LBTC
Lombard

HALBORN

LBTC - Lombard

Prepared by: HALBORN

Last Updated 08/20/2024

Date of Engagement by: July 23rd, 2024 - August 5th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
10	0	1	0	2	7

TABLE OF CONTENTS

Introduction

Lombard engaged Halborn to conduct a security assessment on their smart contracts beginning on July 23rd, 2024, and ending on August 5th, 2024. The security assessment was scoped to the smart contracts provided in the lombard-finance/evm-smart-contracts GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

Assessment Summary

Halborn was provided 2 weeks for the engagement, and assigned one full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.
The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some security issues, that were successfully addressed by the Lombard team. The main security issues were:

- **Denial of Service and Permanent loss of funds.**
- **Centralization risk for trusted owners.**
- **ERC1271 implementation divergences.**

Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Static Analysis of security for scoped contract, and imported functions (slither).
- Testnet deployment (hardhat).

1. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

1.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

1.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

1.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2. SCOPE

FILES AND REPOSITORY

(a) Repository: evm-smart-contracts

(b) Assessed Commit ID: 0482146

(c) Items in scope:

- ./consortium/LombardConsortium.sol
- ./consortium/LombardTimeLock.sol
- ./libs/OutputCodec.sol
- ./libs/BridgeDepositCodec.sol
- ./libs/EIP1271SignatureUtils.sol
- ./libs/BitcoinUtils.sol
- ./bascule/Bascule.sol
- ./bascule/interfaces/IBascule.sol
- ./LBTC/ILBTC.sol
- ./LBTC/LBTC.sol
- lombard-finance/evm-smart-contracts/commit/8c9fd06192fb47a2b704a07e80f0698305071e1b

Out-of-Scope:

REMEDIATION COMMIT ID:

- 76e6b85
- e162000

Out-of-Scope: New features/implementations after the remediation commit IDs.

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	0	2	7

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DENIAL OF SERVICE AND PERMANENT LOSS OF FUNDS	HIGH	SOLVED - 08/07/2024
CENTRALIZATION RISK FOR TRUSTED OWNERS	LOW	SOLVED - 08/07/2024
ERC1271 IMPLEMENTATION DIVERGENCES	LOW	SOLVED - 08/07/2024
FLOATING PRAGMA	INFORMATIONAL	SOLVED - 08/07/2024
EVENTS FIELDS ARE MISSING "INDEXED" ATTRIBUTE	INFORMATIONAL	SOLVED - 08/07/2024
CHANGE 'PUBLIC' FUNCTIONS THAT ARE NOT INVOKED INTERNALLY TO 'EXTERNAL'	INFORMATIONAL	SOLVED - 08/07/2024
PUSH0 AND OTHER OPCODES ARE NOT SUPPORTED ON ALL CHAINS	INFORMATIONAL	ACKNOWLEDGED
INTERNAL FUNCTION USED ONCE CAN BE EMBEDDED	INFORMATIONAL	ACKNOWLEDGED
VARIABLES INITIALIZED WITH DEFAULT VALUE	INFORMATIONAL	ACKNOWLEDGED
ITERATE WITH '++I' FOR ENHANCED GAS-EFFICIENCY	INFORMATIONAL	ACKNOWLEDGED

4. FINDINGS & TECH DETAILS

4.1 DENIAL OF SERVICE AND PERMANENT LOSS OF FUNDS

// HIGH

Description

The `burn` function in the `LBTC.sol` smart contract is designed to burn a specified amount of `$LBTC` from the `msgSender()` on the source EVM-chain and transfer the corresponding amount to a destination `scriptPubkey` on the Bitcoin network. The `scriptPubkey` can be of type Taproot (P2TR) or Pay-To-Witness (P2WPKH).

The `scriptPubkey` format is validated through the `BitcoinUtils` library, specifically using the internal `getOutputType` function. The amount to be burned and transferred to the caller's provided `scriptPubkey` on the Bitcoin network is calculated by deducting a fee amount, which is fetched from the `burnCommission` state variable.

`- evm-smart-contracts/contracts/LBTC/LBTC.sol`

```
152     function burn(bytes calldata scriptPubkey, uint256 amount) external
153         OutputType outType = BitcoinUtils.getOutputType(scriptPubkey);
154
155         if (outType == OutputType.UNSUPPORTED) {
156             revert ScriptPubkeyUnsupported();
157         }
158
159         LBTCStorage storage $ = _getLBTCStorage();
160
161         if (!$.isWithdrawalsEnabled) {
162             revert WithdrawalsDisabled();
163         }
164
165         uint64 fee = $.burnCommission;
166         if (amount <= fee) {
167             revert AmountLessThanCommission(fee);
168         }
169         amount -= fee;
170
171         address fromAddress = address(_msgSender());
172         _transfer(fromAddress, getTreasury(), fee);
173         _burn(fromAddress, amount);
174
175         emit UnstakeRequest(
176             fromAddress,
177             scriptPubkey,
178             amount
179
```

```
180         );
        }
```

However, the `burnCommission` state variable is not initialized through the constructor, and the provided scripts do not enforce calling the `changeBurnCommission` function during contract initialization or setup. Consequently, the `fee` internal variable of the `burn` function will fetch the **default value for the type**, which is `zero (0)`.

The issue arises because, in this scenario, **no minimum amount is enforced on each burn operation**. The `burnCommission` fee is intended to serve as a security threshold to **prevent the burning of small amounts of satoshis**, which would result in **loss-making transactions** and **overloaded off-chain processes**.

Since the `burnCommission` state variable is never initialized and thus `fee == 0`, there are no verifications in place to block a malicious user from intentionally requesting the `burn` of smallest amounts, such as `1 satoshi`. The consequences of not having a minimum amount to burn can lead to critical issues, including:

1. **Denial of Service:** The off-chain component will queue all `UnstakeRequest` event emissions and be flooded with requests for small amounts from the same user, effectively blocking or considerably delaying legitimate requests.
2. **Drain of Funds:** The wallet used to sign transactions in off-chain processes will incur gas fees for repetitive small transactions, leading to a drain of funds.

Proof of Concept

To reproduce this vulnerability, any user with a small amount of `LBTC` can call the `burn` function repetitively, passing an extremely low value for `amount`.

In the following example, an attacker sends `10.000` burn transactions. The `burn` function is called and all the events are emitted, effectively leading the off-chain components to a flooded state because of missing input validations on-chain.

PoC Code:

```
describe.only("HAL-10 - Denial of Service and Permanent loss of funds", fu
  beforeEach(async function () {
    await snapshot.restore();
    await lbtc.toggleWithdrawals();
  });

  it("Burn smallest amount leads to DOS and funds drainage", async () =>
    for(let i = 0; i < 10_000; i++) {
      const amount = 1;
      const p2wpkh = "0x00143dee6158aac9b40cd766b21a1eb8956e99b10000"; //
      await lbtc["mint(address,uint256)"](await signer1.getAddress(), amou
      await expect(lbtc.connect(signer1).burn(p2wpkh, amount))
        .to.emit(lbtc, "UnstakeRequest")
        .withArgs(await signer1.getAddress(), p2wpkh, amount);
    }
```

```
    });
  })
```

```
    Burn smallest amounts
      ✔ Burn smallest amount leads to DOS and funds drainage (16752ms)


  1 passing (17s)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:M/D:L/Y:N/R:N/S:C (7.8)

Recommendation

1. Initialize the `burnCommission` state variable during contract deployment to ensure a non-zero fee is applied.
2. Enforce a minimum `burn` amount to prevent the burning of small, non-viable amounts of `satoshis`.

Remediation Plan

SOLVED: The **Lombard team** has solved this issue. The commit hash is `76e6b85a704b9edf5c7ebe9e41675c25a4ce89dd`.

Remediation Hash

https://github.com/lombard-finance/evm-smart-contracts/commit/76e6b85a704b9edf5c7ebe9e41675c25a4ce89dd

4.2 CENTRALIZATION RISK FOR TRUSTED OWNERS

// LOW

Description

The contracts in-scope are safe-guarded by an access control mechanism, which is a good practice in smart contract development and prevents non-authorized parties and bad actors to perform unauthorized activities within the contracts and ecosystem.
Owners with privileged rights to perform administrative operations need to be trusted and have power mitigated to prevent unilateral actions.

- contracts/LBTC/LBTC.sol

```
function toggleWithdrawals() external onlyOwner {
```

- contracts/LBTC/LBTC.sol

```
function changeNameAndSymbol(string calldata name_, string calldata symbol_
```

- contracts/LBTC/LBTC.sol

```
function changeConsortium(address newVal) external onlyOwner {
```

- contracts/LBTC/LBTC.sol

```
function addDestination(bytes32 toChain, bytes32 toContract, uint16 relComm
```

- contracts/LBTC/LBTC.sol

```
function removeDestination(bytes32 toChain) external onlyOwner {
```

- contracts/LBTC/LBTC.sol

```
function changeBascule(address newVal) external onlyOwner {
```

- contracts/LBTC/LBTC.sol

```
function transferPauserRole(address newPauser) external onlyOwner {
```

- contracts/bascule/Bascule.sol

```
function pause() public onlyRole(PAUSER_ROLE) {
```

- contracts/bascule/Bascule.sol

```
function unpause() public onlyRole(PAUSER_ROLE) {
```

- contracts/bascule/Bascule.sol

```
function setMaxDeposits(uint256 aMaxDeposits) public whenNotPaused onlyRole
```

- contracts/bascule/Bascule.sol

```
) public whenNotPaused onlyRole(DEPOSIT_REPORTER_ROLE) {
```

- contracts/bascule/Bascule.sol

```
) public whenNotPaused onlyRole(WITHDRAWAL_VALIDATOR_ROLE) {
```

- contracts/consortium/LombardConsortium.sol

```
function changeThresholdAddr(address newVal) external onlyOwner {
```

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:L/Y:L/R:N/S:U (4.4)

Recommendation

It is recommended to enforce a Multi-signature mechanism, in order to mitigate potential centralization concerns regarding unilateral administrative tasks.

Remediation Plan

SOLVED: The **Lombard team** has solved this issue, by enforcing the usage of **Safe** multi-signature wallets for administrative tasks.

4.3 ERC1271 IMPLEMENTATION DIVERGENCES

// LOW

Description

The current implementation of the `EIP1271SignatureUtils.checkSignature` function contains divergences from the standard ERC1271 specification. Specifically, the address signer is always set to the address of the configured Consortium. Consequently, the `if (isContract(signer))` statement will always return true, as it is expected, that deployed code exists at the specified address. As a result, the condition specified in the `else` block will never execute.

– evm-smart-contracts/contracts/LBTC/LBTC.sol

```
320     function _checkAndUseProof(LBTCStorage storage self, bytes calldata
321         proofHash = keccak256(data);
322
323         // we can trust data only if proof is signed by Consortium
324         EIP1271SignatureUtils.checkSignature(self.consortium, proofHash
325         // We can save the proof, because output with index in unique p
326         if (self.usedProofs[proofHash]) {
327             revert ProofAlreadyUsed();
328         }
329         self.usedProofs[proofHash] = true;
330     }
```

– evm-smart-contracts/contracts/libs/EIP1271SignatureUtils.sol

```
26     function checkSignature(address signer, bytes32 digestHash, bytes m
27
28         if (isContract(signer)) {
29             if (IERC1271(signer).isValidSignature(digestHash, signature
30                 revert SignatureVerificationFailed();
31             }
32         } else {
33             if (ECDSA.recover(digestHash, signature) != signer) {
34                 revert BadSignature();
35             }
36         }
37     }
```

Furthermore, the `checkSignature` function in the `EIP1271SignatureUtils` will forward an external call to the `isValidSignature` function in the `LombardConsortium` contract. This implementation of EIP1271 signature verification through the `isValidSignature` function in the consortium contract differs significantly from the reference implementation as specified in **ERC-1271**.

– evm-smart-contracts/contracts/consortium/LombardConsortium.sol

```
66      function isValidSignature(
67          bytes32 hash,
68          bytes memory signature
69      ) external view override returns (bytes4 magicValue) {
70          ConsortiumStorage storage $ = _getConsortiumStorage();
71
72          if (ECDSA.recover(hash, signature) != $.thresholdAddr) {
73                  revert BadSignature();
74          }
75
76          return EIP1271SignatureUtils.EIP1271_MAGICVALUE;
77      }
```

BVSS

AO:A/AC:L/AX:L/C:L/I:L/A:N/D:N/Y:N/R:N/S:C (3.9)

Recommendation

It is recommended to enforce ERC1271's specifications, such as returning `0xffffffff` **instead of reverting** the transaction when the signature recovery fails.

Remediation Plan

SOLVED: The **Lombard team** has solved this issue. The commit hash is `e1620001e623079ad1040a806f395460d30e61b6`.

Remediation Hash

https://github.com/lombard-finance/evm-smart-contracts/commit/e1620001e623079ad1040a806f395460d30e61b6

4.4 FLOATING PRAGMA

// INFORMATIONAL

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

- contracts/LBTC/LBTC.sol

```
pragma solidity ^0.8.19;
```

- contracts/bascule/Bascule.sol

```
pragma solidity ^0.8.20;
```

- contracts/bascule/interfaces/IBascule.sol

```
pragma solidity ^0.8.20;
```

- contracts/consortium/LombardConsortium.sol

```
pragma solidity ^0.8.24;
```

- contracts/consortium/LombardTimeLock.sol

```
pragma solidity ^0.8.24;
```

- contracts/libs/BitcoinUtils.sol

```
pragma solidity ^0.8.19;
```

- contracts/libs/BridgeDepositCodec.sol

```
pragma solidity ^0.8.24;
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Specify a precise version of Solidity in your contracts instead of using a wide version range. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`. This ensures that the contract is compiled with a specific version of the Solidity compiler, avoiding potential issues that may arise from changes in newer versions.

Remediation Plan

SOLVED: The **Lombard team** has solved this issue. The commit hash is `e1620001e623079ad1040a806f395460d30e61b6`.

Remediation Hash

https://github.com/lombard-finance/evm-smart-contracts/commit/e1620001e623079ad1040a806f395460d30e61b6

4.5 EVENTS FIELDS ARE MISSING "INDEXED" ATTRIBUTE

// INFORMATIONAL

Description

Indexed event fields make the data more quickly accessible to off-chain tools that parse events, and add them to a special data structure known as “topics” instead of the data part of the log. A topic can only hold a single word (32 bytes) so if you use a reference type for an indexed argument, the Keccak–256 hash of the value is stored as a topic instead.
Each event can use up to three indexed fields. If there are fewer than three fields, all the fields can be indexed. It is important to note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed fields per event (three indexed fields).
This is specially recommended when gas usage is not particularly of concern for the emission of the events in question, and the benefits of querying those fields in an easier and straight-forward manner surpasses the downsides of gas usage increase.

- contracts/LBTC/ILBTC.sol

```
        event UnstakeRequest(address indexed fromAddress, bytes scriptPubKe
```

- contracts/LBTC/ILBTC.sol

```
        event WithdrawalsEnabled(bool);
```

- contracts/LBTC/ILBTC.sol

```
        event NameAndSymbolChanged(string name, string symbol);
```

- contracts/LBTC/ILBTC.sol

```
        event OutputProcessed(bytes32 indexed transactionId, uint32 indexed
```

- contracts/LBTC/ILBTC.sol

```
        event DepositToBridge(address indexed fromAddress, bytes32 indexed
```

- contracts/LBTC/ILBTC.sol

```
        event DepositAbsoluteCommissionChanged(uint64 newValue, bytes32 ind
```

- contracts/LBTC/ILBTC.sol

```
        event DepositRelativeCommissionChanged(uint16 newValue, bytes32 ind
```

- contracts/LBTC/ILBTC.sol

```
        event BurnCommissionChanged(uint64 prevValue, uint64 newValue);
```

- contracts/bascule/Bascule.sol

```
      event UpdateValidateThreshold(uint256 oldThreshold, uint256 newThresh
```

- contracts/bascule/Bascule.sol

```
      event MaxDepositsUpdated(uint256 numDeposits);
```

- contracts/bascule/Bascule.sol

```
      event DepositsReported(uint256 numDeposits);
```

- contracts/bascule/Bascule.sol

```
      event WithdrawalNotValidated(bytes32 depositID, uint256 withdrawalAmo
```

- contracts/bascule/Bascule.sol

```
      event WithdrawalValidated(bytes32 depositID, uint256 withdrawalAmount
```

- contracts/consortium/LombardConsortium.sol

```
        event ThresholdAddrChanged(address prevValue, address newValue);
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is recommended to add the `indexed` keyword when declaring events, considering the following example:

```
    event Indexed(
        address indexed from,
        bytes32 indexed id,
        uint indexed value
    );
```

Remediation Plan

SOLVED: The **Lombard team** has solved this issue. The commit hash is `e1620001e623079ad1040a806f395460d30e61b6`.

Remediation Hash

https://github.com/lombard-finance/evm-smart-contracts/commit/e1620001e623079ad1040a806f395460d30e61b6

4.6 CHANGE 'PUBLIC' FUNCTIONS THAT ARE NOT INVOKED INTERNALLY TO 'EXTERNAL'

// INFORMATIONAL

Description

The current contracts in-scope include several functions that are declared as `public` but are not invoked internally within the smart contracts. These functions are intended to be called only from external sources.

- contracts/LBTC/LBTC.sol

```
        function decimals() public override view virtual returns (uint8) {
```

- contracts/LBTC/LBTC.sol

```
        function name() public override view virtual returns (string memory
```

- contracts/LBTC/LBTC.sol

```
        function symbol() public override view virtual returns (string memo
```

- contracts/LBTC/LBTC.sol

```
        function getBurnCommission() public view returns (uint64) {
```

- contracts/bascule/Bascule.sol

```
    function pause() public onlyRole(PAUSER_ROLE) {
```

- contracts/bascule/Bascule.sol

```
    function unpause() public onlyRole(PAUSER_ROLE) {
```

- contracts/bascule/Bascule.sol

```
    function updateValidateThreshold(uint256 newThreshold) public whenNot
```

- contracts/bascule/Bascule.sol

```
    function setMaxDeposits(uint256 aMaxDeposits) public whenNotPaused on
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

To improve code clarity and optimize gas usage, it is recommended to change the visibility of these functions from `public` to `external`.
The `external` keyword is specifically designed for functions that are meant to be called from outside the contract, and it can result in more efficient code execution. By making this change, you can enhance the readability and performance of the smart contract.

Remediation Plan

SOLVED: The **Lombard team** has solved this issue. The commit hash is `e1620001e623079ad1040a806f395460d30e61b6`.

Remediation Hash

https://github.com/lombard-finance/evm-smart-contracts/commit/e1620001e623079ad1040a806f395460d30e61b6

4.7 PUSH0 AND OTHER OPCODES ARE NOT SUPPORTED ON ALL CHAINS

// INFORMATIONAL

Description

Solc compiler `version 0.8.20` switches the default target EVM version to Shanghai. The generated bytecode will include **PUSH0** opcodes. The recently released Solc compiler `version 0.8.25` switches the default target EVM version to Cancun, so it is also important to note that it also adds-up new opcodes such as **TSTORE**, **TLOAD** and **MCOPY**.
Be sure to select the appropriate EVM version in case you intend to deploy on a chain apart from Ethereum mainnet, like L2 chains that may not support **PUSH0**, **TSTORE**, **TLOAD** and/or **MCOPY**, otherwise deployment of your contracts will fail.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

It is important to consider the targeted deployment chains before writing immutable contracts because, in the future, there might exist a need for deploying the contracts in a network that could not support new opcodes from `Shanghai` or `Cancun` EVM versions.

Remediation Plan

ACKNOWLEDGED: The **Lombard team** has acknowledged this issue.

4.8 INTERNAL FUNCTION USED ONCE CAN BE EMBEDDED

// INFORMATIONAL

Description

The current implementation includes an internal function that is called only once within the smart contract. Instead of separating this logic into a separate function, consider inlining the logic directly into the calling function. This approach can reduce the number of function calls and improve code readability.

– evm-smart-contracts/contracts/libs/EIP1271SignatureUtils.sol

```
26      function checkSignature(address signer, bytes32 digestHash, bytes m
27
28          if (isContract(signer)) {
29              if (IERC1271(signer).isValidSignature(digestHash, signature
30                  revert SignatureVerificationFailed();
31              }
32          } else {
33              if (ECDSA.recover(digestHash, signature) != signer) {
34                  revert BadSignature();
35              }
36          }
37      }
38
39      function isContract(address addr) internal view returns (bool) {
40          return addr.code.length != 0;
41      }
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Consider embedding the logic of the internal function used only once **into the caller function itself**. This will enhance code readability and reduce the amount of calls.

Remediation Plan

ACKNOWLEDGED: The **Lombard team** has acknowledged this issue.

4.9 VARIABLES INITIALIZED WITH DEFAULT VALUE

// INFORMATIONAL

Description

For enhanced gas-efficiency, the default values for variables, such as `0` for `uint256` does not have to be initialized.

`- evm-smart-contracts/contracts/bascule/Bascule.sol`

```
226     for (uint256 i = 0; i < numDeposits; i++) {
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Do not initialize variables with their default value.

Remediation Plan

ACKNOWLEDGED: The **Lombard team** has acknowledged this issue.

4.10 ITERATE WITH '++I' FOR ENHANCED GAS-EFFICIENCY

// INFORMATIONAL

Description

Using `++i` costs less gas than `i++`, especially when it's used in `for` loops. The same for `--i` and `i--`, where the first is preferred.

`- evm-smart-contracts/contracts/bascule/Bascule.sol`

```
226     for (uint256 i = 0; i < numDeposits; i++) {
227       bytes32 depositID = depositIDs[i];
228       if (depositHistory[depositID]) {
229         revert AlreadyReported(depositID);
230       }
231       depositHistory[depositID] = true;
232     }
```

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation

Use `++i` / `--i` for enhanced gas-efficiency.

Remediation Plan

ACKNOWLEDGED: The **Lombard team** has acknowledged this issue.

5. AUTOMATED TESTING

Introduction

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

The security team assessed all findings identified by the Slither software, however, findings with severity `Information` and `Optimization` are not included in the below results for the sake of report readability.

```
INFO:Detectors:
TimelockController._execute(address,uint256,bytes) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#412-415) sends eth to arbitrary user
        Dangerous calls:
        - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#413)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#184)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#188)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#189)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#190)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#191)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#192)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#193)
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (node_modules/@openzeppelin/contracts/utils/math/Math.sol#172)
        - result = prod0 * inverse (node_modules/@openzeppelin/contracts/utils/math/Math.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
TimelockController.getOperationState(bytes32) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#207-218) uses a dangerous strict equality:
        - timestamp == 0 (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#209)
TimelockController.getOperationState(bytes32) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#207-218) uses a dangerous strict equality:
        - timestamp == _DONE_TIMESTAMP (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#211)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
TimelockController._execute(address,uint256,bytes) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#412-415) ignores return value by Address.verifyCallResult(success,returndata) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#414)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
Bascule.constructor(address,address,address,address,uint256).defaultAdmin (contracts/bascule/Bascule.sol#107) shadows:
        - AccessControlDefaultAdminRules.defaultAdmin() (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#170-172) (function)
        - IAccessControlDefaultAdminRules.defaultAdmin() (node_modules/@openzeppelin/contracts/access/extensions/IAccessControlDefaultAdminRules.sol#60) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
TimelockController._execute(address,uint256,bytes) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#412-415) has external calls inside a loop: (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#413)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation/#calls-inside-a-loop
INFO:Detectors:
Reentrancy in TimelockController.execute(address,uint256,bytes,bytes32,bytes32) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#358-371):
        External calls:
        - _execute(target,value,payload) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#368)
                - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#413)
        Event emitted after the call(s):
        - CallExecuted(id,0,target,value,payload) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#369)
Reentrancy in TimelockController.executeBatch(address[],uint256[],bytes[],bytes32,bytes32) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#385-407):
        External calls:
        - _execute(target,value,payload) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#403)
                - (success,returndata) = target.call{value: value}(data) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#413)
        Event emitted after the call(s):
        - CallExecuted(id,i,target,value,payload) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#404)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
```

```
INFO:Detectors:
AccessControlDefaultAdminRules.renounceRole(bytes32,address) (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#113-122) uses timestamp for comparisons
        Dangerous comparisons:
        - role == DEFAULT_ADMIN_ROLE && account == defaultAdmin() (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#114)
        - newDefaultAdmin != address(0) || ! _isScheduleSet(schedule) || ! _hasSchedulePassed(schedule) (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#116)
AccessControlDefaultAdminRules._grantRole(bytes32,address) (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#133-141) uses timestamp for comparisons
        Dangerous comparisons:
        - defaultAdmin() != address(0) (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#135)
AccessControlDefaultAdminRules._revokeRole(bytes32,address) (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#146-151) uses timestamp for comparisons
        Dangerous comparisons:
        - role == DEFAULT_ADMIN_ROLE && account == defaultAdmin() (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#147)
AccessControlDefaultAdminRules.defaultAdminDelay() (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#184-187) uses timestamp for comparisons
        Dangerous comparisons:
        - (_isScheduleSet(schedule) && _hasSchedulePassed(schedule)) (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#186)
AccessControlDefaultAdminRules.pendingDefaultAdminDelay() (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#192-195) uses timestamp for comparisons
        Dangerous comparisons:
        - (_isScheduleSet(schedule) && ! _hasSchedulePassed(schedule)) (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#194)
AccessControlDefaultAdminRules.acceptDefaultAdminTransfer() (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#245-252) uses timestamp for comparisons
        Dangerous comparisons:
        - _msgSender() != newDefaultAdmin (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#247)
AccessControlDefaultAdminRules._acceptDefaultAdminTransfer() (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#259-268) uses timestamp for comparisons
        Dangerous comparisons:
        - ! _isScheduleSet(schedule) || ! _hasSchedulePassed(schedule) (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#261)
AccessControlDefaultAdminRules._isScheduleSet(uint48) (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#386-388) uses timestamp for comparisons
        Dangerous comparisons:
        - schedule != 0 (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#387)
AccessControlDefaultAdminRules._hasSchedulePassed(uint48) (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#393-395) uses timestamp for comparisons
        Dangerous comparisons:
        - schedule < block.timestamp (node_modules/@openzeppelin/contracts/access/extensions/AccessControlDefaultAdminRules.sol#394)
TimelockController.getOperationState(bytes32) (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#207-218) uses timestamp for comparisons
        Dangerous comparisons:
        - timestamp == 0 (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#209)
        - timestamp == _DONE_TIMESTAMP (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#211)
        - timestamp > block.timestamp (node_modules/@openzeppelin/contracts/governance/TimelockController.sol#213)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
```

All issues identified by Slither were proved to be false positives, and therefore have not been added to the issue list in this report.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.