OpenZeppelin | security

Lombard Audit

LOMBARD

December 13, 2024

Table of Contents

Summary

Type	DeFi
Timeline	From 2024-11-04 To 2024-12-06
Languages	Golang, Solidity

Total Issues	39 (32 resolved, 3 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	8 (6 resolved, 2 partially resolved)
Medium Severity Issues	5 (4 resolved)
Low Severity Issues	13 (11 resolved)
Notes & Additional Information	13 (11 resolved, 1 partially resolved)
Client Reported Issues	0 (0 resolved)

Scope

We audited the `lombard-finance/ledger` repository at commit 282b484 and the `lombard-finance/smart-contracts` repository at commit 5622904.

In the `lombard-finance/ledger` scope were the following files:

```
cmd/ledgerd/main.go
cmd/ledgerd/cmd/commands.go
cmd/ledgerd/cmd/config.go
cmd/ledgerd/cmd/root.go
notaryd/
proto/
utils/
x/deposit/
x/notary/
```

In the `lombard-finance/smart-contracts` scope were the following files:

```
contracts/LBTC/ILBTC.sol
contracts/LBTC/LBTC.sol
contracts/bridge/oft/EfficientRateLimitedOFTAdapter.sol
contracts/bridge/oft/EfficientRateLimiter.sol
contracts/bridge/oft/LBTCBurnMintOFTAdapter.sol
contracts/bridge/oft/LBTCOFTAdapter.sol
contracts/consortium/Consortium.sol
contracts/consortium/ILombardTimelockController.sol
contracts/consortium/INotaryConsortium.sol
contracts/consortium/LombardTimeLock.solo
contracts/libs/Actions.sol
contracts/libs/BitcoinUtils.sol
contracts/libs/EIP1271SignatureUtils.sol
contracts/libs/FeeUtils.sol
contracts/libs/IProxyAdmin.sol
contracts/libs/RateLimits.sol
```

System Overview

Lombard bridges the gap between Bitcoin staking and DeFi by enabling BTC holders to provide economic security to DeFi protocols. At its core, the protocol allows users to stake Bitcoin through Babylon and receive LBTC, which can then be used to provide liquidity across multiple blockchain networks.

Babylon is a protocol that extends Bitcoin's utility by allowing it to secure Proof-of-Stake networks and applications. While Babylon enables Bitcoin staking, it requires locking up the BTC, which limits its broader utility. Lombard addresses this limitation through LBTC, creating a liquid representation of Babylon staked positions that remains active in DeFi markets.

When users interact with Lombard, they deposit Bitcoin into Lombard addresses, which is then staked through Babylon to generate rewards. Users receive LBTC tokens on their chosen blockchain at a 1:1 ratio with their deposited BTC. These tokens can be used in DeFi applications while continuing to earn staking rewards.

Users can later redeem their LBTC for BTC by burning the tokens, which triggers the release of the original BTC deposit from the Lombard address. They can also transfer LBTC between supported chains through either a canonical bridge or, on certain chains, through the Layer Zero Omnichain ERC-20 bridge.

Key Components

1. **Ledger**: The core module built using the Cosmos SDK that manages the protocol's state and operations. It handles:

- BTC deposit tracking and verification
- BTC withdrawals
- Staking position management with Babylon
- Cross-chain LBTC minting and bridging

- Consortium member management and consensus

1. **Smart Contracts**: A set of interconnected contracts deployed on supported chains that handle LBTC operations:

- **LBTC Token Contract**: An ERC-20 implementation with additional features for minting and burning
- **OFT Bridge Contracts**: A bridge that allows cross-chain transfers through the use of the Layer Zero Omnichain ERC-20 bridge
- **Consortium Contract**: A contract that validates consortium-signed operations and maintains an up-to-date list of consortium members

Security Model and Trust Assumptions

Lombard's security architecture rests on several foundational assumptions and risk mitigations:

1. **Bitcoin Network Security**: The protocol builds upon Bitcoin's underlying security framework for transaction validation and finality confirmation.

2. **Babylon Protocol**: The system inherently trusts Babylon's implementation for:

- Correct handling of BTC staking operations
- Proper distribution of staking rewards
- Secure validator selection and rotation

1. **Lombard Consortium**: The distributed nature of the Consortium provides resilience against individual node failures, requiring more than 1/3 of members to be compromised before security is affected. However, several critical security considerations exist:

- Currently operates as a permissioned system without economic security
- There are no slashing or penalty mechanisms for malicious behavior
- Lacks economic incentives for proper operation

It is crucial for the protocol that future updates address these considerations and implement economic security mechanisms, by leveraging Babylon's economic security or other mechanisms like LBTC.

Privileged Roles

The protocol's governance structure includes several key roles with specific responsibilities:

1. **Consortium Members**:

   Serve as the primary validators and operators of the protocol. They oversee and validate all critical protocol operations through participation in threshold signature schemes for transaction notarization. These members must operate under a 2/3 majority consensus requirement for all significant actions. Their coordinated efforts ensure the secure operation of cross-chain transactions and state management.

2. **Smart Contract Owners**:

Have the ability to upgrade the protocol's Smart Contracts. This role manages contract upgrades through a timelock mechanism that enforces a mandatory one-hour delay on all upgrades. The scope of owner actions should be strictly limited to implementing emergency responses and executing planned enhancements. The timelock mechanism provides transparency and prevents immediate changes while still allowing for potential intervention in case of malicious upgrades.

High Severity

H-01 Lack of Proper Checks for Sanctioned Addresses

In order to adhere to legal systems, it is important that the protocol is not used by sanctioned addresses. As such, the validators running `notaryd` should only sign payloads that interact with non-sanctioned addresses. When processing a notarization, the validator checks whether they should sign the notary session by using the `Verify` function for the corresponding strategy.

In the deposit strategy, all the Bitcoin input addresses are checked to make sure that they are not sanctioned, along with the output EVM address. However, for the deposit bridge strategy, only the recipient address is checked and the input address remains unchecked. In addition, for the unstake strategy, no sanction checks are performed. This oversight allows sanctioned addresses to use the protocol for bridging as well as unstaking, which may violate legal requirements.

Consider checking for sanctioned addresses for both inputs and outputs for the deposit bridge strategy, unstake strategy, and any future strategies that involve moving value across chains.

Update: *Resolved in pull request #95 at commit 0aaf405.*

H-02 User Unstakes Multiple Times But Is Only Paid Once

The purpose of unstake execution is to notarize the fact that Bitcoin was indeed paid back to the user after the user unstaked LBTC from an EVM chain. In order for the validators running `notaryd` to check that this occurred, they use the `Verify` function in `unstake_execution_strategy.go`. This method takes a previously submitted unstake execution payload, checks that the ID is tied to a previously submitted unstake payload, fetches the payment transaction from a Bitcoin node, and checks whether that transaction was made to the intended ScriptPubKey for the correct amount. If these checks pass, the validator will sign the payload.

However, there is no mechanism to verify whether this particular Bitcoin transaction has already been used. Therefore, if a user unstakes LBTC for the same amount (e.g., 2 LBTC) to the same Bitcoin ScriptPubKey two or more times, a single Bitcoin transaction of 2 BTC can be used to satisfy all of these payloads, and thus the user will not receive the intended 4 BTC total amount. Since anyone can submit a payload, it is easy for an attacker to monitor the chain for two unstake payloads with the same amount to the same Bitcoin ScriptPubKey and then submit an unstake execution payload causing the user to lose funds.

Before signing an unstake execution payload, consider validating that a Bitcoin payment transaction has not been used for a previous payment.

Update: *Partially resolved in pull request #110 at commits 9a1ecd9 and 4eb10a8. The current fix stores Bitcoin transaction IDs and output indexes exclusively in memory, resulting in their loss during a node reboot or when a new node is added to the network. The Lombard team stated:*

> *The check is currently implemented in memory only, so history is cleared on node reboot, so issue could be still potentially possible if the two different unstake has a long period of time in the middle such that a majority of notaryd instances received a reboot.*

H-03 Unbounded Subscriber Buffers May Lead to Denial of Service

In the `pubsub` package, the Bus implementation creates unbounded buffers for each subscriber by utilizing `chanx.NewUnboundedChan`. If messages are produced faster than subscribers can consume them, these buffers may grow indefinitely, leading to excessive memory usage and potential system instability or denial of service. This would happen especially with a large amount of slow consumers. An attacker could exploit this by introducing slow or unresponsive subscribers, deliberately causing resource exhaustion.

Consider implementing bounded buffers for subscribers and introducing a backpressure mechanism or strategy to handle buffer overflow, such as dropping messages or disconnecting slow subscribers.

Update: *Resolved in pull request #98 at commit ac4ee9e.*

H-04 Event Errors Are Silently Ignored

In the `events` package, the `handleErrors` function disregards errors received from the `eventErrs` channel by assigning them to the blank identifier `_` without doing any handling. These errors could be caused by important causes such as websocket disconnections, malformed events, or network timeouts. As such, ignoring these errors may prevent the system from detecting and addressing significant problems, potentially leading to degraded performance, unhandled exceptions, or system instability. Because these errors are not even logged, this would also leave developers unaware of failures and without the necessary information to diagnose or address them.

Consider properly handling the errors from the `eventErrs` channel by implementing appropriate responses based on the error type, such as attempting reconnections, logging the errors, or propagating them to an error channel for further action.

Update: *Partially resolved in pull request #78 at commits 38ba8f4 and 7ea5745. The current fix logs the error but it does not propagate or handle it.*

H-05 Locked Mutex During Subscriber Sends May Cause Deadlocks

In `pubsub` package, the bus `run` function locks its mutex `b.subMutex` while sending items to the subscribers' channels. Since sending to a channel is a blocking operation, if a subscriber is slow or its buffer is full, keeping the mutex locked during this operation can lead to deadlocks or severe performance degradation. While the mutex is locked, no other subscribers can receive events, and operations like adding or removing subscribers are blocked. This design would force all subscribers to wait for the slowest one, significantly limiting the bus throughput and potentially causing the system to freeze if a subscriber becomes unresponsive.

Consider modifying the implementation to avoid having the mutex locked during potentially blocking operations. This can be done by unlocking the mutex before sending the items to the subscribers.

Update: *Resolved in pull request #98 at commit ac4ee9e.*

H-06 Unrestricted Creation of Notary Sessions Can Allow for Voting on Expired Sessions

In the `x/notary` module, the session expiration mechanism has been constrained to process a maximum of 200 notary sessions per block. However, there is no corresponding limit on the number of notary sessions that can be created within a single block. This lack of a restriction allows for the generation of more than 200 notary sessions per block (either by accident or by a malicious entity), resulting in a backlog where sessions that should expire are not processed. If this happens, it will become possible to invoke `SubmitSignature` on sessions that should have been expired, because only the current state of the session is being checked.

Consider limiting the number of notary sessions that can be created per block. Additionally, in the future, consider incorporating deterrent mechanisms (e.g., slashing) to penalize participants who attempt to spam the creation of notary sessions.

Update: *Resolved in pull request #83 at commit 8c2e5ff.*

H-07 Topic Mismatch Between Smart Contract and Node

When performing a bridge deposit, which is done to burn LBTC on one EVM chain and minting it on another EVM chain, the `deposit` function in `Bridge.sol` is called, which results in a `DepositToBridge` event being emitted. The `DepositToBridge` event is defined as having the parameters of type `(address, bytes32, bytes32, bytes)` and thus the first topic of this event will be `keccak256(DepositToBridge(address,bytes32,bytes32,bytes))`, which is `0x3dd5691d...`.

After a payload is submitted to the Lombard Ledger, the nodes running `notaryd` will check their validity. Particularly, for the bridge deposit type payload, the node will check the information from the provided transaction hash with the EVM-compatible node to get the log. The events from this transaction hash will be obtained and compared to the `DepositToBridgeTopic`. The `DepositToBridgeTopic` is defined as `0xf42846b1...` and thus the check will fail and the node will not sign the payload. As a result, the bridge deposit will not be notarized and funds cannot move from one bridge to another.

Consider updating the `DepositToBridgeTopic` to reflect the topic of the event that is actually emitted from the smart contract.

Update: *Resolved in pull request #68 at commit f6c35d0.*

H-08 Events Only Read from One Smart Contract Address

After a payload for a bridge deposit or an unstake request is submitted, the nodes running `notaryd` will validate that the events were actually emitted from the smart contracts before signing the payload. For a bridge deposit, this is done using a call to the `GetDepositBridge` function, which as its third parameter takes in `resp.Params.LbtcAddresses` as the contract address where the event should come from. Similarly, for the unstake payload, a call to the `GetUnstake` function is performed which as its third parameter takes in `resp.Params.LbtcAddresses.Bytes()`.

In both of the these types of payloads, the value of `resp.params.LbtcAddresses` comes from the `deposit` module of the Lombard Ledger. The value `LbtcAddresses` only represents one smart contract address.

However, the smart contract that emits the `DepositToBridge` event is `Bridge.sol` and the smart contract that emits the `UnstakeRequest` event is `LBTC.sol`. These are two different contracts that sit behind two different proxy addresses. Thus, the nodes will only successfully notarize events emitted from one of these two smart contracts.

Consider updating the value of the `Params` such that both the smart contract addresses are able to be queried by the nodes running `notaryd` such that they are able to validate both types of payloads.

Update: *Resolved in pull request #112 at commits 9ddfcca and 689d7e9.*

Medium Severity

M-01 Potential Selector Clash in the Future

The available selectors for the payload are defined as the bytes4 cast of the keccak256 hash of "payload(...)", with the expected types of the parameters in the "...". Currently, each selector is unique because each type of payload has different parameters. However, this risks future conflicts if new selectors are introduced with identical parameter types as an existing selector even though they have different functionality, which would cause a function selector clash.

Therefore, instead of the string "payload", consider using the name of the selector instead. For example, the deposit selector could be the bytes4 cast of the keccak256 hash of `"deposit(...)"`.

Update: *Resolved in pull request #96 at commits faf92f7c and 411cea5.*

M-02 Lack of Parameter Validation in `UpdateParams` Allows Setting Invalid Values

In the `x/notary` module, the `UpdateParams` function lacks validation for the parameters being updated, accepting any values without verifying their correctness or acceptable ranges. The function only checks the authority of the sender and then proceeds to set the parameters using `k.SetParams(ctx, req.Params)` without validating critical fields such as `NotaryThreshold` and `NotaryDepositPeriod`. This absence of validation may allow for invalid or harmful configurations to be set accidentally, potentially leading to unexpected behavior, security vulnerabilities, or unexpected errors in the module's functionality.

Consider implementing thorough validation within the `UpdateParams` function to ensure that all parameters meet the defined criteria and constraints before applying them.

Update: *Resolved in pull request #85 at commit 3cb6890.*

M-03 Inability to Remove Compromised Proxy Due to Missing Message Handler

In the `x/notary` module, the `RemoveProxiedNotary` function allows for removing a proxy. However, there is no exposed message handler that would allow for calling this function to remove a proxy. This means that if a proxy's private key becomes exposed or compromised, there is no mechanism to revoke or eliminate the affected proxy from the system. As a result, the validator would be forced to unregister itself from the network completely to avoid allowing its compromised proxy to perform unauthorized actions.

Consider implementing an exposed message handler that enables a validator to remove its proxy when necessary.

Update: *Resolved in pull request #97 at commit 1b065a0.*

M-04 `RegisterNotary` Allows Registration of Non-Validators

In the `x/notary` module, the `RegisterNotary` function does not contain any checks to ensure that the creator address belongs to an active validator. It accepts a creator address and proceeds to register a proxy without checking if the address corresponds to a current validator in the network. This omission allows any user to register a proxy as a notary.

Consider implementing a validation step that confirms that the `Creator` address is associated with an active validator before allowing the registration of a proxied notary.

Update: *Acknowledged, will resolve. The Lombard team stated:*

> *We plan to counter this issue with a fix when implementing economic incentives for the public release, which will come later than the version in scope.*

M-05 Deprecated `ValidateBasic` Function Prevents Unauthorized Validator Set Updates

The Ledger chain has a module that allows for updating the next set of validators. The typical flow of this process comes from `EndBlocker`, which is called at the end of every block. This function will call `createPendingValSet`, which under certain conditions, calls `SetPendingValset`, which creates a new notary session with payload information regarding the new set of validators.

The existing validators, upon the creation of this new notary session, listen for the `NotarizationStarted` event, and will process this notarization. In order to determine the validity of this notary session, the `Verify` function is called and specifically when the update val set strategy is used, the `Verify` function within `update_val_set_strategy.go` will be called. Since this function is essentially a no-op, the existing validators will all sign the payload claiming the set of new validators.

The ability to send an `UpdateValSetSelector` payload is restricted in this `ValidateBasic` function. However, this function is deprecated in the Cosmos SDK `0.50`. While it is still invoked in the current SDK, if future versions cease to call `ValidateBasic`, a malicious attacker can submit an arbitrary payload to Ledger containing a list of "validators" that they control. Note that because this payload is arbitrary and does not pass through the `EndBlocker` function, this set does not need to truly be "validators" on the chain. Since the

`Verify` function as mentioned above is a no-op, all the existing validators will sign this malicious payload.

The user can then submit this malicious payload on the EVM chain by calling `setNextValidatorSet`, as this function can be called by anyone. Since this function only checks that the signatures are valid from the existing validator set, the malicious payload will succeed and the new validator set that the attacker controls will be placed into storage. From this point forward, the attacker controls this chain and can perform actions such as minting themselves free LBTC.

Consider refactoring the logic that prevents anyone to submit a `UpdateValSetSelector` payload from the deprecated `ValidateBasic` function into the keeper module.

Update: *Resolved in pull request #61 at commit 2ad9f12 and pull request #119 at commit 4354b1a. However, if the Lombard team plans to increase the maximum number of validators in the future, it will be crucial to implement pagination in the bonded validator query.*

Additionally, since `ValidateBasic` has been deprecated in Cosmos SDK 0.50, consider refactoring its logic into the respective `Msg services` to ensure compatibility with future SDK versions.

Low Severity

L-01 Abuse of Notation for Public Key

During the registration flow, a validator must use a public key to sign the message which consists of the validator address. To sign this message, the validator passes the validator address cast into a bytes array to the `SignPubKey` function. However, the `SignPubkey` function takes a parameter of type `exported.PublicKey`. This only works because the `exported.PublicKey` type is also an array of bytes. However, it is an abuse of notation that just so happens to have no consequences because both happen to be byte arrays.

Consider changing the `SignPubKey` function to accept a validator address instead of a public key to avoid confusion and future issues.

Update: *Resolved in pull request #89 at commit 7b8b52e. The Lombard team stated:*

> *We added an explicit conversion so that the knowledge that address and key corresponds is part of the codebase. Also, in case of a type change a compilation error will be raised.*

L-02 `SubmitSignature` Accepts New Signatures for Completed Sessions

In the `SubmitSignature` function, the session state is checked to be either pending or completed. If it is neither, an error is returned. However, only sessions with a pending state should be able to accept new signatures. Allowing signatures to be submitted for completed sessions can lead to inconsistent session states and unintended behavior within the system.

Consider updating the state validation logic in the `SubmitSignature` function to restrict signature submissions exclusively to sessions that are pending and returning an error for any other state, including `types.Completed`.

Update: *Acknowledged, not resolved. The Lombard team stated:*

> *The possibility was left open on purpose to cover the common case in which all notaries send a signature but the completion threshold is reached with a subset of them, since threshold could be between 67% and 100%. Once signature is received as transaction, if legitimate, we do not see a clear reason to reject the additional approval, which would be an additional proof of correctness.*

L-03 Improper Use Of Context Inheritance

There are multiple instances across the codebase, where new root contexts are created using `context.Background()` instead of inheriting from parent contexts. This approach bypasses parent context deadlines or cancellation signals, causing timeout handling to malfunction and potentially leading to resource leaks. Examples include:

- `x/deposit/module/module.go`
- `notaryd/events/source.go`
- `notaryd/events/source.go`
- `notaryd/events/source.go`
- `notaryd/events/source.go`

Generally, `context.Background()` should only be used at the topmost level of the application or for isolated testing purposes, while all other contexts should be derived from the top-level context.

Consider replacing all occurrences of `context.Background()` that are not part of top-level initialization or test code with contexts inherited from a parent to ensure proper propagation of deadlines and cancellations.

Update: *Resolved in pull request #101 at commit 167ea9c. The Lombard team stated:*

> *First location in `module.go` provides a configuration call which is not supposed fail or hang on. Context is only used because auto generated from the proto files and the implementation itself does not rely on it, since it is never used. This is a general practice as can be seen in any other module of the Cosmos SDK. - `tryUnsubscribe` (117) is only executed at node shutdown, so there is no cancel to propagate and takes itself a timeout parameter, so there is not any inherited context to pass. - in third, fourth, and fifth location (126, 226, 239) having a different context but having a `select` statement that includes the parent channel, too, allows to avoid performing the loop operations on operation cancellation, which would be unnecessary The attached PR includes a call to cancel that was missing to release resources.*

L-04 Mismatched Validator and Proxy Address Returns Notary

The `GetNotary` function supports a query to obtain a proxied notary. `types.QueryGetNotaryRequest` has two fields, `Validator` and `Proxy`, that allow a user to query for the notary using either the validator address or the proxy address. If there is a non-empty validator address in the request, it returns the notary related to that validator address regardless of whether the proxy address is empty or not. However, if a user submits a request with both a validator and a proxy address and the two do not correspond (i.e., the proxy address is incorrect), it will still return the notary associated with the validator address. This is both error-prone and confusing to the caller.

Consider throwing an error if both the validator address and the proxy address are submitted as parameters for the query at the same time.

Update: *Resolved in pull request #90 at commits eb18265 and 529ec11. The Lombard team stated:*

> *It allows to put both but returns an error if, when both are set, they do not belong to same notary.*

L-05 Disallow Chain ID of 0

In the `DeriveAddress` function of `query_derive_address.go`, a check ensures that `chainId.Sign()` is not equal to -1. If it is, an error stating "bad chain_id" is returned. However, such an error is not returned when the chain ID is 0.

Consider updating the aforementioned check's `if` statement to include the situation where `chainId.Sign()` is equal to 0.

Update: *Resolved in pull request #91 at commit f829013.*

L-06 Length of Payload Body Checked Against Wrong Max Payload Length

The `ValidateBasic` function in `message_submit_payload.go` ensures that `MsgSubmitPayload` passes basic checks. In this function, the `Payload` field is split into the 4-byte selector and the body (the rest of the payload). The length of the body is checked against `exported.MaxPayloadLength` and an error is returned if it exceeds this value. However, `exported.MaxPayloadLength` has been defined to include `SelectorLength`.

Consider adjusting the validation to compare the body to `exported.MaxPayloadLength` - 4 bytes to ensure proper enforcement of the payload size limit.

Update: *Resolved in pull request #92 at commits d259fee and b45fa76.*

L-07 Inconsistency in Name and Symbol

The `LBTC.sol` contract defines a storage struct that contains the storage variables `name` and `symbol` but also inherits from `ERC20PausableUpgradeable`, which inherits from `ERC20Upgradeable`, which contains its own `_name` and `_symbol` fields.

The slots in the `ERC20Upgradeable` contract are set in `initialize` as "LBTC" and "LBTC", but later within the same function, there is as call to `__LBTC_init`, where "Lombard Staked Bitcoin" and "LBTC" are stored as `name` and `symbol`. The getters `name()` and `symbol()` in this contract use the fields defined in `LBTCStorage.`

Having two slots defined for both name and symbol with conflicting values is confusing and misleading for users. Consider only having one slot defined for the name and symbol. Alternatively, if two slots for name and for symbol are desired, consider forcing consistency in the values for these two slots.

Update: *Resolved in pull request #77 at commit 818dad1. The Lombard team stated:*

> *We fixed it by initializing the ERC20Upgradeable storage with empty strings, as we cannot access the storage from the LBTC contract (`private` visibility).*

L-08 Misleading Documentation

Throughout the codebase, multiple instances of misleading comments were identified:

- All the comments in `x/deposit/exported/public_key.go` refer to the LBTC address, but should be updated to say "Public Key" instead.
- This comment says that "ValidateBasic returns an error if the hash is not a valid", but should be updated to refer to `PayloadMetadata` instead of the hash.
- This comment says that the `_proof` should be an encoding of "validators, weights, signatures", but it is only comprised of the signatures
- The documentation for this param calls it `dstEid`, but the struct defines the field as `eid`

Consider revising the aforementioned comments to improve consistency and more accurately reflect the implemented logic, making it easier for auditors and other parties examining the code to understand what each section of code is designed to do.

Update: *Resolved in pull request #103 at commit 28584fa and pull request #86 at commit 23de29c.*

L-09 Function Selector Not Aligned with Data Type

In `Actions.sol`, the function selector for `DEPOSIT_BRIDGE_ACTION` is defined to have a `uint256` as its last parameter. However, when this function selector is actually used as part of the payload in `Bridge.sol`, the last parameter is of type `bytes32`.

Consider updating the function selector for `DEPOSIT_BRIDGE_ACTION` to reflect the actual data types that will be passed into the payload to improve clarity and ensure consistency.

Update: *Resolved in pull request #76 at commit f98c700. The Lombard team stated:*

> *It seemed simpler to just update the type from `bytes32` to `uint256`, as it already seems to just be treated as a nonce.*

L-10 Definition of `MAX_COMMISSION` in More Than One Location

In `Bridge.sol`, the constant `MAX_COMMISSION` is defined as 10000. The constant `MAX_COMMISSION` is also defined in `FeeUtils.sol`. The two values must be equal as `Bridge.sol` relies on `FeeUtils.sol` to validate the commission so that a malicious owner cannot set an effective commission higher than the maximum. While these values are currently consistent, maintaining separate definitions creates a risk of future mismatches. If one value changes without updating the other, it could lead to vulnerabilities.

Consider defining MAX_COMMISSION in a single location, such as a shared library or constants file, and referencing it in both Bridge.sol and FeeUtils.sol. This approach ensures consistency, reduces maintenance overhead, and minimizes the likelihood of errors during future updates.

Update: *Resolved in pull request #75 at commit c30ca92.*

L-11 ValidateBasic Does Not Perform Validation on Metadata

The function `ValidateBasic` from `NotarySession` calls `s.Metadata.ValidateBasic`, but this method is currently a no-op, resulting in no validation of the `Metadata` field. Since the metadata is used by the `notaryd` to decide whether or not to sign, it should not be allowed to have arbitrary or malformed metadata.

Consider implementing validation logic for the `Metadata` field and migrating away from using the `ValidateBasic` method, since as of Cosmos SDK `0.50`, this method has been deprecated in favor of validating messages directly in their respective message services.

Update: *Resolved in pull request #104 at commit 40171bd. The Lombard team stated:*

> *Metadata is meant to be generic information to better look for payload information, so the data structure itself is not supposed to fail validation since this would be carried out*

> *based on context usage. The PR adds checks the metadata to avoid data beyond the necessary and to check only required payloads carry one.*

L-12 Mismatch Between SubmitPayload Method Arguments and Usage String

The `SubmitPayload` command usage string and the actual implementation of its positional arguments are inconsistent. The usage string currently specifies three arguments: `payload`, `type`, and `metadata`. However, the method implementation only accepts two positional arguments: `payload` and `metadata`, omitting the `type` argument entirely. This mismatch can result in errors when users attempt to invoke the command based on outdated or misleading usage information.

Consider updating the usage string to align with the actual method implementation by removing the `type` argument.

Update: *Resolved in pull request #102 at commit dc60e70.*

L-13 Inefficient Retry Mechanism in `queryNotarySessionWithRetry`

The function `queryNotarySessionWithRetry` currently uses an exponential backoff strategy for retries, which is suboptimal for blockchain operations where state updates depend on block intervals. Exponential backoff introduces unpredictable and potentially mismatched delays, leading to longer than necessary wait times.

Consider replacing the exponential backoff strategy with a block-based waiting mechanism. This approach would synchronize retries with the blockchain's average block time, ensuring predictable and efficient intervals between attempts.

Update: *Acknowledged, not resolved in pull request #52. The Lombard team stated:*

> *The context of the mentioned code is that it addresses a known issue of the Cosmos SDK where state does not get updated before the new block event is emitted so if query is right after block arrival, state may be outdated and the newly created session may not be there. Unfortunately we did not find the issue where the problem is tracked but this was confirmed by Informal Systems that mantains part of the Cosmos SDK. Instead of a one-shot wait we implemented a backoff strategy which is more resilient. We did not tie*

> *to block update since the core of the problem is that state update and block arrivals have small misalignment.*

Notes & Additional Information

N-01 Code Redundancy

The `GetUnstake` and `GetDepositBridge` functions are used to query the receipt of the transaction on an EVM chain to validate the submitted payload. These functions perform nearly identical operations, with the difference being just grabbing a different emitted event. Thus, large portions of `GetUnstake` and `GetDepositBridge` have almost identical code, with the only difference being the topic.

Consider moving the redundant code to a helper function to improve code readability and to prevent errors when refactoring.

Update: *Resolved in pull request #94 at commits 0c92606 and 082a261.*

N-02 Inconsistency in Metadata Usage

In `deposit_bridge_strategy.go`, the `Verify` function uses the `message` and `metadata` parameters for verification. Namely, `message` corresponds to the payload of the event emitted from the smart contract and `metadata` contains the `TxHash` and `EventIndex` where this event occurred. In `unstakes_strategy.go`, the `Verify` function takes in the `message` and metadata parameters, but the metadata goes unused. Instead, the `TxHash` and `EventIndex` are fields in the `message` itself. Since both strategies are similar in that they are reading events emitted by an EVM chain, this can confuse the user as to what is supposed to go into the payload and what goes into the metadata. Furthermore, there are locations where the code becomes more complex due to the metadata being in the message as opposed to the metadata.

For consistency, consider updating `unstakes_strategy.go` to require `TxHash` and `EventIndex` to be in the metadata instead of the payload.

Update: *Acknowledged, will resolve. The Lombard team stated:*

> *We will consider the fix in next release since the change may impact other components.*

N-03 Unused Error

Unused code can negatively impact the clarity and maintainability of the codebase. An error named `ErrSample` has been defined in `x/deposit/types/errors.go` and `x/notary/types/errors.go`. However, this error is not being used anywhere in the codebase.

To improve the conciseness and clarity of the codebase, consider removing the `ErrSample` unused error.

Update: *Resolved in pull request #85 at commits 70e4c53 and 7bdaba2.*

N-04 Suboptimal Configuration Defaults

The configuration settings in `notaryd/config` include parameters that may not be optimally tuned for Ethereum's network characteristics, potentially impacting performance and usability. The settings include:

- `EVMConfig.RequiredConfirmations` is set to 64, which is unnecessarily high for Ethereum and significantly delays finalizing operations.
- MaxLatestBlockAge is set to 15 seconds, which is overly aggressive given Ethereum's average block time of ~12 seconds.

Consider reducing `EVMConfig.RequiredConfirmations` to a more reasonable value aligned with common practices on Ethereum (e.g., 12-15 confirmations) which balances security and latency. Similarly, consider increasing the `MaxLatestBlockAge` to 20-30 seconds to account for occasional network delays and to improve tolerance for block propagation.

Update: *Resolved in pull request #93 at commit c72ada4. `EVMConfig.RequiredConfirmations` was kept at **64** because the Lombard team wanted to assess full finalisation.*

N-05 Hardcoded Credentials and Network Configuration

The function `DefaultNotarydConfig()` includes hardcoded credentials like `User: "1"` and `Pass: "1"` for the Bitcoin RPC configuration. These values may lead to confusion or errors if not appropriately overridden during deployment. Additionally, in the function `NewFetcher`, the `mainnet` value is hardcoded instead of being retrieved from configuration. This hardcoded value introduces the risk of accidental interaction with the Bitcoin mainnet during testing or development.

Consider removing the hardcoded credentials and using environment variables or a secure configuration file to ensure flexibility and clarity in production settings.

Update: *Resolved in pull request #76 and pull request #106. The Lombard team stated:*

> *Config is now taken from parameters in the following PRs: https://github.com/lombard-finance/ledger/pull/76 https://github.com/lombard-finance/ledger/pull/106.*

N-06 Internal Functions Can Be Modifiers

In `LBTC.sol`, there are functions `_onlyMinter` and `_onlyClaimer`, which validate if an address is a minter or a claimer, respectively.

These functions are used for access control in the `mint`, `batchMint`, `burn`, `mintWithFee`, and `batchMintWithFee` functions. Consider moving or wrapping these functions with a modifier for code clarity and to save gas.

Update: *Resolved in pull request #78 at commit efb59cf.*

N-07 Owner Can Change Name and Symbol

In the `LBTC.sol` contract, an owner can change the name and the symbol of the token by calling the `changeNameAndSymbol` function. While this function is protected so it can only be called by the owner, the result of changing a token name and symbol can be confusing and result in a poor user experience. This would impact token listings on a CEX or DEX, wallet integrations, and blockchain explorers, potentially misleading users and reducing trust in the token.

Consider removing the `changeNameAndSymbol` function so its not possible to change the name and symbol of the token.

Update: *Resolved in pull request #82 at commit 2771c12.*

N-08 Naming Recommendations

To favor explicitness and readability, there are several locations in the contracts that may benefit from better naming:

- The `SetPubkey` function in `proxied_notary.go` not only sets the pubkey, but also sets `Active` to true. Consider renaming it to `SetPubkeyAndActive`.
- The parameter `lbtc_addresses` in `params.proto` only stores a single LBTC address. Consider renaming it to `lbtc_address`.
- The selector `UnstakeExecutionSelector` is used to validate that a Bitcoin payment was made. Consider renaming it to `UnstakeClaimedSelector` or `UnstakePaidSelector`.
- The file `unstakes_strategy.go` could be named `unstake_strategy.go`.
- The error `FeeGreaterThanAmount` could be named `FeeGreaterThanOrEqualToAmount` since it is used in this if statement

Update: *Partially resolved in pull request #105 at commit 9201bff. The Lombard team stated:*

> *Modified only ones that do not involve changes in other components.*

N-09 Unnecessary Cast

In `EfficientRateLimiter.sol`, in the `_checkAndUpdateRateLimit` function, when `rl.lastUpdated` is updated with the `block.timestamp`, it is cast into a `uint128`. Similarly, `oppositeRL.lastUpdated` is cast into a `uint128` as well.

This is unnecessary as the `RateLimit` struct defines the `lastUpdated` field as a `uint256`. Consider removing this unnecessary cast.

Update: *Resolved in pull request #79 at commit fbd54a7.*

N-10 Unnecessary Unchecked Loop Increment

In `Consortium.sol`, there is a for loop within the `_checkProof` function that does not increment the value of `i` on the same line as the `for`, but opts to increment `i` in an unchecked block at the end of each loop iteration, rather than directly in the `for` loop header. Similarly, in the `_setRateLimits` function of `EfficientRateLimiter.sol`, a larger unchecked block contains the loop logic. These patters were likely introduced to save gas, but they introduce unnecessary complexity and risk as they can lead to overflows and underflows, particularly if the code is eventually refactored.

In Solidity `0.8.22`, a release announcement was made that introduced "an overflow check optimization that automatically generates an unchecked arithmetic increment of the counter of `for` loops." Since the codebase is using a version of Solidity greater than 0.8.22, writing code using unchecked blocks solely for the purpose of loop incrementing is unnecessary.

Consider refactoring the loops to remove the unchecked blocks and use idiomatic for loop constructs where the increment is defined in the loop header.

Update: *Resolved in pull request #80 at commit ab368fe.*

N-11 Unused Function

The internal function `_calcRelativeFee` in `Bridge.sol` is unused.

Consider removing this function, and if it is necessary to calculate the relative fees in the future, the `calcRelativeFee` function inside of `libs/FeeUtils.sol` can be used instead.

Update: *Resolved in pull request #83 at commit 0c8c8ff.*

N-12 Unnecessary Use Of `this` Prefix In Internal Function Call

The use of the `this` prefix when calling `checkProof` is unnecessary since `checkProof` is a public function, so it can be called both internally and externally.

The use of the `this` prefix would end up calling `checkProof` as an external call, leading to increased gas costs and resetting the execution context.

Consider removing the `this` prefix when calling `checkProof` to improve gas efficiency and preserve the execution context.

Update: *Resolved in pull request #81 at commit a801094.*

N-13 Code Duplication

The files `EfficientRateLimiter.sol` and `RateLimits.sol` have a lot of similar code. Following the DRY principle of programming, it is encouraged to not write the same code in multiple places for code maintainability. Some instances in these two files of code that could be written once and reused are:

- The `RateLimit` struct in `EfficientRateLimiter` and the `Data` struct in `RateLimits`
- The `RateLimitConfig` struct in `EfficientRateLimiter` and the `Config` struct in `RateLimits`
- The `RateLimitExceeded` error in `EfficientRateLimiter` and the `RateLimitExceeded` error in `RateLimits`
- The `_calculateDecay` function in `EfficientRateLimiter` and the `availableAmountToSend` function in `RateLimits`

Consider only writing code one time and reusing it in multiple locations.

Update: *Resolved in pull request #84 at commits f68125e and 7495dca.*

Conclusion

Several important areas for improvement were identified in Lombard's ledger and smart contracts, particularly regarding payload validation, memory management in the pubsub system, and transaction verification. Key considerations include fixing checks for sanctioned addresses, preventing accounting errors in unstaking transactions, and addressing Cosmos SDK deprecations. As the protocol matures, transitioning the current permissioned Consortium system toward an economically secure model will be crucial for long-term protocol safety.

The Lombard team was highly responsive throughout the audit process and worked diligently to address the identified issues.