpected Length

The `WorldResourceIdInstance` library has helper functions to get information from an encoded `World` resource ID such as type, namespace and name. These fields can also be returned as a string using the `toString` function.

However, while the resource namespace is expected to occupy 14 bytes, the `ROOT_NAMESPACE_STRING` is actually a `bytes16` value. The string output could, therefore, have an unexpected length.

Consider changing the constant's type from `bytes16` to `bytes14`.

Update: *Resolved in pull request #1976.*

L-12 Override Removes Supported Interface

The `Module` contract overrides the `supportsInterface` function and no longer supports the `WORLD_CONTEXT_CONSUMER_INTERFACE_ID`. This is inconsistent with the equivalent `DelegationControl` override, which retains support for the `WORLD_CONTEXT_CONSUMER_INTERFACE_ID`.

Consider ensuring ERC-165 support for all implemented interfaces.

Update: *Resolved in pull request #2032.*

Notes & Additional Information

N-01 Encapsulate Functionality Recommendation

Several functions in the codebase have side effects or handle multiple use cases:

- The `AccessManagementSystem` does not provide a mechanism to renounce the ownership of a namespace, so users will need to transfer ownership to an uncontrolled address (like the zero address). However, this will create an unnecessary record in the `ResourceAccess` table.
- The `registerTable` function might register a namespace as well.
- The `registerSystem` function might register a namespace or delete a system. In fact, this is the only way to delete a system using the `World` functionality.
- The `registerSystem` function also accepts an existing system and system ID pair, which will delete the system records only to recreate them again, possibly changing the `publicAccess` flag in the process.
- Individual delegations must be removed by overwriting them using the regular registration function, which still leaves a stray record in the `UserDelegationControl` table.
- Namespace delegations cannot be set to zero so they must be removed by creating a new delegation control that rejects all calls.

The stray records could potentially be addressed by directly updating the tables, but this is fragile and undermines the abstractions provided by the `World` contract. Similarly, functions with side effects or multiple use cases can be fragile, and the most significant issues in this report are a result of imprecise handling at the boundaries of different functionality.

To reduce the attack surface and increase predictability, consider limiting each function to a single use case with a linear code path where possible.

Update: *Resolved in pull request #2157.*

N-02 Unintuitive Order of Function Arguments

The `Storage` library contains functions to `store` and `load` data according to the storage pointer, offset, length, and memory pointer. However, both functions take these arguments in different orders which can be non-intuitive for a developer.

Consider aligning the order of function parameters when the set of arguments overlaps.

Update: *Resolved in pull request #2033.*

N-03 Naming Suggestions

Throughout the codebase, there are code element names that are ambiguous:

- The `_staticFields` parameter of the `encode` function could be called `_staticFieldLengths`.
- The `requireNoCallback` modifier could be called `prohibitDirectCallback`.
- The `byteCode` parameter in the `deploy` function could be called `creationCode` as referred to in the documentation.
- The `tableWithHooks` parameter of the `filterListByAddress` function could be called `elementWithHooks` to account for the possibility that it refers to a system.

Update: *Resolved in pull request #2091.*

N-04 Unused Functions and Variables

Throughout the codebase there are multiple instances of unused functions and variables:

- In the `Bytes` library, the functions `toBytes32`, `equals`, `setBytes1`, `setBytes2`, `setBytes4`, `setBytes5`, `setBytes7`, as well as some of the `slice` functions
- The `NAME_BITS`, `TYPE_MASK`, `RESOURCE_MODULE` and `MASK_PTR` constants

Consider removing this code to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #2090.*

N-05 Unused Imports

Throughout the codebase, the following imports are unused and could be removed:

- Import `STORE_VERSION` of `StoreRead.sol`
- Import `SliceLib` of `TightCoder.sol`
- Import `FunctionSelectors` of `SystemCall.sol`
- Import `IStore`, `Schema`, `System`, `ROOT_NAMESPACE`, `ROOT_NAME`, `revertWithBytes`, `NamespaceOwner`, `IDelegationControl`, `Systems`, and `SystemHooks` of `World.sol`
- Import `ROOT_NAMESPACE`, `IBaseWorld`, `ResourceId`, `WorldResourceIdLib`, `WorldResourceIdInstance`, `RESOURCE_SYSTEM`, `AccessManagementSystem`, `BalanceTransferSystem`, `BatchCallSystem`, `ModuleInstallationSystem`, and `StoreRegistrationSystem` of `CoreModule.sol`
- Import `IModule`, `WorldResourceIdLib`, and `InstalledModules` of `AccessManagementSystem.sol`
- Import `WorldResourceIdLib` of `BalanceTransferSystem.sol`
- Import `AccessControl` and `ResourceAccess` of `ModuleInstallationSystem.sol`
- Import `ROOT_NAMESPACE`, `WorldContextProviderLib`, `NamespaceOwner`, `ResourceAccess`, `SystemHooks`, `SystemRegistry`, and `FunctionSelectors` of `StoreRegistrationSystem.sol`
- Import `IDelegationControl` of `WorldRegistrationSystem.sol`
- Import `DecodeSlice` of `Slice.sol`

Consider removing unused imports to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #2028.*

N-06 Duplicate Imports

There is a duplicate import of the `WorldContextProviderLib` library in the `SystemCall.sol` file.

Consider removing duplicate imports to improve the readability of the codebase.

Update: *Resolved in pull request #2024.*

N-07 Visibility Not Explicitly Declared

There are some instances of missing explicitly declared visibility:

- The `coreSystem` state variable
- The `_toBool` function
- The `requireInterface` function

Consider always explicitly declaring the visibility of variables and functions, even when the default visibility matches the intended visibility.

Update: *Resolved in pull request #2029.*

N-08 Code Simplification Suggestions

Throughout the codebase, there are multiple instances where the code could be simplified:

- The schema type validation loop in the `Schema` library can be simplified. Consider looping over the `_numStaticFields` schema types first to ensure they all have a non-zero static byte length. Then, loop over the `_numDynamicFields` schema types to ensure they have a zero static byte length. This way, it is not necessary to count the fields (and validate the count later) or branch inside the loop.
- The `Memory.copy` function is redundant to the identity precompile. Consider replacing the function's logic with a call to the precompile for simplicity and efficiency.
- The `leftMask` function is constructed by first creating a right mask (of size `32 - byteLength`) and then inverting it. However, the actual usage involves using both left and right masks [1, 2, 3, 4, 5]. Hence, the left mask is inverted back to a right mask. Consider returning a right mask in the first place.
- In contrast to, for instance, `FieldLayout`, `PackedCounter` and `Schema`, the `ResourceId` does not have a `using` statement in the `ResourceId.sol` file. Instead, it is duplicated several times throughout the codebase.
- There is an inconsistency for the integer base in assembly blocks. While most of the codebase uses hexadecimal numbers, there are instances [1, 2, 3, 4, 5] where decimal numbers are used.
- The `Slice` library uses `pointer(self)` first and `self.pointer()` later.
- For EIP-165 support, the codebase maintains constants [1, 2, 3, 4, 5, 6, 7] that reflect the supported interface IDs which are checked against in the `supportsInterface` functions. Instead, consider making use of the interface ID provided by the interface type like `type(Interface).interfaceId`.

- The `FieldLayout.numFields` function could make use of the `numStaticFields` and `numDynamicFields` functions instead of reimplementing them.
- In the `Slice.sol` and `Schema.sol` files, there is an inconsistency when it comes to the leading underscores for stack variables.
- There is an inconsistency of relying on default initialization to zero [1, 2] versus explicitly initializing to zero [1].
- In the `StoreCore.getFieldLayout` function, instead of getting the field layout through `Storage.loadField`, consider using `Tables._getFieldLayout`, similar to how it is done for the key schema and value schema.
- In `WorldContextConsumer._world`, the call to `StoreSwitch` could invoke `WorldContextConsumerLib._world` for consistency with the rest of the contract.
- The `coreModule` state variable of the `WorldFactory` is set in the constructor and never changed. Consider making it `immutable`.
- The `WorldRegistrationSystem.registerSystem` function accepts a `WorldContextConsumer` type argument, whereas it should accept a `System` type argument.
- The `slice` functions in the `Bytes` library can be simplified by declaring the `output` variable as a named return to save on the extra declaration and return in the function body.
- The validation condition in the `getSubslice` function could be replaced with `start > end || end > data.length`.
- The `extcodesize` opcode is not necessary to check whether the `Create2.deploy` call was successful since the address will also be zero in that case.

Consider applying the code changes as outlined above to improve the overall clarity and readability of the codebase.

Update: *Resolved in pull request #2140. The Lattice Labs team stated:*

> *We chose not to implement three suggestions due to gas use and readability concerns.*

N-09 Magic Number

The codebase defines the `BYTES_TO_BITS` constant, but then fails to use it in several places [1, 2, 3, 4, 5]. Consider using the constant wherever it applies.

Update: *Resolved in pull request #2015.*

N-10 Typographical Errors

The following typographical errors were identified in the codebase:

- "Since the they're max 32 bytes" has an extra "the"
- "@param e The length of the fourth dynamic field's data" should say "[...] fifth dynamic field's data"
- The constant `DYNMAIC_DATA_SLOT` should be called `DYNAMIC_DATA_SLOT`
- "A namespace can includes tables and systems" should say "include"
- "Require the balance balance to be greater or equal to the amount to transfer" says "balance" twice

Consider fixing all instances of typographical errors throughout the codebase.

Update: *Resolved in pull request #2023.*

N-11 Unnecessary Use of Generic Function

There are multiple cases [1, 2, 3] where the full record in the `Systems` table is retrieved but only the system address is retained. Consider using the more specialized `_getSystem` function instead.

Update: *Resolved in pull request #2022.*

Client Reported

CR-01 Store Namespace Unregistered

When the `CoreModule` is installed, the *root* namespace and *world* namespace are both registered with the creator as the owner. However, the *store* namespace is not registered. If another user registers it later, they would control the *store* tables (i.e., Tables, StoreHooks and ResourceIds), which essentially gives them control over the entire `World` contract.

Update: *Resolved in pull request #1712 at commit c14d140.*

Conclusion

The MUD system is a comprehensive framework for rapidly developing blockchain applications. It revolves around a central component called the `Store`, which acts as a flexible database contract allowing developers to dynamically add tables and functionalities. The `World` extends this functionality, enabling higher-level abstractions, access control, system registration, and delegation mechanisms, providing a customizable and upgradable structure for blockchain applications.

Throughout the audit period, the Lattice team was very responsive and provided us with useful explanations and clarifications. We also appreciated the code's organization and documentation. As noted in the report, our main suggestions involved limiting the number of branching code paths to limit the attack surface. The Lattice team implemented all suggested improvements. Overall, we believe the system is well-designed and implemented.