BLOCKSEC

Security Audit
Report for Morph Emerald upgrade

Date: December 23, 2025 **Version:** 1.0
Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Morph
Target	Morph Emerald upgrade

Version History

Version	Date	Description
1.0	December 23, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-12-24 09:16:52+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About the Audit Target

Information	Description
Type	Smart Contract, Chain
Language	Solidity, Golang
Approach	Semi-automatic and manual verification

The audit target (hereinafter referred to as the Target) of this audit is the code repository [1][2] of Morph Emerald upgrade of Morph.

The Morph Emerald upgrade of Morph introduces alternative fee transactions (i.e., `AltFeeTx`), enabling native multi-token gas payments on the Morph L2 chain. This upgrade allows users to pay gas fees using registered ERC-20 tokens. Specifically, this feature is enabled through the introduction of the `L2TokenRegistry` contract and corresponding modifications to the Geth codebase. The `L2TokenRegistry` contract allows managers to register fee tokens and update their associated parameters (e.g., token scale and exchange rate). On the Geth side, the project defines the `AltFeeTx` transaction type with two key parameters (i.e., `FeeTokenID` and `FeeLimit`) and implements the corresponding handling logic, including transaction creation, fee calculation, and fee transfer. Notably, the token information used during transaction processing is fetched directly from the `L2TokenRegistry` contract. In addition, the Emerald upgrade synchronizes recent Ethereum mainnet updates by introducing new precompiles and opcodes (e.g., `CLZ`).

Note this audit only focuses on the code in the following directories/files. Code prior to and including the baseline version 0, where applicable, is outside the scope of this audit and assumes to be reliable and secure.

- morph/contracts/contracts/l2/system/L2TokenRegistry.sol
- go-ethereum/accounts/abi/bind/base.go
- go-ethereum/accounts/external/backend.go
- go-ethereum/core/types/receipt.go
- go-ethereum/core/state_processor.go
- go-ethereum/core/tx_list.go
- go-ethereum/core/tx_pool.go
- go-ethereum/core/types/alt_fee_tx.go
- go-ethereum/core/types/token_fee.go
- go-ethereum/core/types/transaction.go
- go-ethereum/internal/ethapi/api.go
- go-ethereum/internal/ethapi/transaction_args.go
- go-ethereum/light/txpool.go
- go-ethereum/core/state_transition.go

[1] https://github.com/morph-l2/morph

[2] https://github.com/morph-l2/go-ethereum

- go-ethereum/core/token_gas.go
- go-ethereum/rollup/fees/token_info.go
- go-ethereum/rollup/fees/rate.go
- go-ethereum/rollup/fees/rollup_fee.go
- go-ethereum/rollup/fees/token_transfer.go
- go-ethereum/signer/core/apitypes/types.go
- go-ethereum/core/vm/contracts.go
- go-ethereum/crypto/secp256r1/verifier.go
- go-ethereum/core/vm/eips.go
- go-ethereum/core/vm/jump_table.go
- go-ethereum/core/vm/opcodes.go

Other files are not within the scope of the audit. Additionally, all dependencies of the Target are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
morph	`Version 0`	`3cb4687bca674b093a715fc7d328327c35b6e99c`
	`Version 1`	`26233deddf5c77e811a73cfcfb797a2579ed9112`
	`Version 2`	`e64256ee2109d4fcd3f4c6d90aec8abc46a751e7`
go-ethereum	`Version 0`	`62fcaab9b7a732eea298f45d97234d718b522b13`
	`Version 1`	`42d39732bb88bef91c1743efcf10db40ae6b990a`
	`Version 2`	`64e9dcd01e673d9a25efc91090b81f46150cab3c`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the Target, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of the Target.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying

compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan the Target with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the Target and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[3] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[4] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **four** potential security issues. Besides, we have **seven** recommendations and **four** notes.

- High Risk: 1
- Medium Risk: 1
- Low Risk: 2
- Recommendation: 7
- Note: 4

ID	Severity	Description	Category	Status
1	High	Incorrect access control logic in the modifier `onlyAllowed`	Security Issue	Fixed
2	Medium	Inconsistent updates of the price ratio and token scale	Security Issue	Fixed
3	Low	Lack of the value assignment for `FeeLimit` in the `CallArgs` construction	Security Issue	Fixed
4	Low	The misleading return value of the function `getTokenInfo()`	Security Issue	Fixed
5	-	Revise the unused function `Filter()`	Recommendation	Confirmed
6	-	Remove redundant code	Recommendation	Fixed
7	-	Add non-zero checks	Recommendation	Fixed
8	-	Revise the improper error in the function `calculateTokenAmount()`	Recommendation	Fixed
9	-	Revise typos and improper annotations	Recommendation	Fixed
10	-	Unify the existence checks for the balance slot	Recommendation	Fixed
11	-	Use different custom errors for different revert conditions	Recommendation	Fixed
12	-	Ensure the correctness of fee tokens	Note	-
13	-	Potential centralization risks	Note	-
14	-	Openzeppelin upgrade migration risks	Note	-
15	-	Correct handling of the fee tokens in the contract `L2TxFeeVault`	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Incorrect access control logic in the modifier `onlyAllowed`

Severity High

Status Fixed in `Version 2`

Introduced by Version 1

Description In the contract L2TokenRegistry, the functions updatePriceRatio(), batchUpdatePrices(), and updateTokenScale() are protected by the modifier onlyAllowed. However, the design of the modifier onlyAllowed is incorrect, causing these functions to become publicly accessible. Specifically, when the variable allowListEnabled is set to false, the check (i.e., allowListEnabled && !allowList[msg.sender] && msg.sender != owner()) in the modifier onlyAllowed becomes ineffective. As a result, anyone can update the price ratio and token scale. Since the morph chain node directly uses the token information registered in the contract L2TokenRegistry to calculate the amount of required fee tokens, allowing arbitrary updates to these values may lead to denial-of-service (DoS) issues or fund loss.

```
47    modifier onlyAllowed() {
48        if (allowListEnabled && !allowList[msg.sender] && msg.sender != owner()) {
49            revert CallerNotAllowed();
50        }
51        _;
52    }
```

Listing 2.1: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

```
362    function updatePriceRatio(uint16 _tokenID, uint256 _newPrice) external onlyAllowed {
```

Listing 2.2: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

```
378    function batchUpdatePrices(uint16[] memory _tokenIDs, uint256[] memory _prices) external
           onlyAllowed {
```

Listing 2.3: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

```
476    function updateTokenScale(uint16 _tokenID, uint256 _newScale) external onlyAllowed {
```

Listing 2.4: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

Impact The incorrect logic in the modifier allows arbitrary updates for the price ratio and token scale, leading to potential DoS issues or fund loss.

Suggestion Revise the modifier onlyAllowed accordingly.

2.1.2 Inconsistent updates of the price ratio and token scale

Severity Medium

Status Fixed in Version 2

Introduced by Version 1

Description In the morph chain , the calculation of the fee token amount depends on the price ratio (i.e., rate) and token scale (i.e., scale) configured in the contract L2TokenRegistry. Moreover, the annotation (at line 425) of the contract L2TokenRegistry indicates that the price ratio is determined based on the token scale. However, in the contract L2TokenRegistry, the price ratio and token scale are updated via different functions (i.e., the functions updatePriceRatio(), batchUpdatePrices(), updateTokenInfo() or updateTokenScale()). These inconsistent updates

may lead to incorrect calculation of the fee token amount in the morph chain. As a result, the incorrect amount calculation may lead to potential DoS issues or loss of fees. Additionally, the inconsistent update for the variable `tokenAddress` (via the function `updateTokenInfo()`) could lead to the same impact as well.

```
426 if st.msg.FeeTokenID() != 0 {
427   active, err := fees.IsTokenActive(st.state, st.msg.FeeTokenID())
428   if err != nil {
429     return fmt.Errorf("get token status failed %v", err)
430   }
431   if !active {
432     return fmt.Errorf("token %v not active", st.msg.FeeTokenID())
433   }
434   feeRate, tokenScale, err := fees.TokenRate(st.state, st.msg.FeeTokenID())
435   if err != nil {
436     return fmt.Errorf("get token rate failed %v", err)
437   }
438   if feeRate == nil || tokenScale == nil || feeRate.Sign() <= 0 || tokenScale.Sign() <= 0 {
439     return fmt.Errorf("token rate or scale is nil")
440   }
441   st.feeRate = feeRate
442   st.tokenScale = tokenScale
443   return st.buyAltTokenGas()
```

Listing 2.5: go-ethereum/core/state_transition.go

```
13func TokenRate(state StateDB, tokenID uint16) (*big.Int, *big.Int, error) {
14  if tokenID == 0 {
15    return nil, nil, errors.New("token id 0 not support")
16  }
17  info, rate, err := GetTokenInfoFromStorage(state, TokenRegistryAddress, tokenID)
18  if err != nil {
19    log.Error("Failed to get token info from storage", "tokenID", tokenID, "error", err)
20    return nil, nil, err
21  }
22
23  // If token address is zero, this is not a valid token
24  if info.TokenAddress == (common.Address{}) {
25    log.Error("Invalid token address", "tokenID", tokenID)
26    return nil, nil, err
27  }
28
29  // If price is nil or zero, this token doesn't have a valid price
30  if rate == nil || rate.Sign() == 0 {
31    log.Error("Invalid token price", "tokenID", tokenID, "tokenAddr", info.TokenAddress.Hex())
32    return nil, nil, err
33  }
34
35  // Get scale from token info
36  scale, err := GetTokenScaleByIDWithState(state, tokenID)
```

Listing 2.6: go-ethereum/rollup/fees/rate.go

```
255   function updateTokenInfo(
256       uint16 _tokenID,
257       address _tokenAddress,
258       bytes32 _balanceSlot,
259       bool _needBalanceSlot,
260       bool _isActive,
261       uint256 _scale
262   ) external onlyOwner nonReentrant {
263       // Check if token exists
264       if (tokenRegistry[_tokenID].tokenAddress == address(0)) revert TokenNotFound();
265
266       // Check new information
267       if (_tokenAddress == address(0)) revert InvalidTokenAddress();
268
269       // Prevent address being shared across different tokenIDs
270       uint16 existing = tokenRegistration[_tokenAddress];
271       if (existing != 0 && existing != _tokenID) revert TokenAlreadyRegistered();
272
273       // Get decimals from contract
274       uint8 decimals = 18; // Default value
275       try IERC20Infos(_tokenAddress).decimals() returns (uint8 v) {
276           decimals = v;
277       } catch {
278           // If call fails, use default value 18
279       }
280       // Update registration information
281       // Note: balanceSlot is stored as actualSlot + 1 if needBalanceSlot is true, otherwise 0
282       address oldAddress = tokenRegistry[_tokenID].tokenAddress;
283       tokenRegistry[_tokenID] = TokenInfo({
284           tokenAddress: _tokenAddress,
285           balanceSlot: _toStoredBalanceSlot(_balanceSlot, _needBalanceSlot),
286           isActive: _isActive,
287           decimals: decimals,
288           scale: _scale
289       });
```

Listing 2.7: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

```
362   function updatePriceRatio(uint16 _tokenID, uint256 _newPrice) external onlyAllowed {
363       // Check if token exists
364       if (tokenRegistry[_tokenID].tokenAddress == address(0)) revert TokenNotFound();
365
366       if (_newPrice == 0) revert InvalidPrice();
367
368       priceRatio[_tokenID] = _newPrice;
369
370       emit PriceRatioUpdated(_tokenID, _newPrice);
371   }
```

Listing 2.8: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

```
378   function batchUpdatePrices(uint16[] memory _tokenIDs, uint256[] memory _prices) external
          onlyAllowed {
```

```
379         if (_tokenIDs.length != _prices.length) revert InvalidArrayLength();
380
381         for (uint256 i = 0; i < _tokenIDs.length; i++) {
382             if (tokenRegistry[_tokenIDs[i]].tokenAddress == address(0)) continue;
383             if (_prices[i] == 0) continue;
384
385             priceRatio[_tokenIDs[i]] = _prices[i];
386             emit PriceRatioUpdated(_tokenIDs[i], _prices[i]);
387         }
388     }
```

Listing 2.9: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

Impact The inconsistent update for the token information may lead to potential DoS issues or loss of fees.

Suggestion Revise the code accordingly.

2.1.3 Lack of the value assignment for `FeeLimit` in the `CallArgs` construction

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `transaction_args.go`, the function `setDefaults()` fills in default values for unspecified fields of users' transactions. As part of this process, the function `setDefaults()` invokes the function `DoEstimateGas()` to estimate the potential gas usage (i.e., `args.Gas`). However, the user-specified fee limit (i.e., `args.FeeLimit`) is not propagated into the struct `callArgs`, which is passed to the function `DoEstimateGas()` for the estimation. The gas estimation logic is performed based on the user's balance of the alternative fee token, which may exceed the user-specified fee limit (i.e., `args.FeeLimit`). As a result, the gas estimation may fail to reject invalid transactions, as the user-specified fee limit is not taken into account.

```
161 // Estimate the gas usage if necessary.
162 if args.Gas == nil {
163   // These fields are immutable during the estimation, safe to
164   // pass the pointer directly.
165   data := args.data()
166   callArgs := TransactionArgs{
167     From:                 args.From,
168     To:                   args.To,
169     GasPrice:             args.GasPrice,
170     MaxFeePerGas:         args.MaxFeePerGas,
171     MaxPriorityFeePerGas: args.MaxPriorityFeePerGas,
172     FeeTokenID:           args.FeeTokenID,
173     Value:                args.Value,
174     Data:                 (*hexutil.Bytes)(&data),
175     AccessList:           args.AccessList,
176   }
177   pendingBlockNr := rpc.BlockNumberOrHashWithNumber(rpc.PendingBlockNumber)
178   estimated, err := DoEstimateGas(ctx, b, callArgs, pendingBlockNr, b.RPCGasCap())
179   if err != nil {
```

```
180     return err
181   }
182   args.Gas = &estimated
183   log.Trace("Estimate gas usage automatically", "gas", args.Gas)
184 }
```

Listing 2.10: go-ethereum/internal/ethapi/transaction_args.go

```
1204    limit := altBalance
1205    if args.FeeLimit != nil && args.FeeLimit.ToInt().Sign() > 0 {
1206      limit = math.BigMin(altBalance, args.FeeLimit.ToInt())
1207    }
```

Listing 2.11: go-ethereum/internal/ethapi/api.go

Impact The gas estimation may fail to reject invalid transactions, as the user-specified fee limit is not taken into account.

Suggestion Pass the user-specified fee limit into the gas estimation process.

2.1.4 The misleading return value of the function `getTokenInfo()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `L2TokenRegistry`, managers can register fee tokens with an existing balance slot by providing the token's valid balance slot (i.e., `_balanceSlot`) along with the boolean flag `_needBalanceSlot`. When `_needBalanceSlot` is set to `true`, the provided balance slot is further processed (i.e., incremented by one via the function `_toStoredBalanceSlot()`) to distinguish it from a non-exiting balance slot (i.e., `bytes(0)`). For example, if the valid balance slot is `bytes(0)`, it is converted and stored as `bytes(1)`. However, in the function `getTokenInfo()`, the stored balance slot of a registered token is converted to its actual value and returned to users. As a result, when the stored balance slot is `bytes(1)`, users cannot distinguish it from the absence of a balance slot, making the existence of the balance slot unclear.

```
234       tokenRegistry[_tokenID] = TokenInfo({
235           tokenAddress: _tokenAddress,
236           balanceSlot: _toStoredBalanceSlot(_balanceSlot, _needBalanceSlot),
237           isActive: false,
238           decimals: decimals,
239           scale: _scale
240       });
```

Listing 2.12: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

```
175   function _toStoredBalanceSlot(bytes32 _actualSlot, bool _needBalanceSlot) internal pure
          returns (bytes32) {
176       if (!_needBalanceSlot) {
177           return bytes32(0); // Don't store balanceSlot
178       }
179       if (_actualSlot == bytes32(type(uint256).max)) revert InvalidBalanceSlot();
```

```
180         bytes32 storedSlot;
181         assembly {
182             storedSlot := add(_actualSlot, 1)
183         }
184         return storedSlot;
185     }
```

Listing 2.13: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

```
445     function getTokenInfo(uint16 _tokenID) external view returns (TokenInfo memory) {
446         if (tokenRegistry[_tokenID].tokenAddress == address(0)) revert TokenNotFound();
447
448         TokenInfo memory info = tokenRegistry[_tokenID];
449         // Convert stored balanceSlot to actual value
450         info.balanceSlot = _toActualBalanceSlot(info.balanceSlot);
451
452         return info;
453     }
```

Listing 2.14: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

```
192     function _toActualBalanceSlot(bytes32 _storedSlot) internal pure returns (bytes32) {
193         if (_storedSlot == bytes32(0)) {
194             return bytes32(0); // No balanceSlot stored
195         }
196         bytes32 actualSlot;
197         assembly {
198             actualSlot := sub(_storedSlot, 1)
199         }
200         return actualSlot;
201     }
```

Listing 2.15: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

Impact Users cannot distinguish the existence of the balance slot via the function `getTokenInfo()`.

Suggestion Revise the code accordingly.

2.2 Recommendation

2.2.1 Revise the unused function `Filter()`

Status Confirmed

Introduced by `Version 1`

Description In the file `tx_list.go`, the function `Filter()` is intended to remove all transactions from the transaction list (i.e., `l`) based on the provided thresholds (i.e., the inputs `costLimit`, `gasLimit`, and `altCostLimit`). Specifically, at line 388, the function `Filter()` directly returns `nil, nil`, indicating that no transactions should be removed if both thresholds `costLimit` and `gasLimit` are satisfied.

However, the check at line 388 is insufficient because it does not validate the threshold `altCostLimit` before returning `nil, nil`. As a result, transactions exceeding the threshold

`altCostLimit` will not be removed, potentially allowing invalid transactions to remain in the transaction list.

Additionally, the current codebase does not appear to use the function `Filter()`. It is recommended to revise the unused function `Filter()`.

Suggestion Revise the unused function `Filter()` accordingly.

2.2.2 Remove redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are several redundant codes in the project. It is recommended to remove them for better code readability. Specifically, the following code should be removed or revised.

1. Redundant value assignments.

```
35    bool public allowListEnabled = true;
```

Listing 2.16: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

2. Redundant error definitions.

```
68    error InvalidPercent();
```

Listing 2.17: morph/contracts/contracts/l2/system/IL2TokenRegistry.sol

```
72    error AlreadyInitialized();
```

Listing 2.18: morph/contracts/contracts/l2/system/IL2TokenRegistry.sol

3. Redundant query logic for the variable `scale`.

In the function `TokenRate()` of the file `rate.go`, the invocation of the `GetTokenInfoFromStorage()` function is redundant because the token scale (i.e., `scale`) can be obtained directly from the variable `info`.

```
13func TokenRate(state StateDB, tokenID uint16) (*big.Int, *big.Int, error) {
14  if tokenID == 0 {
15    return nil, nil, errors.New("token id 0 not support")
16  }
17  info, rate, err := GetTokenInfoFromStorage(state, TokenRegistryAddress, tokenID)
18  if err != nil {
19    log.Error("Failed to get token info from storage", "tokenID", tokenID, "error", err)
20    return nil, nil, err
21  }
22
23  // If token address is zero, this is not a valid token
24  if info.TokenAddress == (common.Address{}) {
25    log.Error("Invalid token address", "tokenID", tokenID)
26    return nil, nil, err
27  }
28
29  // If price is nil or zero, this token doesn't have a valid price
30  if rate == nil || rate.Sign() == 0 {
31    log.Error("Invalid token price", "tokenID", tokenID, "tokenAddr", info.TokenAddress.Hex())
```

```
32    return nil, nil, err
33  }
34
35  // Get scale from token info
36  scale, err := GetTokenScaleByIDWithState(state, tokenID)
37  if err != nil {
38    log.Error("Failed to get token scale", "tokenID", tokenID, "error", err)
39    return nil, nil, err
40  }
41  if scale == nil || scale.Sign() == 0 {
42    log.Error("Invalid token scale", "tokenID", tokenID, "tokenAddr", info.TokenAddress.Hex())
43  }
```

Listing 2.19: go-ethereum/rollup/fees/rate.go

4. Redundant `nil` checks for the variable `costcap`.

In the file `token_fee.go`, the function `Eth()` will not return `nil`. Therefore, the check `l.costcap.Eth() == nil` in the file `tx_list.go` is redundant.

```
359  if ethCost != nil && ethCost.Sign() > 0 {
360    if l.costcap.Eth() == nil || l.costcap.Eth().Cmp(ethCost) < 0 {
361      l.costcap.SetEthAmount(ethCost)
362    }
363  }
```

Listing 2.20: go-ethereum/core/tx_list.go

```
19func (dca *SuperAccount) Eth() *big.Int {
20  if dca.ethAmount == nil {
21    return new(big.Int)
22  }
23  return dca.ethAmount
24}
```

Listing 2.21: go-ethereum/core/types/token_fee.go

5. Redundant inputs in the function `GetTokenInfoFromStorage()`.

In the file `token_info.go`, the function `GetTokenInfoFromStorage()` has an input `contractAddr`, but this parameter is always assigned the global variable `TokenRegistryAddress` in all invocations. Therefore, the input `contractAddr` of the function `GetTokenInfoFromStorage()` is redundant.

```
171func GetTokenInfoFromStorage(state StateDB, contractAddr common.Address, tokenID uint16) (*
       TokenInfo, *big.Int, error) {
172  // Get token info from TokenInfo struct
173  info, err := GetTokenInfo(state, tokenID)
174  if err != nil {
175    return nil, nil, fmt.Errorf("failed to get token info: %v", err)
176  }
177
178  // Get token price from priceRatio mapping
179  price, err := GetTokenPriceByIDWithState(state, contractAddr, tokenID)
180  if err != nil {
181    return nil, nil, fmt.Errorf("failed to get token price: %v", err)
```

```
182 }
183
184 return info, price, nil
185}
```

Listing 2.22: go-ethereum/rollup/fees/token_info.go

Suggestion Remove or revise the redundant code.

2.2.3 Add non-zero checks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `L2TokenRegistry`, it is recommended to add non-zero checks for the input `_scale` in the function `updateTokenInfo()` to prevent potential mis-operations.

```
255   function updateTokenInfo(
256       uint16 _tokenID,
257       address _tokenAddress,
258       bytes32 _balanceSlot,
259       bool _needBalanceSlot,
260       bool _isActive,
261       uint256 _scale
262   ) external onlyOwner nonReentrant {
263       // Check if token exists
264       if (tokenRegistry[_tokenID].tokenAddress == address(0)) revert TokenNotFound();
265
266       // Check new information
267       if (_tokenAddress == address(0)) revert InvalidTokenAddress();
268
269       // Prevent address being shared across different tokenIDs
270       uint16 existing = tokenRegistration[_tokenAddress];
271       if (existing != 0 && existing != _tokenID) revert TokenAlreadyRegistered();
272
273       // Get decimals from contract
274       uint8 decimals = 18; // Default value
275       try IERC20Infos(_tokenAddress).decimals() returns (uint8 v) {
276           decimals = v;
277       } catch {
278           // If call fails, use default value 18
279       }
280       // Update registration information
281       // Note: balanceSlot is stored as actualSlot + 1 if needBalanceSlot is true, otherwise 0
282       address oldAddress = tokenRegistry[_tokenID].tokenAddress;
283       tokenRegistry[_tokenID] = TokenInfo({
284           tokenAddress: _tokenAddress,
285           balanceSlot: _toStoredBalanceSlot(_balanceSlot, _needBalanceSlot),
286           isActive: _isActive,
287           decimals: decimals,
288           scale: _scale
289       });
```

Listing 2.23: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

Suggestion Add non-zero checks for the input `_scale` in the function `updateTokenInfo()`.

2.2.4 Revise the improper error in the function `calculateTokenAmount()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `L2TokenRegistry`, the function `calculateTokenAmount()` reverts with the error `InvalidPrice()` when the variable `tokenAmount` is zero (i.e., at line 435). This error is misleading since the zero token amount can be produced due to the zero `info.scale` or `_ethAmount`. It is recommended to replace the error `InvalidPrice()` with a proper error.

```
417   function calculateTokenAmount(uint16 _tokenID, uint256 _ethAmount) external view returns (
          uint256 tokenAmount) {
418       // Validate: token must be registered
419       if (tokenRegistry[_tokenID].tokenAddress == address(0)) revert TokenNotFound();
420
421       // Get token information
422       TokenInfo memory info = tokenRegistry[_tokenID];
423
424       // Get priceRatio which follows:
425       // ratio = tokenScale * (tokenPrice / ethPrice) * 10^(ethDecimals - tokenDecimals)
426       uint256 ratio = priceRatio[_tokenID];
427       if (ratio == 0) revert InvalidPrice();
428
429       // Calculate token amount with ceiling division:
430       // tokenAmount = ceil((ethAmount * tokenScale) / ratio)
431       // Using formula: ceil(a/b) = (a + b - 1) / b
432       uint256 numerator = _ethAmount * uint256(info.scale);
433       tokenAmount = (numerator + ratio - 1) / ratio;
434
435       if (tokenAmount == 0) revert InvalidPrice();
```

Listing 2.24: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

Suggestion Replace the error `InvalidPrice()` with a proper error in the `calculateTokenAmount()` function.

2.2.5 Revise typos and improper annotations

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are several typos and improper annotations in the codebase. It is recommended to revise them for better readability.

1. In the file `rate.go`, it is recommended to revise the name of the variable `tokenSacle` in the functions `EthToAlt()` and `AltToETH()`.

```
48func EthToAlt(state StateDB, tokenID uint16, amount *big.Int) (*big.Int, error) {
49  rate, tokenSacle, err := TokenRate(state, tokenID)
50  if err != nil {
51    return nil, err
```

```
52 }
53 return types.EthToAlt(amount, rate, tokenSacle), nil
54}
55
56func AltToETH(state StateDB, tokenID uint16, amount *big.Int) (*big.Int, error) {
57 rate, tokenSacle, err := TokenRate(state, tokenID)
58 if err != nil {
59   return nil, err
60 }
61 return types.AltToEth(amount, rate, tokenSacle), nil
62}
```

Listing 2.25: go-ethereum/rollup/fees/rate.go

2. In the file `token_transfer.go`, the annotation at line 62 incorrectly explains the purpose of the following code.

```
62 // Calculate the storage slot for the user's balance
63 state.SetState(tokenAddress, storageSlot, amountHash)
```

Listing 2.26: go-ethereum/rollup/fees/token_transfer.go

3. In the file `receipt.go`, the annotation at line 127 incorrectly indicates the introduction date of the feature.

```
126// v7StoredReceiptRLP is the storage encoding of a receipt used in database version 7.
127// This version was introduced when AltFee feature was added (2024-11).
```

Listing 2.27: go-ethereum/core/types/receipt.go

4. In the contract `L2TokenRegistry`, the annotation for the function `calculateTokenAmount()` is inconsistent with the implemented logic. Specifically, the formula described in the annotation (i.e., at lines 406-409) is incorrect. It is recommended to revise the formula annotation to accurately reflect the implementation.

```
402   /**
403    * @notice Calculate the corresponding token amount for a given ETH amount
404    * @dev Calculation formula:
405    *      - ratio = tokenScale * (tokenPrice / ethPrice) * 10^(ethDecimals - tokenDecimals)
406    *      - tokenAmount = (ethAmount * 10^tokenDecimals) / ratio
407    *      - Substituting ratio: tokenAmount = (ethAmount * 10^tokenDecimals) / (tokenScale * (
           tokenPrice / ethPrice) * 10^(18 - tokenDecimals))
408    *      - Simplified: tokenAmount = (ethAmount * 10^tokenDecimals * 10^tokenDecimals) / (
           tokenScale * tokenPrice * 10^18 / ethPrice)
409    *      - Final: tokenAmount = (ethAmount * ethPrice * 10^tokenDecimals) / (tokenScale *
           tokenPrice * 10^18)
410    *      - Note: Uses ceiling division to ensure users receive fair token amounts
411    * @param _tokenID Token ID of the ERC20 token
412    * @param _ethAmount ETH amount (unit: wei)
413    * @return tokenAmount Corresponding token amount (unit: token's smallest unit)
414    * - ratio follows: ratio = tokenScale * (tokenPrice / ethPrice) * 10^(ethDecimals -
           tokenDecimals)
415    * - Will revert if token is not registered or priceRatio is not set
416    */
```

```
417   function calculateTokenAmount(uint16 _tokenID, uint256 _ethAmount) external view returns (
          uint256 tokenAmount) {
418       // Validate: token must be registered
419       if (tokenRegistry[_tokenID].tokenAddress == address(0)) revert TokenNotFound();
420
421       // Get token information
422       TokenInfo memory info = tokenRegistry[_tokenID];
423
424       // Get priceRatio which follows:
425       // ratio = tokenScale * (tokenPrice / ethPrice) * 10^(ethDecimals - tokenDecimals)
426       uint256 ratio = priceRatio[_tokenID];
427       if (ratio == 0) revert InvalidPrice();
428
429       // Calculate token amount with ceiling division:
430       // tokenAmount = ceil((ethAmount * tokenScale) / ratio)
431       // Using formula: ceil(a/b) = (a + b - 1) / b
432       uint256 numerator = _ethAmount * uint256(info.scale);
433       tokenAmount = (numerator + ratio - 1) / ratio;
434
435       if (tokenAmount == 0) revert InvalidPrice();
436
437       return tokenAmount;
438   }
```

Listing 2.28: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

Suggestion Revise the above typos and annotation inconsistencies accordingly.

2.2.6 Unify the existence checks for the balance slot

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `token_gas.go`, the `GetAltTokenBalanceHybrid()` and `GetAltTokenBalance()` functions perform an existence check for the balance slot in different ways. It is recommended to use the defined function `HasSlot()` in both functions for better readability.

```
25func (st *StateTransition) GetAltTokenBalanceHybrid(tokenID uint16, user common.Address) (*fees.
      TokenInfo, *big.Int, error) {
26  info, err := fees.GetTokenInfo(st.state, tokenID)
27  if err != nil {
28    return nil, nil, err
29  }
30  balance := new(big.Int)
31  if !info.HasSlot {
```

Listing 2.29: go-ethereum/core/token_gas.go

```
60func GetAltTokenBalance(evm *vm.EVM, tokenID uint16, user common.Address) (*big.Int, error) {
61  info, err := fees.GetTokenInfo(evm.StateDB, tokenID)
62  if err != nil {
63    return nil, fmt.Errorf("failed to get token address for token ID %d: %v", tokenID, err)
64  }
```

```
65 balance := new(big.Int)
66 if !bytes.Equal(info.BalanceSlot.Bytes(), common.Hash{}.Bytes()) {
```

Listing 2.30: go-ethereum/core/token_gas.go

Suggestion Unify the existence checks for the balance slot.

2.2.7 Use different custom errors for different revert conditions

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `L2TokenRegistry`, the checks performed at lines 218 and 219 are semantically different but revert with the same error. It is recommended to implement separate custom errors for better readability and easier off-chain tracking.

```
206   function _registerSingleToken(
207       uint16 _tokenID,
208       address _tokenAddress,
209       bytes32 _balanceSlot,
210       bool _needBalanceSlot,
211       uint256 _scale
212   ) internal {
213       // Check token address
214       if (_tokenAddress == address(0)) revert InvalidTokenAddress();
215
216       // Forbid zero ID and enforce uniqueness for both ID and address
217       if (_tokenID == 0) revert InvalidTokenID();
218       if (tokenRegistry[_tokenID].tokenAddress != address(0)) revert TokenAlreadyRegistered();
219       if (tokenRegistration[_tokenAddress] != 0) revert TokenAlreadyRegistered();
```

Listing 2.31: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

Suggestion Define and use distinct custom errors for these two revert conditions in the function `_registerSingleToken()`.

2.3 Note

2.3.1 Ensure the correctness of fee tokens

Introduced by `Version 1`

Description The contract `L2TokenRegistry` allows the role `owner` to register fee tokens and update their corresponding information (e.g., price ratio, scale, address, and decimal). The registered tokens' information stored in the contract `L2TokenRegistry` is directly fetched by the Morph chain to support the alternative fee token feature. Therefore, the project must ensure fee tokens are carefully selected (e.g., fee-on-transfer tokens should not be registered as the fee token) and the corresponding information is correctly configured to avoid potential DoS issues or loss of fees.

2.3.2 Potential centralization risks

Introduced by Version 1

Description In the contract L2TokenRegistry, several privileged roles (e.g., the role owner) can conduct sensitive operations, which introduces potential centralization risks. For example, the role owner can register fee tokens via the function registerToken(). If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

```
114   function registerTokens(
115       uint16[] memory _tokenIDs,
116       address[] memory _tokenAddresses,
117       bytes32[] memory _balanceSlots,
118       bool[] memory _needBalanceSlots,
119       uint256[] memory _scales
120   ) external onlyOwner nonReentrant {
```

Listing 2.32: morph/contracts/contracts/l2/system/L2TokenRegistry.sol

2.3.3 Openzeppelin upgrade migration risks

Introduced by Version 1

Description The contract L2TokenRegistry currently uses OpenZeppelin's contracts Initializable and ReentrancyGuardUpgradeable (v4.9.3) to implement upgradeable contracts. It is important to note that the contracts Initializable and ReentrancyGuardUpgradeable with versions v5.0.0 and later introduce ERC-7201 namespaced storage to mitigate storage collision risks. This change relocates initialization and reentrancy-guard state variables from direct storage slots (e.g., _initialized and status) to namespaced storage structures (e.g., $._initialized and $._status). When upgrading to the newer versions of Initializable and ReentrancyGuardUpgradeable, the project must ensure that the initialization and reentrancy-guard states are migrated correctly. Otherwise, an improper migration may introduce severe security vulnerabilities (e.g., contract being reinitialized or the reentrancy guard being circumvented).

2.3.4 Correct handling of the fee tokens in the contract L2TxFeeVault

Introduced by Version 1

Description To support the alternative fee token feature, the project must ensure that the contract L2TxFeeVault , which is responsible to receive and hold the fee tokens, contains the logic to correctly handle the fee tokens.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS