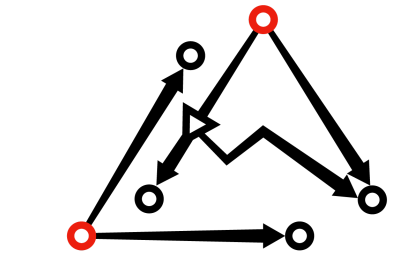

Least Authority

PRIVACY MATTERS

DeGate Smart Contracts

Security Audit Report

DeGate DAO

Updated Final Audit Report: 4 July 2022

Table of Contents

Overview

Background

DeGate DAO has requested that Least Authority perform a security audit of the DeGate Order Book system Smart Contracts.

Project Dates

- **February 14 -April 8:** Initial Review *(Completed)*
- **April 13:** Delivery of Initial Audit Report *(Completed)*
- **May 4 - May 5**: Verification Review *(Completed)*
- **May 6**: Delivery of Final Audit Report *(Completed)*
- **July 4**: Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Zoumana Cise, Security Researcher and Engineer
- Ahmad Jawid Jamiulahmadi, Security Researcher and Engineer
- Steven Jung, Security Researcher and Engineer
- Nishit Majithia, Security Researcher and Engineer
- Rai Yang, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the DeGate Smart Contracts followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repository is considered in-scope for the review:

- DeGate Smart Contracts:
  https://github.com/degatedev/degate-protocols-ext

Specifically, we examined the Git revision for our initial review:

`785cb1000cfff703782be706a4f83bc6830d77ef`

For the verification, we examined the Git revision:

`4ca1ba56b1d37b8e1cd67e48a976f2ed986d3cf7`

For Issue G our team examined the following Git revision:

`9e5f4072533f368dad97825f62f250d33b0cdec0`

For the review, this repository was cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/DeGate-Smart-Contracts

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- DeGate Whitepaper 1.0:
  https://degate.com/
- DeGate overall architecture & changes against Loopring 3.6.1.pdf
  (*shared with Least Authority via email on 22 September 2021*)
- Trail of Bits Audit Report.pdf
  (*shared with Least Authority via email on 25 February 2022*)

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the zk-SNARK circuit and smart contract implementations;
- Adherence to the specifications and best practices;
- Common and case-specific implementation errors in the zk-SNARK circuit and smart contract code;
- Adversarial actions and other attacks on the smart contracts;
- Potential misuse and gaming of the smart contracts;
- Attacks that impact funds such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and security exploits that would impact the code's intended use or disrupt the execution of the code;
- Key management, encryption, and storage;
- Vulnerabilities in the zk-SNARK circuit and smart contract code;
- Protection against malicious attacks and other methods of exploitation;
- Inappropriate permissions and excess authority;
- Performance problems or other potential impacts on performance;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

The DeGate Order Book system is an ERC-20 token exchange platform, which forks from Loopring V3 that uses ZK Rollups to significantly increase transaction throughput and reduce transaction costs. For this audit, we reviewed the smart contracts component of the DeGate system.

The DeGate smart contracts perform a dual function. First, the contracts listen for and process L2 block data and verify the correctness of ZK proofs and second, they manage user assets and perform deposit and withdrawal transactions on the Ethereum mainnet. We performed a comprehensive review of the smart contracts and investigated the ExchangeV3 smart contract, the deposit smart contract, the Exchange library, and the Transaction library. We compared these implementations to the Loopring V3 fork (commit `d936083b182bee098c9d268b1dbed108e5e594d4`) for changes.

We found that the DeGate smart contracts implement complicated L2 transaction data processing logic that, coupled with a lack of adherence to coding best practice as well as insufficient code comments and documentation, make the smart contract codebase challenging to reason about from a security perspective. We also identified issues and suggestions in the design and implementation that, if resolved, would improve the quality and security of the implementation.

System Design

Our team performed a close examination of the design of the DeGate smart contracts and found that the security of the smart contracts can be improved by reducing centralization and complexity. The `ExchangeV3` smart contract is controlled by an Externally Owned Account (EOA), which is vulnerable to an attacker compromising the private key associated with the contract owner account. Compromise of the smart contract owner account could cause the irrevocable draining of funds. We recommend that the smart contract be controlled by a Multi-Sig account until a Distributed Autonomous Organization (DAO) mechanism is put in place (Issue A).

The `ExchangeBlock` smart contract processes L2 blocks that contain commitment data, proof data, and specific metadata. In the event of a missing check, a block with an invalid number of conditional transactions would be committed, potentially resulting in unexpected behavior. We recommend implementing a check to verify the number of conditional transactions in the block header (Issue B).

We identified instances where user inputs are processed without any validation. Lack of appropriate input validation could lead to unexpected behavior and result in the loss of funds. We recommend that all inputs be sanitized and validated appropriately (Issue D).

Our team found that the platform is particularly susceptible to asset price manipulation attacks. First, the platform performs token exchanges using a deprecated decentralized exchange (DEX), which has low levels of liquidity. Pools with low liquidity are more susceptible to the manipulation of the prices of underlying assets. We recommend that actively maintained pools with healthy levels of liquidity be used to exchange tokens (Issue E). Furthermore, the function used to sell tokens using the DEX does not implement sufficient slippage control, which could make `ExchangeV3` transactions vulnerable to sandwich attacks. We recommend that appropriate slippage control be implemented (Issue F).

Code Quality

The codebase could be significantly improved with better organization and adherence to best practices. We found that functions in the smart contracts are not defined according to best practice, which, as a result, increases complexity and reduces readability. We recommend that functions be defined appropriately (Suggestion 1). In addition, we found an instance where the visibility of a variable was incorrectly set. We recommend that the visibility of variables be set according to best practices (Suggestion 7). We recommend that a pinned and up-to-date Solidity compiler be used that is consistent throughout the contracts (Suggestion 6).

Our team noted that the practice of shadowing variables is used in the implementation, which reduces the readability of the code and increases the risk of confusion, leading to implementation errors and security vulnerabilities being missed. We recommend that shadowing variables be avoided and that accepted variable naming conventions be used (Suggestion 5).

The size of the `ExchangeV3` smart contracts is greater than what is practically deployable. We recommend reducing the size of the smart contract as well as the deployment and transaction costs (Issue C). There are many instances of unnecessary reentrancy guards, which increase the complexity of the code and the transaction costs. We recommend that unnecessary reentrancy guards be removed and that the check-effects pattern be implemented (Suggestion 2). Moreover, there are several instances of unused and commented-out code within the codebase, reducing the readability and organization of the code, which could lead to implementation errors and security vulnerabilities being missed. We recommend that unused code be removed from the codebase (Suggestion 8).

The DeGate smart contracts implement gas optimization functionality that is no longer supported, as of the London hardfork. We recommend removing this functionality to improve the readability of the code and to reduce the size of the smart contracts (Suggestion 9). In addition, we identified an optimization

opportunity that would reduce computation and cost. We recommend removing the `offset` variable and using the formula to add multiples of 32 directly (Suggestion 10).

Tests

We found that sufficient test coverage has been implemented. However, some of the tests implemented failed. We recommend that failed tests be corrected (Suggestion 3).

Documentation

The project documentation provided describes the general architecture of the system, each of the components, and the interaction between those components. In addition, it offers a description of the changes completed prior to the previously audited Loopring 3.6.1. However, there is no documentation that describes off-chain data and state management.

Code Comments

There are insufficient code comments in the codebase with security-critical components. For example, the L2 data processing and verification do not have any code comments. The documentation contained within the code should be comprehensive and explain the intended functionality of each of the components. We recommend adherence to NatSpec guidelines for Solidity code comments.

Scope

The scope of this security audit was sufficient for the review of the smart contracts component of the DeGate platform. However, the smart contracts depend on the security of off-chain components that were not in-scope for this review, including the operator, merkle tree, and others.

Dependencies

Our team found that building the system was problematic and inconsistent. Specifically, recursive submodules that are no longer supported were required and developer dependencies had to be resolved forcefully in order to successfully build the system for testing. We recommend optimizing the build process to increase consistency and improve the review and testing (Suggestion 4).

Specific Issues & Suggestions

We list the issues and suggestions found during the review in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS	VERIFICATION
Issue A: ExchangeV3 Smart Contract Is Owned by an EOA	Planned	Unresolved
Issue B: Missing Check for Number of Conditional Transactions in Block Header	Completed	Resolved
Issue C: Smart Contract Undeployable	Completed	Resolved
Issue D: Missing Zero Address Validation	Completed	Resolved
Issue E: Uniswap V1 Is Deprecated	Completed	Resolved
Issue F: Function Vulnerable to Sandwich Attack	Completed	Resolved

Issue G: Compromise of the EOA Can Result in the Loss of Funds	Completed	Resolved
Suggestion 1: Define Functions Appropriately	Completed	Resolved
Suggestion 2: Remove Unnecessary Reentrancy Guard	Will Not Fix	Unresolved
Suggestion 3: Correct Failed Tests	Completed	Resolved
Suggestion 4: Optimize Build Process	Planned	Unresolved
Suggestion 5: Do Not Shadow Variables	Completed	Resolved
Suggestion 6: Use an Updated and Non-floating Pragma Version	Planned	Unresolved
Suggestion 7: Set State Variable Visibility Explicitly	Completed	Resolved
Suggestion 8: Remove Unused and Commented Code	Not Completed	Partially Resolved
Suggestion 9: Remove Gas Tokens	Completed	Resolved
Suggestion 10: Optimize Code	Completed	Resolved

Issue A: ExchangeV3 Smart Contract Is Owned by an EOA

Location

`contracts/core/impl/ExchangeV3.sol#L117`

`contracts/core/impl/ExchangeV3.sol#L454`

`contracts/core/impl/ExchangeV3.sol#L300`

Synopsis

In the current implementation, the `ExchangeV3` smart contract is owned by an EOA. If this account is compromised, the attacker can cause harm to the exchange owner and users.

Impact

The attacker gains control of the `ExchangeV3` smart contract owner account and steals funds from users and the DeGate system.

Preconditions

The attacker must compromise the `ExchangeV3` smart contract owner's private key.

Technical Details

Once the attacker compromises the `ExchangeV3` smart contract owner account, they can launch various attacks. For example, the attacker can reset the smart contract to a malicious verifier smart contract, which could bypass verification or verify a fake proof that would enable an attacker to submit invalid

transactions and steal funds from users of the platform. The attacker could also withdraw fees collected to an attacker-controlled address.

Mitigation

We recommend renouncing the `ExchangeV3` smart contract ownership using renounceOwnership, after the contract is deployed and parameters are set.

Remediation

As an interim remediation, we recommend the use of a Multi-Sig account for the owner of the `ExchangeV3` smart contract. We recommend that the ownership of the smart contract be delegated to a DAO.

Status

The DeGate team has responded that during mainnet deployment, they will use the Multi-Sig wallet to manage the owner account of the `ExchangeV3` smart contract. However, this has not yet been implemented at the time of the verification.

Verification

Unresolved.

Issue B: Missing Check for Number of Conditional Transactions in Block Header

Location

`contracts/core/impl/libtransactions/BlockReader.sol#L21`

`contracts/core/impl/libexchange/ExchangeBlocks.sol#L191`

Synopsis

In the `ExchangeBlocks` smart contract, where the function `processConditionalTransactions` performs L2 block processing for the DeGate system, there is no check to verify the number of conditional transactions recorded in the block header.

Impact

A block with an invalid number of conditional transactions in the header could get commited, which would cause undefined behavior.

Technical Details

In the block header struct, the variable `numConditionalTransactions` records the number of conditional transactions in a block, which should equal the sum of all conditional transactions (`depositSize` + `accountUpdateSize` + `withdrawSize`). In the `processConditionalTransactions` in the `ExchangeBlocks` smart contract, no check is implemented. For a blockheader with an inconsistent number of conditional transactions, and a size corresponding to one of the 3 types of conditional transactions, the invalid block will be processed and committed.

Remediation

We recommend adding a check in the function `processConditionalTransactions`:

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

Status

The DeGate team added a check to verify that the parameters are correct.

Verification

Resolved.

Issue C: Smart Contract Undeployable

Location

Smart contract with exceeding size:

contracts/core/impl/ExchangeV3.sol

Unused functions:

contracts/lib/MathUint.sol#L48

contracts/lib/ERC20SafeTransfer.sol#L38-L52

contracts/lib/ERC20SafeTransfer.sol#L93-L172

Synopsis

The size of the `ExchangeV3` smart contract exceeds the 24KB limit introduced in the Spurious Dragon hardfork. This could prevent deployment to mainnet.

Remediation

We recommend converting `require` statements with string messages into `revert` statements with customized errors to reduce the smart contract size and transaction gas costs. Additionally, we recommend enabling the optimizer with a low `runs` value configuration to further reduce the smart contract size. Moreover, we recommend removing the unused referenced functions and checking for other unused functions (if any) in libraries that `ExchangeV3` uses, which were out of scope for this review.

Status

The DeGate team has responded that the `ExchangeV3` smart contract does not exceed the 24KB limit. Thus, there is no impact on deployment.

Verification

Resolved.

Issue D: Missing Zero Address Validation

Location

contracts/aux/agents/OpenGSN2Agent.sol#L21

contracts/aux/token-sellers/UniswapTokenSeller.sol#L44

Synopsis

In the `OpenGSN2Agent` smart contract, when initializing the smart contract, input validation for a zero address is missing on the `_exchange` address variable. In addition, in the `UniswapTokenSeller` smart contract, the input validation for a zero address is missing when setting the `_recipient` address.

Impact

Passing a zero address to the aforementioned smart contract functions, without any input validation, might result in the loss of funds sent to exchange addresses, which can be set as zero addresses .

Remediation

We recommend implementing appropriate input validation of user-supplied values.

Status

The DeGate team has removed these smart contracts, as they were part of the original Loopring code and are not used by the application.

Verification

Resolved.

Issue E: Uniswap V1 Is Deprecated

Location

`contracts/aux/token-sellers/UniswapTokenSeller.sol`

`contracts/aux/token-sellers/ITokenSeller.sol`

`contracts/thirdparty/uniswap/UniswapExchangeInterface.sol`

`contracts/thirdparty/uniswap/UniswapFactoryInterface.sol`

Synopsis

The above-referenced smart contracts and interfaces use the Uniswap V1 platform to perform token exchanges. Uniswap V1 is now deprecated and, as a result, has low levels of liquidity in its liquidity pools, which makes the pools more vulnerable to asset price manipulation attacks.

Impact

The use of liquidity pools with low levels of liquidity makes asset price manipulation within the pool more practical and improves the probability of a successful sandwich attack, which would cause a loss in exchange assets.

Mitigation

We recommend the use of liquidity pools that are not deprecated, are actively maintained, and have healthy levels of liquidity to safeguard against asset manipulation attacks.

Status

The DeGate team removed the smart contracts referenced in the examples.

Verification

Resolved.

Issue F: Function Vulnerable to Sandwich Attack

Location

contracts/aux/token-sellers/UniswapTokenSeller.sol#L49

Synopsis

The function `sellToken` exchanges ERC-20 tokens and ETH using a DEX. However, the function does not implement appropriate slippage control, making transactions vulnerable to sandwich attacks. Although the slippage detection functionality is implemented, only the DEX price is impacted by the current transaction. If an attacker makes a front running transaction, this function will get pool data to calculate the price impact after the front running transaction. With that information alone, it is impossible to calculate slippage due to the front running transaction. As a result, this functionality cannot detect or prevent the front running of DEX transactions.

Impact

An attacker can front run exchange transactions with the DEX used to exchange tokens. As a result, the `ExchangeV3` could receive much lower amounts than expected for tokens sold on the DEX.

Mitigation

We recommend the implementation of appropriate slippage control where a token exchange transaction is reverted if the value of the transaction falls below a certain expected threshold.

Status

The DeGate team removed the functionality referenced in the issue.

Verification

Resolved.

Issue G: Compromise of the EOA Can Result in the Loss of Funds

Location

contracts/core/impl/ExchangeV3.sol#L117

contracts/core/impl/ExchangeV3.sol#L454

contracts/core/impl/ExchangeV3.sol#L300

Synopsis

In the current implementation, the `ExchangeV3` smart contract is owned by an EOA. If this account is compromised, the attacker can cause harm to the exchange owner and users.

Impact

The compromise of the EOA private key could result in the loss of all platform assets..

Preconditions

The attacker must compromise the `ExchangeV3` smart contract owner's private key.

Technical Details

Once the attacker compromises the `ExchangeV3` smart contract owner account, they can launch various attacks. For example, the attacker can reset the smart contract to a malicious verifier smart contract,

which could bypass verification or verify a fake proof that would enable an attacker to submit invalid transactions and steal funds from users of the platform. The attacker could also withdraw fees collected to an attacker-controlled address.

Mitigation

We recommend renouncing the `ExchangeV3` smart contract ownership using renounceOwnership, after the contract is deployed and parameters are set.

Remediation

As an interim remediation, we recommend the use of a Multi-Sig account for the owner of the `ExchangeV3` smart contract. We recommend that the ownership of the smart contract be delegated to a DAO.

Status

The DeGate team has removed the interfaces `registerCircuit`, `disableCircuit`, and `refreshBlockVerifier`. Additionally, a maximum limit and a time delay for resetting platform fees have been added. As a result, in case of a compromise, the attacker can only reset the platform fees, which are capped at 1%.

Verification

Resolved.

Suggestions

Suggestion 1: Define Functions Appropriately

Location

Defined Internal:

contracts/core/impl/DefaultDepositContract.sol#L183

contracts/core/impl/libtransactions/AccountUpdateTransaction.sol#L74

contracts/core/impl/libtransactions/AccountUpdateTransaction.sol#L108

contracts/core/impl/libexchange/ExchangeDeposits.sol#L109

contracts/core/impl/libtransactions/WithdrawTransaction.sol#L231

contracts/core/impl/libtransactions/WithdrawTransaction.sol#L260

Defined Public:

contracts/core/iface/ILoopringV3.sol#L67

contracts/core/impl/LoopringV3.sol#L71

contracts/core/iface/ILoopringV3.sol#L113

contracts/core/impl/LoopringV3.sol#L148

contracts/core/impl/ExchangeV3.sol#L74

contracts/aux/access/LoopringIOExchangeOwner.sol#L68

contracts/core/iface/IAgentRegistry.sol#L38

contracts/aux/agents/FastWithdrawalAgent.sol#L72

contracts/aux/agents/OpenGSN2Agent.sol#L39

contracts/aux/fee-vault/IProtocolFeeVault.sol#L87

Synopsis

Several of the above-referenced functions are defined as `internal`. However, they are not being used in any of the derived smart contracts. Others are defined as `public` when they should actually be defined as `external`. It is considered best practice to define function access modifiers based on where the function is going to be used in order to improve the readability of the code and make it easier to identify incorrect assumptions about who can call the function.

Mitigation

We recommend defining the referenced internal and public functions, in the function definition, as private and external respectively by replacing the `internal` keyword with `private` and the `public` keyword with `external` .

Status

The DeGate team has updated the referenced functions.

Verification

Resolved.

Suggestion 2: Remove Unnecessary Reentrancy Guard

Location

Examples (non-exhaustive):

contracts/core/impl/BlockVerifier.sol#L36

contracts/core/impl/LoopringV3.sol#L63

contracts/core/impl/LoopringV3.sol#L111

/contracts/core/impl/LoopringV3.sol#L131

Synopsis

Unnecessary reentrancy guards introduce complexity and increase gas costs. It is not needed where no reentrancy could happen or if a check-effects pattern could be used in the code.

Mitigation

We recommend applying the check-effects pattern in the code and removing unnecessary reentrancy guards.

Status

The DeGate team has responded that they intend to keep this part of the reentrancy guard since these functions are only called in limited instances and have little impact on gas costs.

Verification

Unresolved.

Suggestion 3: Correct Failed Tests

Location

`test/testExchangeUtil.ts`

Synopsis

There are unit tests that fail in the `testExchangeUtil.ts`. Failed tests prevent verifying that the implementation behaves as intended.

Mitigation

We recommend correcting the failed tests.

Status

The DeGate team has corrected the failing test cases.

Verification

Resolved.

Suggestion 4: Optimize Build Process

Location

Unsatisfied Recursive submodules:
`DeGate-Smart-Contracts/packages/loopring_v3/ethsnarks/depends/libsnark/depends/`

`Ate-pairing,gtest, libff, libfqfft, libsnark-supercop, xbyak`

Dev Dependencies causing build conflicts:

`packages/loopring_v3/package.json#L62`

`packages/loopring_v3/package.json#L63`

`packages/loopring_v3/package.json#L70`

`packages/loopring_v3.js/package.json#L23`

`packages/loopring_v3.js/package.json#L24`

`packages/loopring_v3.js/package.json#L30`

Synopsis

We found that the build process is not optimized and that unexpected and random errors may occur while building the project, which makes testing the project challenging. Specifically, several external recursive submodules cannot be satisfied due to Github no longer supporting the unauthenticated git protocol on port 9418. In addition, there are some devdependency conflicts that cause the build to fail. These can be resolved using the `--force` flag. However, this is not a recommended practice.

Mitigation

We recommend optimizing the build process to make the testing of the project consistent and more straightforward. For reference, see reproducible builds.

Status

The DeGate team has responded that they plan to research how to optimize the build process.

Verification

Unresolved.

Suggestion 5: Do Not Shadow Variables

Location

contracts/core/impl/ExchangeV3.sol#L204

contracts/core/impl/ExchangeV3.sol#L413

contracts/core/impl/ExchangeV3.sol#L428

contracts/core/impl/ExchangeV3.sol#L491

contracts/core/impl/ExchangeV3.sol#L519

contracts/core/impl/ExchangeV3.sol#L605

contracts/core/impl/ExchangeV3.sol#L634

contracts/core/impl/ExchangeV3.sol#L57

contracts/aux/agents/ForcedWithdrawalAgent.sol#L27

Synopsis

The owner parameter in the referenced functions or modifier shadows the owner defined in Ownable.sol. We did not identify any issues with these shadowings. However, they degrade the readability of the code, cause confusion, and may result in a bug in future versions.

Mitigation

We recommend that the aforementioned variables be renamed to explicitly and specifically describe their intended functionality.

Status

The DeGate team has renamed the variables.

Verification

Resolved.

Suggestion 6: Use an Updated and Non-floating Pragma Version

Location

contracts/core/impl/ExchangeV3.sol#L3

Synopsis

Smart contracts in the project have their pragma set to ^0.7.0. Compiling with different versions of the compiler might lead to unexpected results. In addition, older versions of the Solidity compiler contain bugs that have been fixed in more recent versions of the compiler, including up-to-date security patches.

Mitigation

In order to maintain consistency and to prevent unexpected behavior, we recommend that the Solidity compiler version be pinned by removing "^" and that the developers use the latest version of the Solidity compiler.

Status

The DeGate team has responded that they plan to research in order to determine which compiler version they should deploy.

Verification

Unresolved.

Suggestion 7: Set State Variable Visibility Explicitly

Location

`contracts/core/impl/DefaultDepositContract.sol#L32`

Synopsis

Visibility is not set for the referenced state variable. It is best practice to explicitly label the visibility to prevent incorrect assumptions about who can access the state variable.

Mitigation

We recommend explicitly labeling the visibility of the variable as `internal`.

Status

The DeGate team has updated the visibility of the variable.

Verification

Resolved.

Suggestion 8: Remove Unused and Commented Code

Location

`contracts/lib/Poseidon.sol#L710-L1505`

`contracts/aux/access/DelayedTransaction.sol#L304-L308`

`contracts/aux/access/LoopringIOExchangeOwner.sol#L126-L280`

`contracts/aux/transactions/TransactionReader.sol#L19-L73`

Synopsis

There are many instances of commented-out or unused code in the codebase. This reduces readability and can confuse reviewers and maintainers.

Mitigation

We recommend that unused and commented-out code be removed to make the smart contracts more readable and to reduce their size, thereby reducing gas costs.

Status

The DeGate team has removed some, but not all, unused and commented-out code.

Verification

Partially Resolved.

Suggestion 9: Remove Gas Tokens

Location

`contracts/aux/gas/ChiDiscount.sol`

`contracts/test/Chi/ChiToken.sol`

`contracts/test/Chi/DummyWriteContract.sol`

Synopsis

Gas tokens are no longer useful after the London hardfork.

Mitigation

We recommend removing gas tokens and related tests.

Status

The DeGate team has removed the impacted smart contracts.

Verification

Resolved.

Suggestion 10: Optimize Code

Location

`contracts/aux/access/LoopringIOExchangeOwner.sol#L250-L264`

Synopsis

In the referenced location, multiple additions of 32 to `offset` are not needed and consume unnecessary gas.

Mitigation

We recommend removing the `offset` variable and using the formula to add multiples of 32 directly.

Status

The DeGate team has updated the smart contract, resolving the suggestion.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.