SECBIT

SECURITY AUDIT REPORT

DeGate Protocol

A Decentralized Exchange(DEX) protocol based on a ZK Rollup

✅ Smart Contracts Code Audit

✅ Groth16 Circuits Code Audit

Final Report

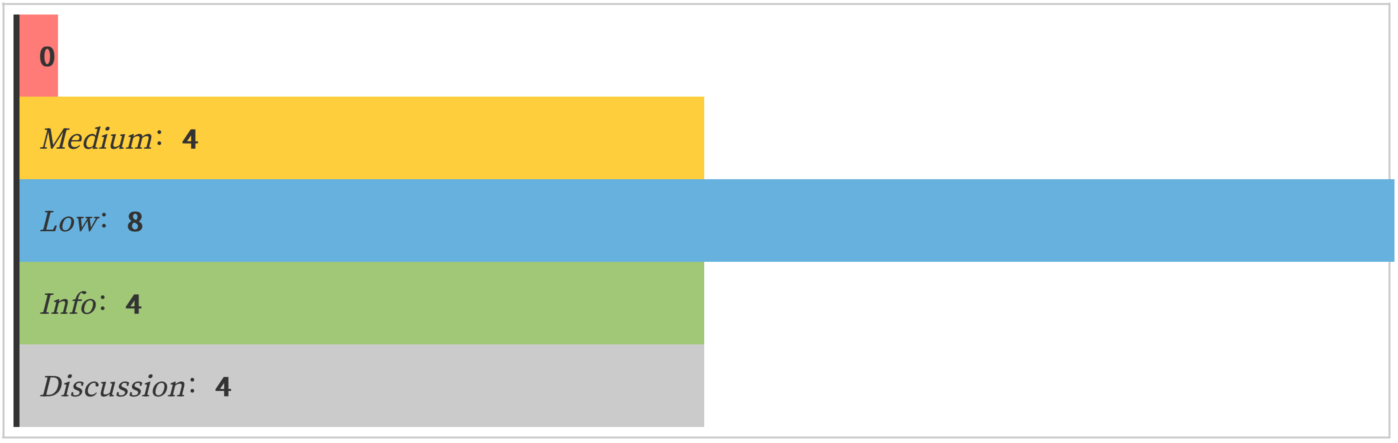
By SECBIT Labs

Nov 15th, 2023

1. Introduction

DeGate is a Decentralized Exchange (DEX) protocol built on Zero Knowledge (ZK) technology. SECBIT Labs conducted an audit of the smart contracts and circuits in 4 areas: **code bugs**, **logic flaws**, **risk assessment** and **code optimization** from Mar 7th, 2022 to Sep 16th, 2022. Then a second audit was conducted from Jan 3rd to Feb 24th, 2023. And a third audit was conducted from Jul 2nd to Aug 25th, 2023. During the period from Oct 27th to Nov 13th, 2023, we conducted the fourth audit, which focused on the "Delayed Upgrade Scheme" (see part 3.4 for further details). The assessment found a few security risks in the DeGate protocol, for which the SECBIT team provided suggestions to minimize the risk (see part 4 for further details).

As shown in the figure below, we found a total of 20 issues. All of them have been fixed and/or responded to (see part 2.4 for further details).



Issues Statistics

Note: Please see Appendix for the glossary of terms on classification of risks.

2. Overview

This section describes the basic information and code structure.

2.1 Basic Information

The following list shows basic information about DeGate:

Name	DeGate
Source	DeGate-protocols
Initial Review Commit	b7c88981ddff78421f51a4e77c3dd8333c7ce197
Final Review Commit(the first audit)	46737e464bba2c456e4fef10befbd9454e55eb39
Review Commit(the second audit)	4264a22c82eff3d9f197b15060bf8a7c18c13427
Review Commit(the third audit)	59dbb6a569488020cc12b19999c7ed2e3e96c365
Review Commit(the fourth audit)	dee9b9f708fbfb176805b58e7bdf0964afab43f2
Lines	circuit: 13767; solidity: 8319
Languages	Solidity; C++
Stage	Completed Development

2.2 Contract List

The following shows the contracts included in DeGate Protocol, which the SECBIT team audited:

Name	Description
DelayedOwner.sol	An Owner contract where certain functions have a mandatory delay for security purposes.
LoopringIOExchangeOwner.sol	Contract used by the Prover to submit blocks with zkSNARK proofs.
SelectorBasedAccessManager.sol	Function selector based access management contract.
FastWithdrawalAgent.sol	Fast withdrawal agent implementation contract.
ExchangeData.sol	Define data struct and key constants.

Name	Description
FastWithdrawalLiquidityProvider.sol	Basic contract storing funds for a liquidity provider.
BlockVerifier.sol	Block verifier contract for circuit proofs.
DefaultDepositContract.sol	Deposit contract, store and transfer funds for exchange.
ExchangeV3.sol	Exchange core contract, includes the libraries to process block and tx.
LoopringV3.sol	Loopring contract for global parameters setting.
ExchangeAdmins.sol	Library implementation of exchange admin features.
ExchangeBalances.sol	Library implementation of exchange balance features.
ExchangeBlocks.sol	Library implementation of exchange block processing features.
ExchangeDeposits.sol	Library implementation of exchange deposits features.
ExchangeGenesis.sol	Library implementation of exchange init genesis block features.
ExchangeMode.sol	Library implementation of exchange mode features.
ExchangeTokens.sol	Library implementation of exchange token mapping features.
ExchangeWithdrawals.sol	Library implementation of exchange withdrawals features.
AccountUpdateTransaction.sol	Utility library to process account update transaction.
BlockReader.sol	Utility library to read block data.
DepositTransaction.sol	Utility library to process deposit transactions.
WithdrawTransaction.sol	Utility library to process withdraw transaction.
ERC1271.sol	ERC1271 interface.
ERC20SafeTransfer.sol	ERC20 safe transfer library.
MathUint248.sol	Utility Functions for uint248.
Poseidon.sol	Poseidon hash function.
BytesUtil.sol	Bytes utility library.
SafeCast.sol	Wrappers over uintXX/intXX casting operators with overflow checks.
Timelock.sol	Delay the execution of functions in a smart contract.

The DeGate smart contracts were developed based on Loopring v3.6.1. This audit mainly focuses on the changed contracts above.

These smart contracts have been deployed on the Ethereum mainnet. We have verified the consistency of the smart contracts by examining their bytecode. The following are the deployed contract names and addresses:

Contract Name	Contract Address
BatchVerifier	0x1453525d5D9AaEc4Fd9BF7FEf485189F98e152C2
BlockVerifier	0xE3B7fE3ce0fa54C5AC7F48E7ED9E52dA045bE4d6
LoopringV3	0x9385aCd9d78dFE854c543294770d0C94c2B07EDC
ExchangeAdmins	0x0a5d144ADF62e18eE222f2D05a2Bf2037ce8EeAe
ExchangeBalances	0xF799e5CEF24528b9409502E99f5837ee3446D11d
ExchangeBlocks	0x4AD92EE2019A6A26c9E38caEDd46503BD7f79C10
ExchangeDeposits	0xe8d1AcFa31A9f133Fe5E05F8eA6A358B56a43937
ExchangeGenesis	0x57f2BAa929EcE41D8B6FDc4c7f7b60B95522AbcB
ExchangeTokens	0x2Bf7021a3Aa041e1a8a5082DB720d0202C70A3aE
ExchangeWithdrawals	0x9FB27470d766A29Ac126CF99eE103087a4072e33
ExchangeV3Imp	0xc56C1dfE64D21A345E3A3C715FFcA1c6450b964b
DefaultDepositContractImp	0x8CCc06C4C3B2b06616EeE1B62F558f5b9C08f973
ExchangeV3Proxy	0x9C07A72177c5A05410cA338823e790876E79D73B
DepositContractProxy	0x54D7aE423Edb07282645e740C046B9373970a168
LoopringIOExchangeOwner	0x9b93e47b7F61ad1358Bd47Cd01206708E85AE5eD

2.3 Circuits List

The following shows the circuit source files for aduiting included in DeGate Protocol:

Name	Description
Circuits/UniversalCircuit.h	Outer layer circuits. It contains the structure of outermost layer circuits.
Circuits/Circuit.h	Base circuits for UniversalCircuit.
Circuits/AccountUpdateCircuit.h	Circuits of AccountUpdate transaction.
Circuits/AppKeyUpdateCircuit.h	Circuits of AppKeyUpdate transaction.
Circuits/BaseTransactionCircuit.h	Base circuits for transactions.

Name	Description
Circuits/BatchSpotTradeCircuit.h	Circuits of BatchSpotTrade transaction.
Circuits/DepositCircuit.h	Circuits of Deposit transaction.
Circuits/NoopCircuit.h	Circuits of Noop transaction.
Circuits/OrderCancelCircuit.h	Circuits of OrderCancel transaction.
Circuits/SpotTradeCircuit.h	Circuits of SpotTrade transaction.
Circuits/TransferCircuit.h	Circuits of Transfer transaction.
Circuits/WithdrawCircuit.h	Circuits of Withdraw transaction.
Circuits/AccountGadgets.h	Gadgets for Circuits. It contains some gadgets used by AccountCircuit.
Gadgets/BatchOrderGadgets.h	Gadgets for Circuits. It contains some gadgets for batch orders
Gadgets/MatchingGadgets.h	Gadgets for Circuits. It contains some gadgets for order matching
Gadgets/MathGadgets.h	Gadgets for Circuits. It contains some gadgets for math operation.
Gadgets/MerkleTree.h	Gadgets for Circuits. It contains some gadgets for the Merkle Tree.
Gadgets/OrderGadgets.h	Gadgets for Circuits. It contains some gadgets for the orders.
Gadgets/SignatureGadgets.h	Gadgets for Circuits. It contains some gadgets for signature.
Gadgets/StorageGadgets.h	Gadgets for Circuits. It contains some gadgets for merkle tree.
Utils/Constants.h	Utils for Circuits. It contains some constants used in the circuits.
Utils/Data.h	Utils for Circuits. It contains some dummy data and functions for parsing json format data.
Utils/Utils.h	Utils for Circuits. It contains functions for assertion, logs and format conversions.

2.4 Finding

The following shows the findings on issues in the circuit codes and contract codes:

No	Issue Title	Type	Level	fixed
1	「circuit」 Insufficient check for the noop order	Security Risk	Medium	✅
2	「circuit」 Invalid logic for transfer verification	Security Risk	Medium	✅
3	「circuit」 Insufficient range constraints	Security Risk	Medium	✅
4	「circuit」 Insufficient data for data availability	Security Risk	Low	🚫
5	「circuit」 Unchecked Consistency for txType	Security Risk	Low	✅
6	「circuit」 Invalid length check	Potential Risk	Low	✅
7	「circuit」 Redundant independent witnesses	Code Optimization	Low	✅
8	「circuit」 Too large NUM_BITS_AUTOMARKET_LEVEL	Potential Risk	Info	🚫
9	「circuit」 Uncleaned code	Code Optimization	Info	🚫
10	「circuit」 Reduce hash circuits	Code Optimization	Discussion	✅
11	「circuit」 Permission risk for appKey	Potential Risk	Discussion	🚫
12	「circuit」 Redundant checks for forwarding	Code Optimization	Discussion	🚫
13	「contract」 Inconsistent logic about `isconditional` for `UpdateAccount`	Logical Implementation	Medium	✅
14	「contract」 There is a DOS risk that may make it impossible to submit blocks properly	Security Risk	Low	🚫
15	「contract」 The `blockVerifier` parameter cannot be updated properly	Potential Risk	Low	✅
16	「contract」 Unable to return user assets under special circumstances	Potential Risk	Low	✅
17	「contract」 Unmodified withdraw logic for protocol account	Potential Risk	Low	🚫
18	「contract」 `loopringAddr` is redundant	Code Optimization	Info	✅
19	「contract」 `withdrawExchangeFees` needs to enhance the checking of token parameters	Code Optimization	Info	🚫
20	「contract」 Is `submitBlocksWithCallbacks` missing the necessary validation?	Code Optimization	Discussion	✅

3. Project Analysis

This section describes a detailed analysis of the content of DeGate code in four parts: function implementation, circuits analysis, contracts analysis, and the Delayed Upgrade Scheme analysis.

3.1 Feature Analysis

DeGate supports a variety of functions, including Spot Trading, Grid Trading, etc., and with more features to be added to enable a coherent trading experience.

- Spot Trading:

  Conventional AMM DEXes incur high gas fees on Ethereum and provide only market orders, where traders have to accept the current market price for a trading pair. DeGate allows for spot trading through limit orders, similar in experience to a centralized exchange.

- Grid Trading:

  This replicates the grid trading on a CEX, which enables users to implement trading strategies based on price movement in any particular trading pair. This feature can help users earn long-term and stable returns safely in the highly volatile cryptocurrency market without the assets being custodied by a centralized entity. Data availability for all grid strategies on DeGate is secured by Ethereum through zero-knowledge technology.

- Deposit: The user deposits tokens from layer1 to layer2.
- Withdraw: The user withdraws tokens from layer2 to layer1.
- Transfer: The user transfers some tokens to another layer2 account.
- Update the public key and owner information of the user

  Any user needs to use layer2's public key to generate a signature for his transaction, so the user can update the public key and other information for his account.

- Update the generated "appKey" and its permission switch of user

  AppKey is a pair of keys for traders, and any user can update his Appkey.

3.2 Contract Analysis

There are several key roles in the protocol: Protocol Owner, Exchange Owner, Exchange User, and Normal User.

- Protocol Owner
    - Description

      Protocol administrator
    - Authority
        - Update settings of protocol
        - Update fee settings
    - Method of Authorization
        - The contract deployer automatically becomes the `owner`
        - Authorized by transferring the ownership of the contract
- Exchange Owner
    - Description

      Exchange administrator, also known as the exchange operator
    - Authority
        - Register Tokens
        - Withdraw the assets staked by the exchange owner
        - Withdraw exchange fees
        - withdraw protocol fees
        - Handling assets that were mistakenly transferred to the exchange address
        - Set deposit params
        - Submit blocks
        - Shutdown the exchange after properly processing the users' assets
    - Method of Authorization
        - Determined at the time of deployment
        - Authorized by transferring the ownership of the contract
- Exchange User
    - Description

      Users who created accounts in the exchange
    - Authority
        - Account registration and updating
        - Deposit and withdraw
        - Register Tokens

    - Method of Authorization

      User registered through the exchange contract
- Normal User
    - Description

      Normal Ethereum account
    - Authority

      Execute other operations allowed by the contract
    - Method of Authorization

      No authorization required

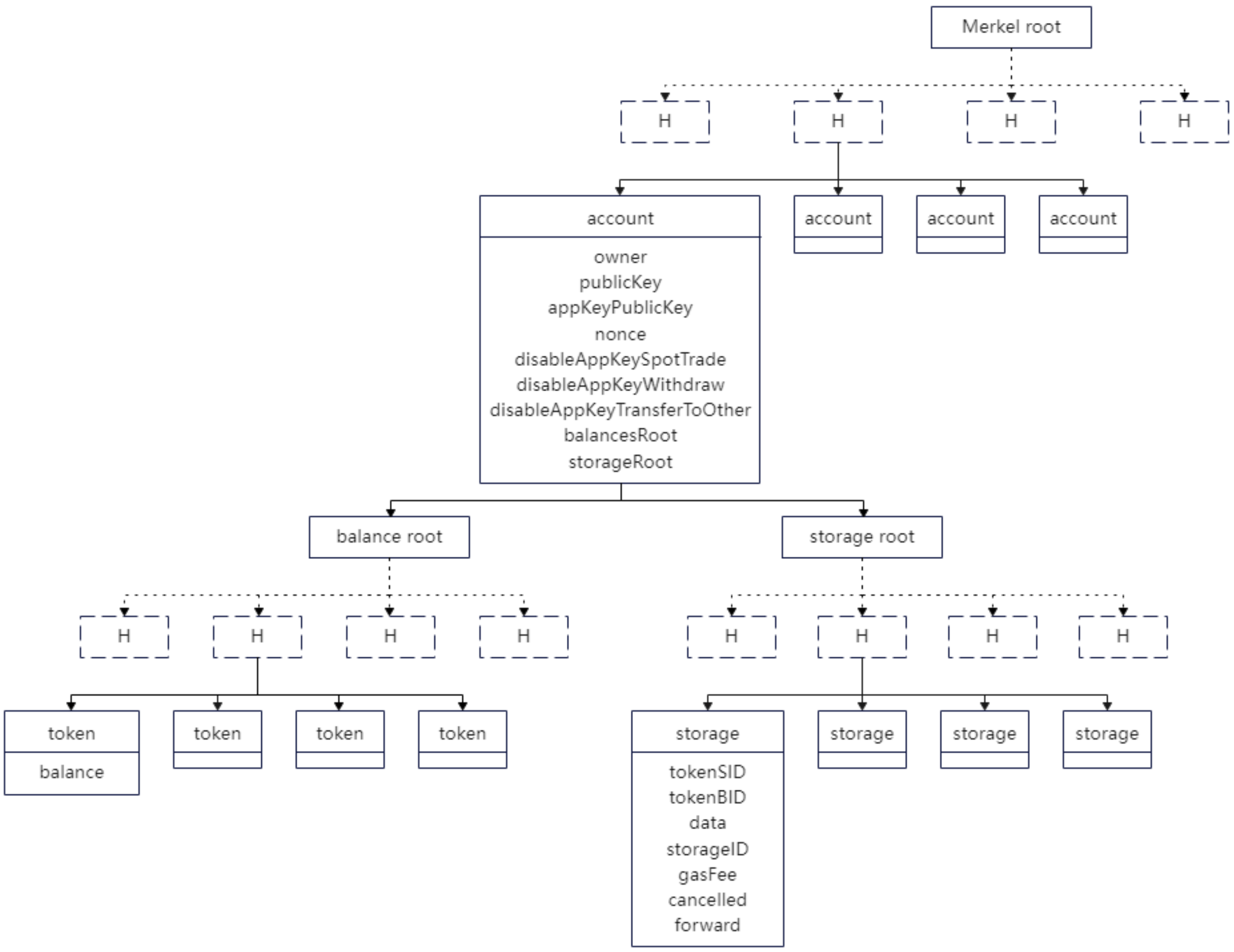
`ExchangeV3.sol` is the main entry contract and its core features include the following.

- `deposit()`
  The user deposits to exchange by calling the contract. The exchange owner/operator must process the request within a certain period and package it into the block for submission.
- `submitBlocks()`
  The exchange owner/operator is responsible for submitting blocks to the contract. Blocks will also be committed and verified in this function. Smart contracts will only verify the necessary logic, while zero-knowledge proofs perform more complex validation to ensure the correctness of contract state changes. The smart contract will process specific types of transactions in blocks conditionally.
- `forceWithdraw()`
  Usually, the user initiates a withdrawal request to exchange admin off-chain and waits for the admin to process it. It is possible to force the admin/operator to process a withdrawal for the complete balance in an account. It is done by doing a withdrawal request on-chain using `forceWithdraw()`.
- `withdrawFromDepositRequest()`
  Deposits not yet included in a submitted block can be withdrawn (even when not in withdrawal mode after some time) using `withdrawFromDepositRequest()`.
- `withdrawFromMerkleTree()`
  Exchange can go into withdrawal mode when a forced on-chain request is open for longer than a specific period. Users can withdraw balances stored in the Merkle tree with a Merkle proof by calling `withdrawFromMerkleTree()`.

3.3 Circuits Analysis

DeGate layer 2 uses the Merkle tree to store all permanent data required in the circuit. The Merkle tree is divided into two layers. The first layer is the account tree. The leaf node of each account tree represents the information of an account. Under each leaf node, there are two Merkle trees on the second layer, namely balance tree and storage tree. The balance tree stores all balance information of accounts, and the storage tree stores intermediate process information of transactions.

The structure of the Merkle tree is as follows:



- In the account tree, each account has an accountId, which represents the index of the leaf node where the account is located. The fields stored in the account have the following meanings:
    - owner: layer 1 address corresponding to the account.
    - publicKey: The public key of the account used for signature in layer 2.

    - appKey: The public key used by third-party traders. This public key pair is derived from the user's layer 1 private key, and the user can update the appkey at any time. Holding appkey can initiate orders, redemption and transfer transactions for account assets. The user decides whether to open the permission. "disableAppKeySpotTrade", "disableAppKeyWithdraw", "disableAppKeyTransferToOther" are the setting switch.
    - nonce: To prevent replay attacks, the nonce field must be +1 calculated for some transaction types.
    - tokenRoot: Root of balance tree.
    - storageRoot: Root of the storage tree.
- In the balance tree, each leaf node represents the balance information of a token type held by the user in the layer 2 account. Each token has a tokenId, which represents the index of the leaf node where the token is located. The leaf stores only one field, token balance.
- In the storage tree, each leaf node represents the status information of an order or a transaction (withdraw, transfer). Since the state information data does not need to be permanently saved, the leaf nodes of the storage tree can be reused. Each status information has a storageId. The operator assigns a storageId to each order or transaction (withdraw, transfer) of the user in order. The information is saved in the position of index equal to "storageId % 4^7".

The outermost layer of the DeGate circuit is UniversalCircuit, which is a complete circuit for processing a block:

- Verify whether the update of Merkle root is correct.
- Verify that the update of the operator account is correct.
- Verify whether the parameters are legal.
- Verify whether all types of transactions are legal.
- Check the transaction order and calculate the quantity of each type of transaction: to optimize the uploaded data, transactions must be sorted in the order of "deposit -> updateaccount -> others -> withdraw".
- Verify the EdDSA signature of layer 2.
- Hash public inputs: because public inputs are very long and difficult to handle in the contract, all public inputs are hashed in the circuit, and the hash is taken as the real public inputs. The original public inputs are changed to witness. It can save the gas of the contracts.

The DeGate circuit needs to process each transaction separately. The processing of each transaction should include the logic of all transaction types, then select the required results, and update the account tree, balance tree and storage tree according to those results.

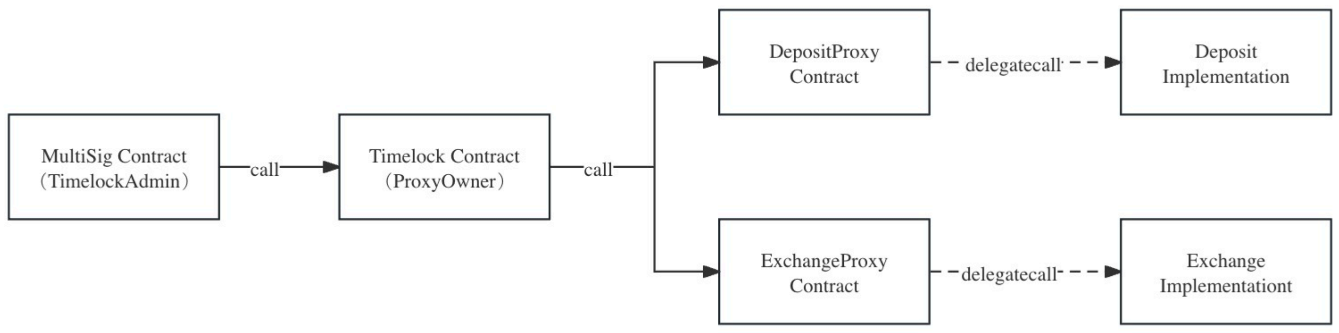
- AccountUpdate transaction: It is used to update the user's key and owner information.
    - A new owner can be written and updated only when the owner is empty. Because the relationship between layer 2 accounts and layer 1 accounts can only be bound once.

    - The user passes the new key to the operator and attaches a signature. The signature can be signed with a Layer-1 private key (ECDSA). When setting the key for the first time, you can only sign with the private key of layer 1.
- AppKeyUpdate transaction: used to update the user's appKey and Its three permission for appKey stored in his account leaf node.
- Deposit transaction: process user deposits. The operator listens to the deposit transactions on chain, and then get the data to construct a deposit transaction by himself for circuit verification. Then the user's assets will be transferred to the layer2 account from layer1.
- Withdraw transaction: process user refunds. There are 4 types of withdrawal. The user can initiate a withdrawal transaction from layer1 or layer2. If he sends a transaction to layer2, the operator gets his transaction with a signature. If he sends a transaction to Layer 1, the operator listens to the transactions from layer1 and reads the data to contract a withdrawal transaction by himself for circuit verification.
- Transfer transaction: Layer2 account A transfers tokens directly to the layer2 account B.
- SpotTrade transaction: matching two orders.
    - DeGate introduces Grid Trading. When it is judged as a grid order according to the "type" field in the transaction, the circuit will turn to verify the signature of the "startOrder" corresponding to the grid order. The order itself only verifies its relationship with the "startOrder" according to the rules of Grid Trading.
- BatchSpotTrade transaction: match multiple orders of up to 6 users. Set the number of orders that each user can participate in matching as 4, 2, 1, 1, 1, 1 respectively.
    - There are three kinds of tokens involved. The circuits will verify the legitimacy of the order, and then ensure that the expenditure and income of each order meet the order price limit. At the same time, it will ensure the cumulative revenue and expenditure balance of all currencies of all users at the outermost layer.
- OrderCancel transaction: set the "cancelled" field of the storage leaf bound to the order to 1, which means that the order can no longer be matched.
- Noop transaction: the circuit block contains a fixed number of transactions, so it sometimes needs to be filled with empty transactions.

3.4 Delayed Upgrade Scheme Analysis

DeGate uses the Delayed Upgrade Scheme to ensure business continuity and security, achieving the highest level of trustlessness. The details are as follows: Firstly, DeGate uses the Proxy Upgrade Pattern to achieve contract upgradability. It can increase the smart contract system's flexibility, security, and maintainability. Then, DeGate utilizes the Timelock contract to delay the deployment of upgradable contracts (more than 30 days). The authority of the related contracts is strictly restricted. These mechanisms ensure that upgradable contracts possess the same level of trustlessness as immutable contracts. Users have ample time to manage their funds before the completion of the contract upgrade.

The structure of contracts in the scheme is as follows:



The contract upgrade process involves the following steps: First, initiate a multisignature proposal by one of the management addresses of the Multisig contract. Use the Multisig contract to store the transactions for the upgrade in the Timelock contract. Then, wait for a sufficient delay period. Finally, initiate another multisignature proposal by one of the management addresses of the Multisig contract. Execute the transactions stored in the Timelock contract. The contract upgrade is completed. Only the Deposit Implementation (DefaultDepositContract.sol) and Exchange Implementation (ExchangeV3.sol) can be upgraded, and the upgrade delay is set to more than 30 days.

There are two key roles in the scheme: Proxy Owner and Timelock Admin.

Proxy Owner

- Description

  The owner of Deposit Proxy contract and Exchange Proxy contract. In the future, it may use two separate Timelock contracts for management.
- Authority
  - Upgrade the implementation of the proxy
- Method of Authorization
  - Authorized by transferring the ownership of the contract

Timelock Admin

- Description

  The admin of Timelock contract
- Authority
  - Store future transactions to be executed in the Timelock contract
  - Execute transactions that meet the delay requirements
- Method of Authorization
  - Authorized by setting the admin of the contract

4. Audit Detail

This section describes the process and detailed results of the audit also demonstrates the problems and potential risks.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bugs, logical implementation potential risks and code optimization. The process consists of four steps:

- Full analysis of codes line by line.
- Evaluation of vulnerabilities and potential risks revealed in the code.
- Risk assessment and confirmation.
- Audit report writing.

4.2 Audit Result

Combining the results from scanning tools and manual assessments. The team inspected the code line by line and the results can be categorized into twenty types for smart contracts, fifteen types for circuits, and nine types for the Delayed Upgrade Scheme.

4.2.1 Results for smart contract

Number	Classification	Result
1	Normal functioning of features defined by the contract	✓
2	No obvious bug (e.g. overflow, underflow)	✓
3	Pass Solidity compiler check with no potential error	✓

Number	Classification	Result
4	Pass common tools check with no obvious vulnerability	✓
5	No obvious gas-consuming operation	✓
6	Meet with ERC20	✓
7	No risk in low level call (call, delegatecall, callcode) and in-line assembly	✓
8	No deprecated or outdated usage	✓
9	Explicit implementation, visibility, variable type and Solidity version number	✓
10	No redundant code	X
11	No potential risk manipulated by timestamp and network environment	✓
12	Explicit business logic	✓
13	Implementation consistent with annotation and other info	✓
14	No hidden code about any logic that is not mentioned in design	✓
15	No ambiguous logic	✓
16	No risk threatening the developing team	✓
17	No risk threatening exchanges, wallets and DApps	✓
18	No risk threatening token holders	✓
19	No privilege on managing others' balances	✓
20	Correct managing hierarchy	✓

4.2.2 Results for circuits

Number	Classification	Result
1	The circuits design is consistent with the logic of business requirements	✓
2	Circuits implementation is consistent with annotations, project white paper and other design materials	✓
3	Non-circuit code is consistent with the circuit implementation logic	✓
4	Correct implementation of key cryptographic circuits	✓
5	Circuit constraints complete	✓
6	Input parameters are safe	✓
7	Pass compiler check with no potential error	✓
8	No obvious gas-consuming operation	✓
9	No redundant code	X
10	Explicit implementation, variable type	✓
11	No hidden code about any logic that is not mentioned in design	✓
12	No risk threatening the developing team	✓
13	No risk threatening exchanges, wallets and DApps	✓
14	No risk threatening token holders	✓
15	No privilege on managing others' balances	✓

4.2.3 Results for Delayed Upgrade Scheme

Number	Classification	Result
1	The design of the scheme meets the requirement for contract upgradability	✓
2	Timelock contract is audited and secure	✓
3	The deployment script is consistent with the deployment logic of contracts	✓
4	The scheme ensures the continuity and upgradability of DeGate's business	✓
5	The same level of trustlessness as immutable contract	✓
6	No risk threatening the developing team	✓
7	No risk threatening exchanges, wallets and DApps	✓
8	No risk threatening token holders	✓
9	No privilege on managing others' balances	✓

It should be noted that part of the audit results for the scheme is based on DeGate's documents. The system's security and trustlessness are ensured when certain parameters in the contracts are correctly configured. Therefore, it is necessary to check the status of the contracts after deployment. In subsequent contract upgrades, it is essential to conduct audits on the new implementation contracts to ensure the security of the system.

4.3 Issues

1. 【circuit】Insufficient check for the noop order

   Security Risk   Medium

   - Description

     In the batchSpotTrade transaction, an order can be a noop order. There is no constraint to make sure other fields in order are 0. While the order is noop, the non-zero data will still be updated and the value in the storage leaf corresponding to the storageId will be overwritten.

     Besides the `isNoop` in BatchSpotTradeUser is never used.

     ```
     class BatchUserGadget: public GadgetT
     {
         ...
         DualVariableGadget isNoop;
     }

     class BatchOrderGadget : public GadgetT
     {
         ...
         DualVariableGadget isNoop;
     }
     ```

   - Consequence

     - User's storage leaf is overwritten by the wrong data.
     - When the isNoop field of the order is 0, the amount in the order will still be included in the "three token change values". This will lead to the failure of the judgment of "ensuring the cumulative revenue and expenditure balance of all currencies of all users". In turn, it will lead to wrong matching results.

   - Suggestion

     - When BatchSpotTradeUser.isNoop == 1, Constrain "isNoop" of all orders of the user euqal to 1.
     - Make sure the first order of userA and userB must not noop. Because at least two orders of different users are needed to match.
     - In "SelectOneTokenAmountGadget", when Order.isNoop == 1, set all outputs to 0.

   - Status

     The team has adopted this suggestion and fixed it.

2. 【circuit】Invalid logic for transfer verification

Security Risk　Medium

- Description

  In the Loopring, Transfer has two methods of signing, one is to verify with ECDSA on layer 1, and the other is to verify EdDSA on layer 2. DeGate's Transfer only retains the layer 2 signature, but the code logic of the other part is still retained.

```
isConditional(pb, type.packed, ".isConditional"),
needsSignature(pb, isConditional.result(), ".needsSignature")
```

- Consequence

  "isConditional" is determined by "type" in the transfer. "type" is filled in by Operator at will, so Operator only needs to make "isConditional" == 1, "isConditional" == 1 is to verify the transaction on layer 1. The signature of the transfer transaction will not be checked, and the operator can forge the transfer transaction.

- Suggestion

  Add constraints in TransferCircuit to check type == 0.

- Status

  The team has adopted this suggestion and added constraints to check type == 0.

3. 【circuit】Insufficient range constraints

   Security Risk　Medium

   - Description

     "fillS_A", "fillS_B" in SpotTradeCircuit are used to represent the filling value of this SpotTrade transaction to order A/B respectively. These two variables are of the Float32Encoding encoding type. However, in the whole process of using these two variables, there are no range constraints on them.

```
class SpotTradeCircuit : public BaseTransactionCircuit
{
    FloatGadget fillS_A;
    FloatGadget fillS_B;
}
```

   - Consequence

     The incorrect value of "fillS_A", and "fillS_B" may pass the circuit's verification. Users' money will be lost

   - Suggestion

     Add constraints for the two values, so that the binary length of these two values is not more than NUM_BITS_AMOUNT (96 bits).

   - Status

The team has adopted this suggestion and added constraints for length checking.

4. 【circuit】Insufficient data for data availability

   Security Risk Low

   - Description

     "Public data" of appKeyUpdate transaction does not contain "PublicKey", "disableAppKeySpotTrade", "disableAppKeyWithdraw", "disableAppKeyTransferToOther". This will cause data availability problems.

     ```
     const VariableArrayT getPublicData() const
         {
             return flattenReverse({
               typeTx.bits,
               typeTxPad.bits,
               accountID.bits,
               feeTokenID.bits,
               fFee.bits(),
               nonce.bits
             });
         }
     ```

   - Consequence

     The zkRollup needs to implement data availability. On the one hand, users need to recover the Merkle tree by themselves. Under extreme cases, layer 2 is completely lost, and users need to directly redeem their assets onchain, and users need to construct their own Merkle Path.

     And on the other hand, If a trader is evil, the user needs to know key data to be able to trace or hold accountable afterward.

   - Suggestion

     Add "PublicKey", "disableAppKeySpotTrade", "disableAppKeyWithdraw", "disableAppKeyTransferToOther" to the Public Data of AppKeyUpdate transaction.

   - Status

     There are two Merkle trees in the system, even if you don't know these fields of "PublicKey", "disableAppKeySpotTrade", "disableAppKeyWithdraw", and "disableAppKeyTransferToOther", you can still construct a valid Merkle Path to redeem assets.
     If a trader is evil, the team doesn't think it's necessary to be held accountable by call data. It is just a product design issue, so it does not need to be fixed.

5. 【circuit】Unchecked Consistency for type

   Security Risk Low

   - Description

UniversalCircuit check whether the transactions in Block are sorted and count the number of transactions according to 'txTypes', but 'txTypes' are not established between the 'type' in the TransactionGadget. As a result, the two can be filled in with different values.

```
class UniversalCircuit : public Circuit {
    ...
    std::vector<DualVariableGadget> txTypes;
}

class TransactionGadget : public GadgetT {
    ...
    DualVariableGadget type;
}
```

- Consequence

  For example, the Operator can tamper with "txTypes" then the statistics of "withdraw" are less than the real value, which will cause the contracts unable to handle those refund transactions, and the users' money will be locked.

- Suggestion

  The 'type' variable in the TransactionGadget should be treated as an initialization parameter and given directly by UniversalCircuit through 'txTypes'. Or you can check the consistency between 'type' and 'txTypes'.

- Status

  The team has adopted this suggestion and modified the code to check the consistency between 'type' and 'txTypes'.

6. 【circuit】Invalid length check

   Potential Risk Low

   - Description

     The `MulDivGadget` uses `assert(numBitsValue + numBitsNumerator <= NUM_BITS_FIELD_CAPACITY);` to check the length of `value` and `numerator`. However `numBitsValue` and `numBitsNumerator` are independent variables, they are not the length of valueandnumerator`.

```
  MulDivGadget(
        ...
        const VariableT &_value,
        const VariableT &_numerator,
        const VariableT &_denominator,
        unsigned int numBitsValue,
        unsigned int numBitsNumerator,
        unsigned int numBitsDenominator,
        ...
      {
          assert(numBitsValue + numBitsNumerator <=
 NUM_BITS_FIELD_CAPACITY);
      }
```

- Consequence

  This leads to the possibility that the parameters of FeeCalculatorGadget are invalid. the sum of the length of `value` and `numerator` may be larger than NUM_BITS_FIELD_CAPACITY.

- Suggestion

  check the length of `value` and `numerator`.

- Status

  The team has adopted this suggestion and added the logic of checking the length of the `value` and `numerator`.

7. 【circuit】Redundant independent witnesses

   Code Optimization Low

   - Description

     In the Groth16 protocol, the verifier verifies whether the prover knows some knowledge by specific circuits. The prover fills in the witnesses(private inputs) and public inputs to the prover system, and generates the proof that the input satisfies the constraints. For some unconstrained inputs, it is not necessary to exist, because they can be filled in any value. In DeGate code, the storage in `TransactionAccountState` is no need for user C, user D, user E, user F.

   - Suggestion

     Remove these independent witnesses.

   - Status

     The team adopted the suggestion to remove these independent witnesses based on our suggestions.

8. 【circuit】Too large NUM_BITS_AUTOMARKET_LEVEL

   Potential Risk Info

   - Description

According to documentation, the level of grid order does not exceed 64. But `NUM_BITS_AUTOMARKET_LEVEL=8` means the max_level can be set as 255.

```
static const unsigned int NUM_BITS_AUTOMARKET_LEVEL = 8;

level(pb, NUM_BITS_AUTOMARKET_LEVEL, FMT(prefix, ".level")),
maxLevel(pb, NUM_BITS_AUTOMARKET_LEVEL, FMT(prefix, ".maxLevel")),
```

- Consequence

  As well as if the level is more than 64 and less than 255, it can be verified successfully in the circuit.

- Suggestion

  Set `NUM_BITS_AUTOMARKET_LEVEL = 6`

- Status

  The documentation is wrong, and the code is right, so there is no risk.

9. 【circuit】Uncleaned code

   Code Optimization Info

   - Description

     The `feeMultiplier` field and related logic are removed from OrderGadget, but the feeMultiplier field and related constraints remain in class `Constants`, which are useless.

```
class Constants : public GadgetT
{
  ...
    feeMultiplier(make_variable(pb, ethsnarks::FieldT(FEE_MULTIPLIER),
FMT(prefix,
  ".feeMultiplier"))),
    ...
    feeMultiplier(make_variable(pb, ethsnarks::FieldT(FEE_MULTIPLIER),
FMT(prefix,
  ".feeMultiplier"))),
}
```

   - Suggestion

     Removes all code associated with the `feeMultiplier` field.

   - Status

     The team thinks it is only one constraint to be modificated and does not affect circuit safety or scale, so it is left unchanged for now.

10. 【circuit】Reduce hash circuits

    Code Optimization Discussion

- Description

  Because of Grid Trading, circuits in spotTrade and batchSpotTrade will calculate two Hash, one is required to verify regular order signature, and the other is required to verify grid order signature. But in the end, the circuit will only select one Hash to verify the signature. Because the hash operation in the circuit uses many constraints.

```
class AutoMarketOrderCheck : public GadgetT
{
    hash(
            pb,
            var_array(
              {blockExchange,
               storageID.packed,
               accountID.packed,
               tokenS.packed,
               tokenB.packed,
               amountS.packed,
               amountB.packed,
               validUntil.packed,
               fillAmountBorS.packed,
               taker,
               feeTokenID.packed,
               maxFee.packed,
               orderType.packed,
               gridOffset.packed,
               orderOffset.packed,
               maxLevel.packed,
               useAppKey.packed
               }),
            FMT(this->annotation_prefix, ".hash")),
    verifyHash(pb, isAutoMarketOrder.result(), hash.result(),
orderGadget.hash.result(), FMT(prefix, ".verifyHash"))
}
```

- Suggestion

  Instead of selecting the correct result from two Hashes, select the data that needs to be hashed. So that you only need one Hash in total.

- Status

  The team has adopted this suggestion and replaced two hashes with only one hash.

11. 【circuit】Permission risk for appKey

    Potential Risk Discussion

    - Description

Once a trader gets the user's appKey with the permission to withdraw, he can emit a withdraw transaction to refund the user's all assets to any L1 address. That is very dangerous. Although the user can turn off the permission to withdraw for appKey, there is still some risk here.

- Suggestion

  Remove the permission to withdraw for appKey.

- Status

  The user can turn off the permission to withdraw for appKey, so the team thinks the risk is manageable.

12. 【circuit】Redundant checks for forwarding

    Code Optimization Discussion

    - Description

      `NextForwardGadget` is used to judge order is a forward order or a reserve order. But `isForward` has represented the result.

    - Suggestion

      Remove `NextForwardGadget`.

    - Status

      The team thinks the current implementation logic is better understood, so no fix is needed.

13. 【contract】[ExchangeBlock.sol]Inconsistent logic about `isconditional` for `UpdateAccount`

    Logical Implementation Medium

    - Description

      The `isconditional` field is used to mark whether a transaction is checked in the contract or if the check is done in the circuit. If you do the check in the contract, the ecdsa signature verification needs to be done on the contract at the same time, otherwise the eddsa signature verification is done in the circuit.

      For an `UpdateAccount` transaction, the `isconditional` field can be either 0 or 1 in the circuit, but in the contract, all update account transactions are treated as `isconditional=1`, which leads to a problem with checking the number of numConditionalTransactions in the contract, and thus the block does not perform validation correctly in the contract.

      ```
      require(
        header.numConditionalTransactions == (header.depositSize +
      header.accountUpdateSize + header.withdrawSize),
        "invalid number of conditional transactions"
      );
      ```

    - Suggestion

The team needs to modify the processing logic in the circuit or contract to keep the logic consistent.

- To restrict the `UpdateAccount` transactions in the circuit to satisfy `isconditional=1`
- To modify the method of counting the number of the conditional `UpdateAccount` transaction in the contract

- Status

  The team has adopted this suggestion and fixed it by modifying the logic in the circuit.

14. 【contract】[DepositTransaction.sol] There is a DOS risk that may make it impossible to submit blocks properly

    Security Risk Low

    - Description

      Currently, the token listing is not a whitelist process, allowing arbitrary tokens to be registered. There is a large loop in the code to process deposit requests in the batch. An attacker could create a malicious token that he could manipulate with the `balanceOf()` function, which would cause the `process()` function to fail.

      Precisely, an attacker can control the result returned by the `balanceOf` interface for the malicious token. If the balance is made small, the code will trigger integer underflow protection and eventually revert. The attacker can make the `balanceOf` interface call fail to achieve the same goal. Finally, the entire submitBlocks transaction will fail.

```
function process(
     ExchangeData.State        storage S,
     ExchangeData.BlockContext memory  /*ctx*/,
     bytes                     memory  data,
     uint                              offset,
     bytes                     memory  /*auxiliaryData*/
     )
     internal
 {
     ...
     else if(deposit.depositType == 1) {
         uint32 tokenId = deposit.tokenID;
         uint256 unconfirmedBalance;
```

```
        if (tokenId == 0) {
            unconfirmedBalance = address(S.depositContract).balance
.sub(S.tokenIdToDepositBalance[tokenId]);
         } else {
            address token = S.tokenIdToToken[tokenId];
            // @audit DOS attack
            unconfirmedBalance =
ERC20(token).balanceOf(address(S.depositContract)).sub(S.tokenIdToDepos
itBalance[tokenId]);
         }
         require(unconfirmedBalance >= deposit.amount,
"INVALID_DIRECT_DEPOSIT_AMOUNT");
         S.tokenIdToDepositBalance[deposit.tokenID] =
S.tokenIdToDepositBalance[deposit.tokenID].add(deposit.amount);
     }
     ...
 }
```

```
unconfirmedBalance =
ERC20(token).balanceOf(address(S.depositContract)).sub(S.tokenIdToDepos
itBalance[tokenId]);
```

- Suggestion

  Reconsider the need to support the "Gas Saving Deposit" feature. Also, since arbitrary token support is allowed, all code calling the ERC20 interface should be carefully checked to exclude similar DOS risks.

- Status

  The team will handle the Gas Saving Deposit feature based on a whitelist token list in the backend, so there is no risk of DOS.

15. 【contract】[ExchangeV3.sol] The `blockVerifier` parameter cannot be updated properly

    Potential Risk Low

    - Description

      An interface for updating the `blockVerifierAddress` variable exists in `LoopringV3.sol`. However, a different `state.blockVerifier` instance is used in `ExchangeV3.sol`. When the `blockVerifierAddress` is updated in `LoopringV3.sol`, `ExchangeV3.sol` cannot synchronize the update.

```
    // located in LoopringV3.sol
    function updateSettings(
        address payable _protocolFeeVault,
        address _blockVerifierAddress,
        uint    _forcedWithdrawalFee
        )
        external
        override
        nonReentrant
        onlyOwner
    {
        updateSettingsInternal(
            _protocolFeeVault,
            _blockVerifierAddress,
            _forcedWithdrawalFee
        );
    }
```

- Suggestion

  Add an interface to refresh the `state.blockVerifier` variable from `LoopringV3.sol` in `ExchangeV3.sol`.

- Status

  Once ExchangeV3 is deployed, the `blockVerifier` contract address is no longer allowed to be updated, so there is no need to add an interface to update it. The team has removed the `verifier` parameter from the `updateSettings` function in LoopringV3.

16. 【contract】[DefaultDepositContract.sol] Unable to return user assets under special circumstances

   Potential Risk Low

   - Description

     When `amount` is zero, and `msg.value` is not zero, the presence of the `ifNotZero` modifier will cause the function to end early, and the assets deposited by the user into the contract will not be appropriately returned.

```
// located in DefaultDepositContract.sol
function deposit(
    address from,
    address token,
    uint248  amount,
    bytes   calldata /*extraData*/
    )
    external
    override
    payable
    onlyExchange
    ifNotZero(amount)
    returns (uint248 amountReceived)
{
    ...
}
```

```
modifier ifNotZero(uint amount)
{
    if (amount == 0) return;
    else  _;
}
```

- Suggestion

  Refer to Loopring protocols PR 2592 for a fix.

- Status

  The team fixed it based on our suggestions.

17. 【contract】[WithdrawTransaction.sol]Unmodified withdraw logic for protocol account

Potential Risk Low

- Description

  The operator account is used to receive the protocolfee. In the current withdraw logic, the redemption logic for the operator account is the same as the general account, i.e. the operator account is no longer treated as a special account. In the contract code, however, the special processing logic for the operator account is still retained in the `forcewithdraw` mode.

  On the other hand, the value of the operator account's AccountID is 1, but in the contract code, the field `ACCOUNTID_PROTOCOLFEE` is 0.

```
if (forcedWithdrawal.timestamp != 0) {
...
} else {
  // Allow the owner to submit full withdrawals without authorization
  // - when in shutdown mode
  // - to withdraw protocol fees
  require(
  withdrawal.fromAccountID == ExchangeData.ACCOUNTID_PROTOCOLFEE ||
  S.isShutdown(),
  "FULL_WITHDRAWAL_UNAUTHORIZED"
  }
 }
```

- Suggestion

  It results to the account with AccountID=0 can withdraw directly without the necessary check in the `forcewithdraw` mode.

- Status

  The team think it is necessary to keep the special logic for the account with AccountID=0 even though it is no longer a protocol account, and therefore no fix is needed.

  But we should notice here that the meaning of `ACCOUNTID_PROTOCOLFEE` is no longer a protocol account.

18. 【contract】[ExchangeV3.sol] `loopringAddr` is redundant

Code Optimization Info

- Description

  `ExchangeV3.sol` has a `loopringAddr` variable, duplicated with the `state.loopring`.

  ```
  // located in ExchangeV3.sol
  address public loopringAddr; // @audit not used?
  ```

- Suggestion

  Remove `loopringAddr` and use `state.loopring` directly.

- Status

  The team fixed it based on our suggestions.

19. 【contract】[ExchangeV3.sol] `withdrawExchangeFees` needs to enhance the checking of token parameters

    Code Optimization Info

    - Description

      Since the `token` variable can be any value, administrators can set it to `state.depositContract` and thus interact directly with the crucial contracts that house the assets.

      ```
      // located in ExchangeV3.sol
      function withdrawExchangeFees(
          address token,
          address recipient
          )
          external
          override
          nonReentrant
          onlyOwner
      {
          require(recipient != address(0), "INVALID_ADDRESS");
          if (token == address(0)) {
              uint amount = address(this).balance;
              recipient.sendETHAndVerify(amount, gasleft());
          } else {
              uint amount = ERC20(token).balanceOf(address(this));
              token.safeTransferAndVerify(recipient, amount);
          }
          emit WithdrawExchangeFees(token, recipient);
      }
      ```

    - Suggestion

      Although it is not currently exploitable, it is still recommended to strengthen the parameter checking here. Refer to code here.

    - Status

The team believes that there is no attack scenario here in the current implementation and therefore no fix is needed.

20. 【contract】[LoopringIOExchangeOwner.sol] Is `submitBlocksWithCallbacks` missing the necessary validation?

    Code Optimization Discussion

    - Description

      The `submitBlocksWithCallbacks()` function implementation removes a lot of code from the original version, resulting in the `config` variable being passed in without actually being used.

```
function submitBlocksWithCallbacks(
    bool                          isDataCompressed,
    bytes            calldata data,
    CallbackConfig  calldata config
    )
    external
{
    if (config.blockCallbacks.length > 0) {
        require(config.receivers.length > 0, "MISSING_RECEIVERS");
    }
    require(
        hasAccessTo(msg.sender, SUBMITBLOCKS_SELECTOR) || open,
        "PERMISSION_DENIED");
    bytes memory decompressed = isDataCompressed ?
        ZeroDecompressor.decompress(data, 1): data;
    require(
        decompressed.toBytes4(0) == SUBMITBLOCKS_SELECTOR,
        "INVALID_DATA");
    target.fastCallAndVerify(gasleft(), 0, decompressed);
}
```

    - Suggestion

      Discuss the reason for the code deletion here and whether the `config` variable is necessary to keep. Also follow this update.

    - Status

      The callback function is not used here in DeGate, so the team removes the `config` parameter.

5. Conclusion

DeGate Protocol is a common ZK Rollup with additional trading features for specific project goals and use cases. The contract analysis revealed 8 issues as detailed above, and the circuit code analysis revealed 12 issues as detailed above. The SECBIT team identified a critical issue in the smart contract. All of these issues have been described above, along with suggestions for fixing them. The DeGate team has adequately responded to and/or fixed all issues as requested.

Disclaimer

APPENDIX

Appendix 1: Vulnerability/Risk Level Classification

Level	Description
High	Severely damage the system's integrity and allow attackers to steal ethers and tokens, or lock ethers inside the contract.
Medium	Damage the application's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to the finance.
Info	Relevant to practice or rationality of the code, could possibly bring risks.
Discussion	Some suggestions for optimizing the code logic

Appendix 2: Type Classification

Type	Description
Security Risk	The risk of compromising system security directly
Code Optimization	Optimize code implementation
Logical Implementation	Vulnerability in design or implementation logic
Potential Risk	The potential risk of compromising system security
Code Revising	Non-standard usage of code writing

SECBIT Labs is devoted to construct a common-consensus, reliable and ordered blockchain economic entity.

http://www.secbit.io

audit@secbit.io

@secbit_io