SHERLOCK

Security Review For Lombard Finance

Collaborative Audit Prepared For: **Lombard Finance**
Lead Security Expert(s): **Bauchibred**
defsec
n4nika
Date Audited: **June 24 - July 15, 2025**
Final Commit: **86b00b6**

Introduction

This audit is focused on a major upgrades to the Lombard protocol:

1. Yield-Bearing LBTC
2. General Message Passing (GMP)
3. Multi-asset Support

Scope

Repository: lombard-finance/smart-contracts

Audited Commit: 794ca81b17b6d98af0096a3f2e8758c035a45642

Final Commit: 86b00b641bbf799dbfa5f27b5b9dd1cab332f7ed

Files:

- contracts/LBTC/AssetRouter.sol
- contracts/LBTC/BaseLBTC.sol
- contracts/LBTC/NativeLBTC.sol
- contracts/LBTC/StakedLBTC.sol
- contracts/LBTC/StakedLBTCOracle.sol
- contracts/LBTC/interfaces/IAssetRouter.sol
- contracts/LBTC/interfaces/IBaseLBTC.sol
- contracts/LBTC/interfaces/INativeLBTC.sol
- contracts/LBTC/interfaces/IOracle.sol
- contracts/LBTC/interfaces/IStakedLBTC.sol
- contracts/LBTC/libraries/Assert.sol
- contracts/LBTC/libraries/Assets.sol
- contracts/LBTC/libraries/Redeem.sol
- contracts/LBTC/libraries/Validation.sol
- contracts/bridge/BridgeV2.sol
- contracts/bridge/IBridgeV2.sol
- contracts/bridge/adapters/CLAdapter.sol
- contracts/bridge/adapters/TokenPool.sol
- contracts/bridge/providers/LombardTokenPoolV2.sol
- contracts/gmp/IHandler.sol
- contracts/gmp/IMailbox.sol

- contracts/gmp/Mailbox.sol
- contracts/gmp/libs/GMPUtils.sol
- contracts/gmp/libs/MessagePath.sol
- contracts/interfaces/IERC20MintableBurnable.sol
- contracts/libs/Actions.sol
- contracts/libs/LChainId.sol
- contracts/stakeAndBake/StakeAndBake.sol

Final Commit Hash

86b00b641bbf799dbfa5f27b5b9dd1cab332f7ed

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
5	5	8

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: BridgeV2 deposits are not rate limited

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/310

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`BridgeV2` applies the `RateLimits` library only during the withdrawal path, completely omitting it in `deposit() / _deposit()`, allowing for unprotected deposits.

Vulnerability Detail

First note:

```
function _withdraw(
    BridgeV2Storage storage $,
    bytes32 chainId,
    bytes memory msgBody
) internal {
    (address token, address recipient, uint256 amount) = decodeMsgBody(
        msgBody
    );

    if (
        $.allowedDestinationToken[
            _calcAllowedTokenId(chainId, GMPUtils.addressToBytes32(token))
        ] == bytes32(0)
    ) {
        revert BridgeV2_TokenNotAllowed();
    }

    // check rate limits
    RateLimits.Data storage rl = $.rateLimit[
        _calcRateLimitId(chainId, token)
    ];
    RateLimits.updateLimit(rl, amount);

    IERC20MintableBurnable(token).mint(recipient, amount);

    emit WithdrawFromBridge(recipient, chainId, token, amount);
}
```

As seen during withdrawals, we rightly apply the rate limit check, however this is completely missing in the deposit path, whcih is in contrast of the classic implementation of even the v1 bridge where we apply rate limits on both deposit and withdrawals.

Impact

Protection provided by rate limitting is completely sidestepped on the deposit side, allowing for deposits of arbitrarily large amounts of the supported token(s) in a short window if chain is compromised.

Recommendation

Re-introduce the rate limit check in the deposits flow.

Discussion

555-andrew

In general we are considering rate limit logic implemented on Ledger side as a primary and sufficient for deposits and withdrawals. We decided to keep rate limits for withdrawals in smart-contract as and extra safety measure to prevent/protect us from minting huge quantities of tokens. But still it is an extra protection. We do not think deposit need the same since deposit does not presume any token to be minted.

Bauchibred

Acknowledged, but currently there is no ratelimitting via GMP on ledger, only the IBC one from the previous iteration.

I mean currently we don't check a rate limit for the EVM <-> Ledger flow since we use GMP and it's not wired in the hooks. Also thing with rate limits is that it's a to/fro flow, so if we block only one side on a chain, then the other path is vulnerable even on another chain connected to this.

Issue H-2: `BridgeV2::deposit()` when providing sender wrongly burns tokens from relayer

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/311

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`BridgeV2.deposit(bytes32,address,address,bytes32,uint256,bytes32)` allows one to relay a deposit on behalf of another user by passing an explicit `sender` address. While that `sender` is encoded into the GMP payload, the bridge wrongly deducts tokens from `msg.sender` instead of `sender`. As a result the ledger chain will later mint tokens for `sender` although the value was burned from a different account, allowing users siphon tokens from the relayer if they have enough assets that can be burnt.

Vulnerability Detail

We have two paths of depositting, one where we can pass the sender address and one where we can't:

https://github.com/sherlock-audit/2025-06-lombard-june-23rd/blob/bf819ce727cac95ee9e8c2408688fa610fe34730/smart-contracts/contracts/bridge/BridgeV2.sol#L267-L313

```
    function deposit(
        bytes32 destinationChain,
        address token,
        address sender,
        bytes32 recipient,
        uint256 amount,
        bytes32 destinationCaller
    ) external payable override nonReentrant returns (uint256, bytes32) {
        return
            _deposit(
                destinationChain,
                IERC20MintableBurnable(token),
|>                sender,
                recipient,
                amount,
                destinationCaller
            );
    }

    function deposit(
        bytes32 destinationChain,
        address token,
```

```
        bytes32 recipient,
        uint256 amount,
        bytes32 destinationCaller
    ) external payable override nonReentrant returns (uint256, bytes32) {
        return
            _deposit(
                destinationChain,
                IERC20MintableBurnable(token),
|>                _msgSender(),
                recipient,
                amount,
                destinationCaller
            );
    }
```

Now during the internal `_deposit()` call this sender gets encoded as the sender and then we should be burning from this `sender`

https://github.com/sherlock-audit/2025-06-lombard-june-23rd/blob/bf819ce727cac95ee9e8c2408688fa610fe34730/smart-contracts/contracts/bridge/BridgeV2.sol#L314-L377

```
    function _deposit(
// ..snip
    ) internal returns (uint256 nonce, bytes32 payloadHash) {
// ..snip

|>        _burnToken(token, amount);

        bytes memory body = _encodeMsg(
            destinationToken,
|>            GMPUtils.addressToBytes32(sender),
            recipient,
            amount
        );
..snip

        emit DepositToBridge(sender, recipient, payloadHash);
        return (nonce, payloadHash);
    }
```

Issue however is that `_burnToken` unconditionally debits `msg.sender` and then

```
function _burnToken(IERC20MintableBurnable token, uint256 amount) internal {
    SafeERC20.safeTransferFrom(token, _msgSender(), address(this), amount);
    token.burn(amount);
}
```

Impact

Relayed deposits are broken, in the worse case this allows for depositors to deplete the relayer's address of the tokens other the attempt to depoist just reverts if there are not enough tokens to send from the relayer's address.

Recommendation

Transfer the assets from the encoded `sender`.

Discussion

555-andrew

This implementation is intentional, but not the final. Please response to the issue https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/320

Issue H-3: Ratio change is wrongly activated causing for wrong staked LBTC to native LBTC conversion

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/318

Summary

`StakedLBTCOracle::_ratio()` uses a comparison logic that activates the freshly-published ratio for the entire interval before the intended `switchTime`, instead of explicitly waiting for the switch time to pass before using the new rates, causing for wrong convversion rates of stakedLBTC to native LBTC.

Vulnerability Detail

The consortium is to sign in new ratios with their switch times when needed, and when updating the ratio we set the current ratio to the `prevratio` and then the new ratio to the `currRatio`, and in the same light store the switch time, which is when we are then expected to start using the new ratio:

StakedLBTCOracle.sol#L129-L154

```
function _publishNewRatio(
    bytes calldata rawPayload,
    bytes calldata proof
) internal {
    Assert.selector(rawPayload, Actions.RATIO_UPDATE);
    Actions.RatioUpdate memory action = Actions.ratioUpdate(rawPayload[4:]);
    StakedLBTCOracleStorage storage $ = _getStakedLBTCOracleStorage();
    if (
        $.switchTime > action.switchTime ||
        (action.switchTime - block.timestamp) > $.maxAheadInterval
    ) {
        revert WrongRatioSwitchTime();
    }
    bytes32 payloadHash = sha256(rawPayload);
    $.consortium.checkProof(payloadHash, proof);
    _setNewRatio(action.ratio, action.switchTime);
}

function _setNewRatio(uint256 ratio_, uint256 switchTime_) internal {
    StakedLBTCOracleStorage storage $ = _getStakedLBTCOracleStorage();
    $.prevRatio = $.currRatio;
    $.currRatio = ratio_;
    $.switchTime = switchTime_;
    emit Oracle_RatioChanged($.prevRatio, $.currRatio, $.switchTime);
```

```
}
```

StakedLBTCOracle::_ratio() is to be used to decide which rate is active:

```
function _ratio() internal view returns (uint256) {
    if (block.timestamp <= $.switchTime) {
        return $.currRatio;          // |> becomes active while we are still
↪  "before" switchTime
    }
    return $.prevRatio; // |> Wrongly goes back to older price after switch time
}
```

But as seen on `_ratio()` because the check uses <=, the new `currRatio` is already in force for the whole open interval `[now … switchTime]` and then we wrongly set the rate back to the older rate when the instead of only from `switchTime` forward.
An attacker (or even an honest user) can therefore front-run the official activation time and mint or redeem at a more favourable rate.

Step-by-step attack path:

1. Consortium publishes a new higher ratio `R` with `switchTime = T`.
2. Immediately after the transaction is finalised (time `now < T`) a user:
    - calls `AssetRouter.mint()` which indirectly reads `oracle.ratio()`;
    - receives LBTC at the **high** ratio `R` rather than the still-valid `R`.
3. When the scheduled time arrives, the ratio does **not** change (it was already active), so no one notices the early usage.

Conditions required: [consortium] signs a valid `RATIO_UPDATE` and sets `switchTime` to be in the future.

Impact

Wrong convertion rates are always going to be used for staked LBTC to native LBTC and viceversa which depending on the path funds are currently being moved would cause for a leak of funds for the users or protocol.

Recommendation

Swap the comparison logic:

```
function _ratio() internal view returns (uint256) {
-    if (block.timestamp <= $.switchTime) {
+    if (block.timestamp < $.switchTime) {
-        return $.currRatio;
+    return $.prevRatio;
```

```
    }
-    return $.prevRatio;
+        return $.currRatio;
}
```

(or equivalently >= for the new ratio) so that the update only becomes effective **at or after** `switchTime`. Add unit tests covering boundary timestamps to prevent regressions.

Discussion

555-andrew

This was fixed in https://github.com/lombard-finance/smart-contracts/commit/eb434f152b538385b40d251649d397be8b6e846e

Issue H-4: Incorrect length check in `decodeMsgBody` causes user funds to be stuck when bridging

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/326

Summary

A discrepancy in the size check of bridging payloads between the notaries and the `BridgeV2` contract cause user funds to be stuck when bridging to an EVM chain via GMP messages.

Vulnerability Detail

GMP bridging messages have the following format:

```
var bridgeMsgV1 = abi.Arguments{
    {Type: abi.Bytes32Ty}, // destination token
    {Type: abi.Bytes32Ty}, // sender
    {Type: abi.Bytes32Ty}, // recipient
    {Type: abi.Uint256Ty}, // amount
}
```

This is properly used in the notaries and `_encodeMsg` in `BridgeV2`.

`decodeMsgBody`, however, assumes the message to only contain three 32-bytes fields instead of four:

```
if (msgBody.length != MSG_LENGTH) {
        revert BridgeV2_InvalidMsgBodyLength(MSG_LENGTH, msgBody.length);
}

uint8 version;
bytes32 token;
bytes32 recipient;
uint256 amount;
```

Note that `MSG_LENGTH = 97`.

This means that if as user initiates bridging from one EVM chain to another (GMP bridging is only supported for EVM chains right now), they will not be able to execute the notary-signed bridging transaction on the destination chain as it will always revert.

Impact

Freezing of user funds.

Recommendation

Consider properly decoding the message in `decodeMsgBody`, accounting for all four fields of the payload.

Discussion

555-andrew

was fixed in https://github.com/lombard-finance/smart-contracts/commit/74a5e945346420cd89274c98943f3431f7bd0aed

Issue H-5: The possibility to swap `CBBTC` and `BTCB` to `LBTC` via minting breaks the system

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/412

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Since it is possible to swap `CBBTC` and `BTCB` to `LBTC` and that function utilising a permissioned `mint` to create the corresponding `LBTC`, the internal accounting of per-chain balances is completely broken.

Vulnerability Detail

Currently it is possible to swap `CBBTC` and `BTCB` to `LBTC`:

`CBBTC.sol`:

```
function swapCBBTCToLBTC(uint256 amount) external whenNotPaused {
        // [...]
        cbbtc.safeTransferFrom(_msgSender(), address(this), amountCBBTC);
        lbtc.mint(_msgSender(), amountLBTC - fee);
        lbtc.mint(address(this), fee);
}
```

Here `LBTC` is minted through the permissioned `mint` function, not requiring a notarized payload to execute.

This means that this mint cannot be tracked by the ledger. Since it cannot be tracked, the here minted `LBTC` does not count towards the per-chain supply of the affected chain, meaning the internally tracked supply will be smaller than the actual supply of `LBTC`.

NOTE here: this is an inherent design problem. If any minting logic exists in other supported chains' contracts (OOS for this audit), this just makes this issue worse and fixing it harder.

Impact

This will make it impossible for some LBTC to be redeemed back to BTC, breaking a crucial part of the system.

Additionally, this makes the effective `ratio` incorrect as that `LBTC` (`StakedLBTC` in v2) is not backed by `NativeLBTC` as it should be.

Recommendation

Fixing this is far from trivial. First of all this means that already when migrating to v2, the internal balances will be wrong since migration only takes `LBTC` into consideration which was minted through a notarized session which is not the case here.

This issue inherently breaks the system and fixing it would require completely disabling the `CBBTC` and `BTCB` swaps alltogether as one step of the mitigation.

Discussion

555-andrew

We deprecated (switched off) CBBTC and do not have plans to enable it in future. In worst case we will update it to match new approach to control total token supply.

n4nika

Have both `CBBTC` and `BTCB` been disabled? And when they were disabled, was the previously minted LBTC burnt or kept?

le0n229

CBBTC was disabled some months ago, BTCB was disabled yesterday and we do not plan to enable it. LBTC will be kept, we will include it in the migration logic to mint this supply.

n4nika

Perfect, please also be aware that any mint logic on any other chain which is not triggered by a notarized payload causes the same problem. If there is any, this must be considered as well

Issue M-1: Updating ratio pre current switch time wrongly overwrites pending ratio

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/313

Summary

`_setNewRatio` discards any still-pending ratio by rewriting both `prevRatio` and `currRatio` every time a new update is accepted. If a second future ratio is published before the first one reaches its `switchTime` (which could be done if the consortium is to update the ratio), the first ratio is lost and the system transitions directly to the newer value, bypassing the intended interim period on the first change

Vulnerability Detail

Note that the only restriction in regards to time is that when setting a new ratio we must have the new `switchTime` be higher than the previous one and currently in the future for not more than `maxAheadInterval`:

StakedLBTCOracle.sol#L129-L146

```
function _publishNewRatio(
    bytes calldata rawPayload,
    bytes calldata proof
) internal {
    Assert.selector(rawPayload, Actions.RATIO_UPDATE);
    Actions.RatioUpdate memory action = Actions.ratioUpdate(rawPayload[4:]);
    StakedLBTCOracleStorage storage $ = _getStakedLBTCOracleStorage();
    if (
        $.switchTime > action.switchTime ||
        (action.switchTime - block.timestamp) > $.maxAheadInterval
    ) {
        revert WrongRatioSwitchTime();
    }
    bytes32 payloadHash = sha256(rawPayload);
    $.consortium.checkProof(payloadHash, proof);
    _setNewRatio(action.ratio, action.switchTime);
}
```

And finally in order to change the ratio, StakedLBTCOracle::_setNewRatio ends up being called:

```
function _setNewRatio(uint256 ratio_, uint256 switchTime_) internal {
    $.prevRatio = $.currRatio;  //  overwrites history
    $.currRatio = ratio_;       //  replaces pending ratio
```

```
    $.switchTime = switchTime_; //  replaces pending switch time
}
```

So a classic scenario where, at timestamp 08:00 the consortium agrees to set `B-ratio` with `switchTime = 12:00`.
• Oracle now holds `prev = A`, `curr = B`, `switchTime = 10:00`. Before timestamp 12:00 a second update is passed to set `C-ratio` with `switchTime = 18:00`.
• The code above replaces `prev = B`, `curr = C`, `switchTime = 18:00`.

But the correct behaviour is to use `B` between 08:00 → 12:00, then `C` from 12:00 → 18:00, which doesn't happen since the current logic intends to have **B** become active immediately (since `prev` was moved to `B`), starting at 10:00 instead of 12:00, shortening or eliminating the intended waiting window.

NB: The above hypothesis is what happens after deploying the fix on the wrong comparison check in the other report

Impact

The oracle relays wrong conversion rates for a specific period, contrary tot he hypothesis in the *vulnerailty details* this can be for long timestamps.

Recommendation

Consider storing pending updates in a separate struct and then adjust `_ratio()` to return the correct value based on the current time.

Discussion

555-andrew

This will be fixed in a bit different way: we will overwrite existing scheduled ratio (with the `switchTime` in future). Also most probably we will not be using oracle for the time being due some product-side considerations. Anyway, the fix will be provided soon.

555-andrew

Here is the commit with the fix: https://github.com/lombard-finance/smart-contracts/commit/d57bca6ec425692379d81681cca0a5979e46bb11 We do not want to make ratio update logic to be over-complicated. So any pending ratio (with switchTime in future) can be overwritten by a new value.

Issue M-2: Custom `transfer()` blocks multi-asset support for the router

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/315

Summary

`AssetRouter` is intended to support multiple assets, however it assumes that every supported token exposes a non-standard `transfer(address from, address to, uint256 amount)` function, identical to the one implemented by Lombard's native and staked LBTC.
Classic ERC-20s implement the canonical `transfer(address to, uint256 amount)` and will therefore revert whenever the router calls the three-argument variant during the redeem or mint with fee flow.

Vulnerability Detail

Both NativeLBTC and StakedLBTC implement the non-standard `transfer(address from, address to, uint256 amount)` function.

NativeLBTC::transfer()

```
function transfer(address from, address to, uint256 amount) external
↪  onlyRole(MINTER_ROLE) {
    _transfer(from, to, amount);
}
```

StakedLBTC::transfer()

```
function transfer(address from, address to, uint256 amount) external onlyMinter {
    _transfer(from, to, amount);
}
```

Router fee deduction (redeem & mint) all include a block like the below when fee is non zero:

```
|> tokenContract.transfer(recipient, treasury, fee);
```

Because generic assets lack this 3-arg selector the call reverts immediately blocking minting or redemption for that asset.

Impact

All to-be introduced assets would hit a revert when trying to use the router for minting or redeeming.

Recommendation

Refactor to ERC-20-compatible fee collection.

Discussion

555-andrew

Acknowledged. The commit with the fix will be provided soon.

555-andrew

Here is the commit with the fix: https://github.com/lombard-finance/smart-contracts/commit/d57bca6ec425692379d81681cca0a5979e46bb11

Issue M-3: Non-btc redemption is allowed when the token is disabled

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/316

Summary

`redeemForBtc` checks `tokenConfigs[fromToken].isRedeemEnabled` before proceeding, but the two other `redeem()` path for non-btc omit this guard.

Vulnerability Detail

Guard present in the BTC-native path:

```
function redeemForBtc(
    address fromAddress,
    address fromToken,
    bytes calldata recipient,
    uint256 amount
) external nonReentrant {
    AssetRouterStorage storage $ = _getAssetRouterStorage();
    uint64 fee = $.toNativeCommission;
    if (!$.tokenConfigs[fromToken].isRedeemEnabled) {
        revert AssetRouter_RedeemsForBtcDisabled();
    }
```

But missing in both general redemption paths as neither calls `isRedeemEnabled`:

AssetRouter::redeem(bytes32,...)
AssetRouter::redeem(address,...)

Impact

`_redeem` executes successfully even while the token's redemption is supposed to be paused.

Recommendation

Consider moving the check inside `_redeem` so it covers every call path.

Discussion

555-andrew

Probably the issue is about parameter naming, because `isRedeemEnabled` is assumed to keep information about if redeem for BTC enabled only. So we will add check to ensure that `redeem` cannot be called with Bitcoin Chain ID as destination chain, because `redeemForBtc` should be called instead. We will provide commit with the fix soon.

555-andrew

Actually we decided to get rid of `isRedeemEnabled` completely. And use route setting instead. So if route type is UNKNOWN, the route is considered to be disabled. This helps to simplify the storage and control over different actions and simplifies code a bit. Here is the commit with the fix: https://github.com/lombard-finance/smart-contracts/commit/d57bca6ec425692379d81681cca0a5979e46bb11

Issue M-4: Incorrect length check in `validatePayload`

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/410

Summary

`validatePayload` incorrectly checks the length of the payload since it does not take the selector into account.

Vulnerability Detail

The function checks that the raw payload is not smaller than `MIN_GMP_LENGTH`:

```
function validatePayload(bytes calldata rawPayload) internal pure {
        if (bytes4(rawPayload) != GMP_V1_SELECTOR) {
                revert GMP_InvalidAction(GMP_V1_SELECTOR, bytes4(rawPayload));
        }
        if (rawPayload.length < MIN_GMP_LENGTH) {
                revert GMP_WrongPayloadLength();
        }
}
```

Since `MIN_GMP_LENGTH = 32*6`, this is wrong since it does not account for the selector. For example in the ledger, the payload is also checked to be not smaller thane `32*6 bytes`, however, there the selector is already cut away when checking the length.

Impact

Incorrect size check, causing correctly formatted payloads to not be deliverable

Recommendation

Consider changing the check to:

```
if (rawPayload.length < (MIN_GMP_LENGTH + 4)) {
        revert GMP_WrongPayloadLength();
}
```

Discussion

555-andrew

Acknowledged. Fixed here:
https://github.com/lombard-finance/smart-contracts/pull/263

Issue M-5: [Ledger/Contracts] Onchain ratio can't be updated

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/411

Summary

Since the `RATIO_UPDATE` selector is no supported in the ledger, such payloads can never be notarized, making it impossible to update the ratio in `StakedLBTCOracle.sol`.

Vulnerability Detail

In order to update the ratio of the `StakedLBTCOracle`, `publishNewRatio` must be called with a payload and a corresponding proof. Such a payload must have the `RATIO_UPDATE` selector.

```
function _publishNewRatio(
        bytes calldata rawPayload,
        bytes calldata proof
) internal {
        Assert.selector(rawPayload, Actions.RATIO_UPDATE);
        Actions.RatioUpdate memory action = Actions.ratioUpdate(rawPayload[4:]);
        // [...]
}
```

Since this selector is not supported in the ledger, it is not even possible to create a session with that selector, let alone get it notarized by the notaries:

`selector.go`:

```
var (
    DepositSelectorV0 Selector = [SelectorLength]byte{0xf2, 0xe7, 0x3f, 0x7c}
    DepositSelectorV1 Selector = [SelectorLength]byte{0xce, 0x25, 0xe7, 0xc2}
    UnstakeSelector Selector = [SelectorLength]byte{0x9e, 0xa1, 0x13, 0x2b}
    BridgeDepositSelector Selector = [SelectorLength]byte{0x5c, 0x70, 0xa5, 0x05}
    UpdateValSetSelector Selector = [SelectorLength]byte{0x4a, 0xab, 0x1d, 0x6f}
    UnstakeExecutionSelector Selector = [SelectorLength]byte{0x2a, 0x4c, 0x23, 0xfc}
    GMPMessageV1Selector Selector = [SelectorLength]byte{0xe2, 0x88, 0xfb, 0x4a}
)
```

Impact

Impossible to update onchain ratios, causing ratio in ledger and ratios on supported chains to be out of sync.

Additional note here: Even if this was properly implemented, it would be impossible to properly synchronise the ratio in the ledger with the ratios on supported chains in realtime. This may lead to issues in itself.

Recommendation

Fixing this is non-trivial. It requires implementing a whole new ratio-update strategy with code changes in the ledger and the notary clients.

Discussion

555-andrew

The ratio update logic on ledger side is in development, should be ready soon or is ready already.

555-andrew

Here is the link to the code that has ratio selector:
https://github.com/lombard-finance/ledger/blob/4e4dde6e138a726391f4f6ed0ee4001ef3e65673/x/notary/exported/selector.go
allowbreak #L47

n4nika

In order to fix this, explicit handling in the notary clients is required, just adding the descriptor does not fix this

russanto

Same commit, notaryd folder, https://github.com/lombard-finance/ledger/blob/4e4dde6e138a726391f4f6ed0ee4001ef3e65673/notaryd/verifier/update_ratio_strategy.go

n4nika

A few notes here:

- Even then, EVM and ledger will not always be perfectly in sync due to signing overhead
- Right now the length of the payload is not checked to be exact, meaning we can again get two payload notarized which have the same values but not the same hash (I don't see it as that impactful here since replay is prevented in a different way)
- With this implementation it is possible to set a ratio between LBTC/LBTC which will not be 1, this makes no sense. This is because we have the LBTC denom in the `denomHashes` map, I would remove that
- The way replay protection is implemented is questionable. The only thing preventing a user from resetting the ratio to an older value seems to be the check `\$.switchTime > action.switchTime` in `StakedLBTCOracle.sol`. If for example there

is a ratio which gets notarized at `block.timestamp + 100` and then 50 blocks later another one gets notarized for now `block.timestamp + 50` (so the same timestamp), both of them point at the same block, meaning any user can submit either of the two interchangeably. From how I see it, that check for the `switchTime` should be >= instead of > to ensure it's not possible to set multiple ratios at the same timestamp

Also, could you please provide the PR where all these changes were made? It's very hard to verify this properly without it

lpetroulakis

After speaking with the team, fixes are being added/designated on their roadmap.

555-andrew

Here is the fix for the fourth point (smart-contract side):
https://github.com/lombard-finance/smart-contracts/pull/268

n4nika

Another thing. Updating the ratio will never work due to this:

```
newRatio = types.NewRatio(uint64(ctx.BlockTime().Unix()), newValue)
```

Here in `UpdateRatio`, the `switchTime` of the payload is set to `now`.

When we then check this:

```
(switchTime_ - block.timestamp) > \$.maxAheadInterval
```

It will fail since `block.timestamp` will be bigger than `switchTime_`, underflowing

555-andrew

Acknowledged. Here is the fix:
https://github.com/lombard-finance/smart-contracts/pull/269/files

Issue L-1: `StakedLBTC` does not correctly enforce pause

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/309

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`StakedLBTC` enforces `paused()` only in its batch mint pathways, leaving the single-mint functions unchecked.

Vulnerability Detail

StakedLBTC::batchMint reverts when paused:

```
function batchMint(
    bytes[] calldata payload,
    bytes[] calldata proof
) external nonReentrant {
    if (paused()) {
        revert EnforcedPause();
    }
    StakedLBTCStorage storage $ = _getStakedLBTCStorage();
    if (address($.assetRouter) == address(0)) {
        revert AssetRouterNotSet();
    }
    $.assetRouter.batchMint(payload, proof);
}
```

So also `batchMintWithFee` reverts when paused:

But the regular mint/mintWithFee path lacks this guard:

StakedLBTC::mint:

```
function mint(bytes calldata rawPayload, bytes calldata proof)
    external
    nonReentrant
    returns (address recipient)
{
    ...
    return $.assetRouter.mint(rawPayload, proof);   // ← no pause check
}
```

Impact

During an emergency pause attempts to mint still get passed on to the mailbox via the router even though they would end up failing.

Recommendation

Add the same `paused()` check `mint(bytes,bytes)` and ‘mintWithFee(...).

Discussion

555-andrew

Both `NativeLBTC` and `StakedLBTC` have `BaseLBTC` as a parent class. And `BaseLBTC` inherits from `ERC20PausableUpgradeable`. That means any call to `_mint`, `_burn` or other function that update balance will revert if contract is paused. It might be not so obvious in case of `StakedLBTC`, but `mint(rawpayload, proof)` on token side calls AssetRouter's mint which passes message to the mailbox and ultimately token's `mint(recepient, amount)` gets called so it will revert if token contract is paused too.

Bauchibred

Unlike the case in `NativeLBTC`, i.e #314, I'd recommend we relay the same protection as is available on `StakeLBTC\#batchMint()` on `StakeLBTC\#mint()`, considering if it is paused and the mint would end up getting reverted on the final call, we can just block the attempt at the outermost level than querying the router to then pass a message to the mailbox that would end up reverting.

Bauchibred

Updated the report to outline the last comment.

555-andrew

Yes, it is possible, but `mint()` function on `StakedLBTC` side was left for backward compatibility reasons. AssetRouters's mint is supposed to be called directly from now on. And any checks in `StakedLBTC` mint() will not have any effect if `AssetRouter`'s mint() is called.

Issue L-2: Discounted fees are not refunded to users

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/312

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`BridgeV2._assertFee` only checks `msg.value >= expectedFee` and forwards all `msg.value` to `Mailbox`, never refunding any surplus. Issue hwoever is that there is currently a discount system which would mean that in some cases users could overestimate the in which case they would pay in excess.

Vulnerability Detail

Owners can set a discount for whitelisted senders, which affects how much they have to send as fees to the mailbox, however during deposits the full msg.value is sent to the mailbox even if higher than the fee:

https://github.com/sherlock-audit/2025-06-lombard-june-23rd/blob/bf819ce727cac95ee9e8c2408688fa610fe34730/smart-contracts/contracts/bridge/BridgeV2.sol#L366-L373

```
// send message via mailbox
(nonce, payloadHash) = $.mailbox.send{value: msg.value}(
    destinationChain,
    _destinationBridge,
    destinationCaller,
    body
);
```

Impact

Silent loss of funds, albeit this would be in minute sums

Recommendation

Check when msg.value > expectedFee and refund `msg.sender` the difference, similar to what's been done in the CLAdapter when more than needed fees are sent: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/blob/bf819ce727cac95ee9e8c2408688fa610fe34730/smart-contracts/contracts/bridge/adapters/CLAdapter.sol#L187-L190

```
if (msg.value > fee) {
    uint256 refundAm = msg.value - fee;
```

```
    refunds[fromAddress] += refundAm;
}
```

Discussion

555-andrew

We do not do refunds intentionally due to the time constrains. We are going to consider any extra fee amount paid by contract caller as a request to treat this bridge transaction as a higher priority. Still we might decide to implement refund in future. (But not right now.)

Issue L-3: Tokens can still access the router post removal

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/317

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`removeRoute()` deletes the route mapping but never revokes the `CALLER_ROLE` previously granted to the token when the route was added.
Consequently the token contract keeps its privileged status and may keep calling AssetRouter functions even after all routes pointing to it were supposedly disabled.

Vulnerability Detail

When a route is registered the helper `_checkAndSetNativeToken()` calls `grantRole(CALLER_ROLE, tokenAddress);`.
That role is what `_isAllowedCaller()` checks before every redeem / deposit.

`removeRoute()` only deletes the mapping entry:

AssetRouter::removeRoute():

```
function removeRoute(
    bytes32 fromToken,
    bytes32 fromChainId,
    bytes32 toToken,
    bytes32 toChainId
) external onlyRole(DEFAULT_ADMIN_ROLE) {
    AssetRouterStorage storage $ = _getAssetRouterStorage();
    bytes32 key = keccak256(abi.encode(fromToken, toChainId));
    Route storage r = $.routes[key];
|>    delete r.toTokens[toToken];
    emit AssetRouter_RouteRemoved(fromToken, fromChainId, toToken, toChainId);
}
```

It never executes `revokeRole(CALLER_ROLE, token)`, so the privilege lingers forever.

Impact

A token that was once supported and had a route can still access the redeem functionality.

Recommendation

Call `revokeRole(CALLER_ROLE, tokenAddress)` inside `removeRoute()`.

Discussion

555-andrew

The roles should be revoked only manually. There is a good chance that some token is being a source or destination for several routes (like `StakedLBTC` and be destination for both BTC and `NativeBTC` deposit). So it would be wrong to blindly revoke `CALLER_ROLE` for the token composing the route to be removed.

Bauchibred

Acknowledged, lowering the severity since if need be the manual revoke role is queried.

Issue L-4: [Smart Contract] Inconsistent whitelist validation on the sender parameter

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/320

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The BridgeV2 contract has inconsistent whitelist validation that allows whitelisted callers to bridge tokens on behalf of non-whitelisted addresses, effectively bypassing the whitelist mechanism designed to control who can use the bridge.

Vulnerability Detail

The BridgeV2 contract implements a whitelist system where only approved addresses can initiate bridge deposits. However, there's an issue in how this validation is implemented across different deposit functions.

The contract provides two deposit functions:

1. `deposit(destinationChain, token, recipient, amount, destinationCaller)` - burns tokens from `msg.sender`
2. `deposit(destinationChain, token, sender, recipient, amount, destinationCaller)` - accepts a `sender` parameter

Both functions call the internal `_deposit` function, which performs the whitelist check:

```
if (!$.senderConfig[_msgSender()].whitelisted) {
    revert BridgeV2_SenderNotWhitelisted(_msgSender());
}
```

The issue is that this check only validates that the caller (`_msgSender()`) is whitelisted, but doesn't validate the `sender` parameter that gets encoded into the cross-chain message. The `sender` parameter is used to identify who the tokens are coming from in the destination chain message, but the whitelist doesn't restrict this value.

Impact

The whitelist is not applied for the sender.

Code Snippet

```
function deposit(
    bytes32 destinationChain,
```

```
    address token,
    address sender,  // ← This parameter is not validated against whitelist
    bytes32 recipient,
    uint256 amount,
    bytes32 destinationCaller
) external payable override nonReentrant returns (uint256, bytes32) {
    return _deposit(
        destinationChain,
        IERC20MintableBurnable(token),
        sender,  // ← Used in cross-chain message
        recipient,
        amount,
        destinationCaller
    );
}
```

Whitelist check only validates caller:

```
function _deposit(...) internal returns (uint256 nonce, bytes32 payloadHash) {
    // ...
    if (!$.senderConfig[_msgSender()].whitelisted) {  // ← Only checks caller
        revert BridgeV2_SenderNotWhitelisted(_msgSender());
    }
    // ...
    bytes memory body = _encodeMsg(
        destinationToken,
        GMPUtils.addressToBytes32(sender),  // ← But uses sender parameter here
        recipient,
        amount
    );
}
```

Token burning always from caller:

```
function _burnToken(IERC20MintableBurnable token, uint256 amount) internal {
    SafeERC20.safeTransferFrom(token, _msgSender(), address(this), amount);  // ←
 ↪  Always from caller
    token.burn(amount);
}
```

Tool Used

Manual Review

Recommendation

Add a whitelist check for the `sender` parameter in addition to the caller.

Discussion

555-andrew

This is not the final version of the bridgeV2. The `sender` is supposed to be the address who owns the tokens to be bridged. The `whitelist` is not supposed to filter token owners, it is supposed to filter the caller of `deposit()` function. The final version will check if sender is whitelisted only if `deposit(destinationChain, token, sender, recipient, amount, destinationCaller)` is called. The purpose is to make sure that only trusted addresses can call it and provide `sender` that we cannot actually verify. Another version (`deposit(destinationChain, token, recipient, amount, destinationCaller)`) will have a dedicated flag to enable/disable it and will be available to everyone. The reason for such complexity is to make bridge available for both external addresses and trusted smart-contracts while having more or less reliable way to figure know who is/was the token owner.

defsec

Marked as an acknowledged.

Issue L-5: Validation threshold should never be set as non zero

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/337

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

`Bascule::updateValidateThreshold` lets governance raise the validation threshold; afterwards any withdrawal whose amount is equal to or below that new limit skips on-chain validation. Also the deposit state never gets set as withdrawn when the value is less than this threshold, since `AssetRouter::handlePayload` only checks that `validateWithdrawal` does not revert.

Vulnerability Detail

When the guardian raises `_validateThreshold`, `validateWithdrawal` falls through to the "not validated" branch for any `withdrawalAmount < newThreshold`, and as seen we only set the deposit status to withdrawn when it is in the reported state:

contract::validateWithdrawal():

```
    function validateWithdrawal(
        bytes32 depositID,
        uint256 withdrawalAmount
    ) public whenNotPaused onlyRole(WITHDRAWAL_VALIDATOR_ROLE) {
        DepositState status = depositHistory[depositID];
        // Deposit found and not withdrawn
        if (status == DepositState.REPORTED) {
|>            depositHistory[depositID] = DepositState.WITHDRAWN;
            emit WithdrawalValidated(depositID, withdrawalAmount);
            return;
        }
        // Already withdrawn
        if (status == DepositState.WITHDRAWN) {
            revert AlreadyWithdrawn(depositID, withdrawalAmount);
        }
        // Not reported
        if (withdrawalAmount >= validateThreshold()) {
            // We disallow a withdrawal if it's not in the depositHistory and
            // the value is above the threshold.
|>            revert WithdrawalFailedValidation(depositID, withdrawalAmount);
        }
        // We don't have the depositID in the depositHistory, and the value of the
        // withdrawal is below the threshold, so we allow the withdrawal without
        // additional on-chain validation.
```

```
        //
        // Unlike in original Bascule, this contract records withdrawals
        // even when the validation threshold is raised.
        depositHistory[depositID] = DepositState.WITHDRAWN;
        emit WithdrawalNotValidated(depositID, withdrawalAmount);
    }
```

Impact

Unreported withdrawals of up to the threshold value are directly accepted and not validated or even checked if reported.

Recommendation

Consider deleting the `WithdrawalNotValidated` fallback so every withdrawal must match a pre-reported deposit or set the `depositHistory[depositID]` to `DepositState.WITHDRAWN` even if the deposit has not been report but we are going to proceed with the mint.

Alternatively never set the threshold to a value other than `0`.

Discussion

555-andrew

Due to the business reasons we need small deposits to pass through even if deposit has not been reported. Regarding the second recommendation, I think `validateWithdrawal()` is setting deposit status in `depositHistory` to `WITHDRAWN` state already. At the last two lines if this function.

Bauchibred

Acknowledged

Issue L-6: Wrong emissions are made post `lockOrBurn`

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/338

Summary

`LombardTokenPoolV2` emits `Burned(msg.sender, amount)` where `msg.sender` is the on-ramp contract, not the actual user who initiated the burn. This hides the true actor from logs.

Vulnerability Detail

In `LombardTokenPoolV2::lockOrBurn` the contract incorrectly uses `msg.sender` when emitting the `Burned` event:

contracts/bridge/providers/LombardTokenPoolV2.sol::lockOrBurn:

```
emit Burned(msg.sender, lockOrBurnIn.amount);
```

`msg.sender` is an intermediary whereas the true initiator is provided in `lockOrBurnIn.originalSender.`.

Impact

Off-chain systems relying on the event will map burns to the wrong address, undermining monitoring and incident response

Recommendation

Emit `originalSender` instead of `msg.sender`:

```
emit Burned(lockOrBurnIn.originalSender, lockOrBurnIn.amount);
```

Discussion

555-andrew

Acknowledged. Fixed here: https://github.com/lombard-finance/smart-contracts/commit/9e5c6b10e2fb8d06aed0dfa4fe6a34ca73b4721d

Issue L-7: IChainId cannot represent IDs larger than `256-8 bits`

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/402

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

The `lChainId` used in the system can not represent all possible chain ids

Vulnerability Detail

When getting an `lChainId`'s ecosystem, the ID'S MSB byte is taken. Since chainIDs themselves can be up to 64 bytes big, this means any chain which chooses a chainID utilising more than `256-8 bits` cannot be meaningfully supported by the system.

In addition to that, looking at `LChainId.sol`, this could lead to bigger problems if the contracts are deployed on a chain which has a chain id like this: `0xff0000...0000`

```
return
        bytes32(
                block.chainid &

↪   0x00FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
        );
```

This would then collide with mainnet's chainid.

Impact

Restriction on chainID.

Recommendation

Fixing this would require rewriting all payloads to include a ”from ecosystem”. Since this is not in proportion with the added robustness, it is recommended to not ”fix” this but keep it in mind for the future for the very unlikely case that such a chain is supposed to be supported.

Discussion

russanto

Any chain that either does not have a defined chain Id, or cannot fit into the 31 available bytes will have its chain information hashed and mapped to LChainId. This gives possibility to support 2^256 ecosystems and 2^31 chains, which is way beyond what necessary.

Issue L-8: When initializing `StakedLBTCOracle`, `switchTime` should not be set in the future

Source: https://github.com/sherlock-audit/2025-06-lombard-june-23rd/issues/413

Summary

Setting the `switchTime` to anything larger than `block.timestamp` when initializing `StakedLBTCOracle` would lead to incorrect computations.

Vulnerability Detail

Currently the `switchTime` when setting the initial ratio is provided by the entity initializing the contract. If this value is set in the future, any calls to `ratio` will return `0` until the set `switchTime` is reached. Since a value of `0` is obviously wrong here, this should NOT be done.

Impact

Incorrect calculations

Recommendation

Consider hardcoding the `switchTime` to be `block.timestamp` when initializing, preventing this alltogether.

Discussion

555-andrew

This was fixed in https://github.com/lombard-finance/smart-contracts/pull/250/files

n4nika

Fix looks good

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.