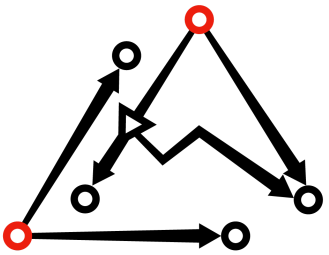

Least Authority

PRIVACY MATTERS

DeGate zk-SNARK Circuit

Security Audit Report

DeGate Technology Ltd

Final Audit Report: 6 May 2022

Table of Contents

Overview

Background

DG Technologies Ltd has requested that Least Authority perform a security audit of the DeGate Order Book system zk-SNARK Circuit implementation.

Project Dates

- **March 7 - April 12:** Initial review *(Completed)*
- **April 15:** Delivery of Initial Audit Report *(Completed)*
- **May 4 - 5**: Verification Review *(Completed)*
- **May 6**: Delivery of Final Audit Report *(Completed)*

Review Team

- Jasper Hepp, Cryptography Researcher and Engineer
- Anna Kaplan, Cryptography Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the DeGate zk-SNARK Circuit 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 zk-SNARK Circuit: https://github.com/LeastAuthority/DeGate-Circuit

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

`40a9bc2588aa6c43f25205a1b4e759ebdbef1366`

For the verification, we examined the Git revision:

43dc00a78425694fc1a703e03a642ac45c06e750

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

DeGate zk-SNARK Circuit: https://github.com/LeastAuthority/DeGate-Circuit

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:

- `Statement.md`: https://github.com/LeastAuthority/DeGate-Circuit/blob/master/packages/loopring_v3/circuit/statements.md

- DeGate Whitepaper 1.0:
  https://common-resource.degate.com/doc/whitepaper_en.pdf
- DeGate overall architecture & changes against Loopring 3.6.1:
  https://github.com/LeastAuthority/DeGate-Circuit/blob/master/packages/loopring_v3/security_audit/LoopringV3_6_vs_V3_1.pdf
- Security Audit Reports:
  Least Authority - Loopring Design and Implementation: Circuit 16 March 2021
  Least Authority - Loopring Design and Implementation: Smart Contracts 16 March, 2021
  SECBIT Labs Loopring V3 15 November, 2019
- Testnet:
  https://testnet.degate.com/
- `Design.md`:
  https://github.com/degatedev/degate-protocols-ext/blob/degate_0.2.0/packages/loopring_v3/DESIGN.md

In addition, this audit report references the following documents:

- D. Hopwood, 2019, "Designing efficient R1CS circuits." [H19]

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the zk-SNARK circuit implementations;
- Adherence to the specifications and best practices;
- Common and case-specific implementation errors in the zk-SNARK circuit;
- Adversarial actions and other attacks on the zk-SNARK
- Potential misuse and gaming of the zk-SNARK;
- 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;
- Protection against malicious attacks and other methods of exploitation;
- Inappropriate permissions and excess authority;
- 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 the previously audited Loopring 3.6 that uses ZK-Rollups to significantly increase transaction throughput and reduce transaction costs. For this audit, we reviewed the zk-SNARK circuit component of the DeGate system. As with Loopring 3.6, DeGate utilizes a zk-SNARK to off-load large parts of the computation off-chain. This solution is classified as a validity proof system, which ensures that state transition must be correct based on the properties provided in the encryption scheme.

Since DeGate originated as a fork of Loopring 3.6, we looked at the changes the DeGate team made to Loopring's circuit and gadget codebase and focused in particular on the `AppKeyUpdateCircuit` as well as the `BatchSpotTradeCircuit`. Both circuits are new implementations that were absent in the original Loopring code.

We analyzed the previous security reports and compared them against DeGate's deviations from Loopring 3.6. We extensively looked at the correct use of the math gadgets since the `add`-, `sub`-, and `mul`- gadgets are only safe for operands small enough such that the result does not overflow the base field modulus. During an overflow, the range check would pass, resulting in unintended behavior. In such a case, the `addgadget` could be used to decrease nonces. However, we could not identify any issues with the math gadgets.

We examined differences between on-chain and off-chain verification and the `compressKeyGadget` and noticed that a cofactor-checkless EdDSA algorithm is used and public keys are not enforced to be cofactor cleared. We analyzed how this interacts with EthSNARK's EdDSA implementation. Assuming that `(s, R)` is a signature for a public key `A`, EthSNARK does not require `s` to be an element of the scalar field of the associated Edwards curve. Therefore, signature verification does not have strong unforgeability under chosen message attacks. Moreover, since there are no small-subgroup checks enforced on `A`, signatures are not strongly binding. Although this is not a recommended approach (see reference), after a close analysis of these limitations within the DeGate zk-SNARK circuit code, we could not identify any attacks based on the DeGate implementation of the EdDSA algorithm.

A well written and comprehensive zk-SNARK statement serves as the foundation of any implementation in code. The statement is critical for testing and verifying that the implementation realizes the intended design. Without such a statement, there is no meaningful frame of reference with which to reason about the security of any implementation. In the case of DeGate, the zk-SNARK's statement is reverse engineered from the code.However, zk-SNARK development best practices (to the extent that they currently exist) recommend starting a project on the statement level, and only then implementing the code [H19].

Although the DeGate zk-SNARK circuit implementation followed Loopring's reverse engineering approach to develop the statement, the DeGate implementation did not update the statement properly. We recommend that an accurate and up-to-date DeGate zk-SNARK statement be completed in order to improve the review of the security characteristics of the system (Issue E).

System Design

Our team performed a review of the design of the zk-SNARK circuit (the prover) and examined how the prover component fits into the design of the DeGate system at large. We found that the current design of the DeGate system requires that the operator process user transactions and send them on to the prover. As a result, the operator is able to censor specific transactions. To remediate this, we recommend considering a less centralized operator model as an alternative (Issue B).

Code Quality

Given the challenges of writing complex zk-SNARK systems, the DeGate team organized the code well by using appropriate classes and abstraction where needed. However, since the high-level circuits and gadgets rely on several tiers of sub-gadgets, navigating the hierarchy to clearly understand the functionality of the circuits is challenging nonetheless. Our team noted that the `ToBitsGadget` wraps the `DualVariableGadget`. However, it is not entirely clear why the functionality of the `DualVariableGadget` in this case requires a separate class. In addition, the signature verification gadgets are more nested than is necessary.

The `TransferCircuit` implementation includes functionality to require a generated proof for a transfer transaction to be verified on-chain by the DeGate smart contracts. However, the smart contracts will only verify special transaction types, which do not include transfers. This functionality is redundant and can be used to grief the smart contracts or perform a Denial of Service (DoS) attack. We recommend that the redundant functionality be removed (Issue A).

Due to a missing check needed to verify the token ID of the protocol transaction fee payment, a user may pay reduced fees by providing the token ID of the less valuable token in the exchange pair. We recommend the implementation of a check to verify the token ID (Issue C).

Tests

We found that sufficient test coverage has been implemented.

Documentation

We found that the project is missing proper documentation and does not provide any explanation of proper specification and rationale for design choices. Additionally, our team noted that the whitepaper was only partially helpful in facilitating reasoning about the implementation, as the coded implementation significantly deviates from the design described in the whitepaper. We recommend significantly expanding the documentation to include an extensive description of the specification as well as a rationale for system design choices.

Code Comments

Our team found that the lack of code comments and implementation details in the documentation inhibited our ability to perform the security review as efficiently and effectively as is ideal. For complex implementations, comprehensive code comment coverage is imperative to reason about the functionality and security characteristics of the system. The lack of code comments and implementation documentation evokes security by obfuscation, which is not a valid approach to securing systems potentially controlling large amounts of value. We recommend that the code comments in the repository be rewritten to comprehensively describe the intended behavior of each function and component and how those components interact with each other (Issue D).

Scope

The scope for this security audit was sufficient for a comprehensive review of the zk-SNARK design and implementation. However, the DeGate Order Book system is composed of several components, which were out of scope for this review. Specifically, the operator component was out of scope for this audit and the component was assumed to function as intended. However, issues in, and improper implementation of, the operator component would result in the generation of false proofs that would be rejected by the on-chain verifier. As a result, operator security is not critical to the security of the overall system, except in the context of censorship resistance.

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
Issue A: TransferCircuit May Be Used for DoS Attack	Resolved
Issue B: DeGate is Not Censorship Resistant	Unresolved
Issue C: Missing Check Allows Users to Pay Lower Fees	Resolved
Issue D: Code Comments Coverage Critically Insufficient	Unresolved

Issue A: TransferCircuit May Be Used for DoS Attack

Location

`Circuits/TransferCircuit.h#L351-L355`

Synopsis

The DeGate system distinguishes between on-chain and off-chain verification. Special transactions need on-chain verification. Only deposits, account updates, and withdrawals are considered to be special transactions from the smart contract verifier side. From the prover side, a user can request an on-chain verification for a transfer transaction. The disagreement between prover and verifier about the validity of proofs can be used for a DoS attack. The code in `TransferCircuit`, which allows for this behavior, is redundant.

Impact

A malicious attacker can use this to waste the resources of the prover or perform a DoS attack.

Feasibility

The attack is feasible given that the operator code allows the on-chain verification of a transfer transaction.

Technical Details

A malicious attacker sends a transfer transaction with a non-zero type. The circuit code then constrains the proof to require an on-chain signature verification and, as a result, the number of conditional transactions increases. From the perspective of the prover, a valid proof is generated. However, the proof cannot be verified by the smart contract if the transaction is not of the special transaction type deposit, account update, or withdrawal.

Remediation

We recommend removing the redundant code in `TransferCircuit` to disable this behavior.

Status

The DeGate team has added a gadget enforcing the type to be zero.

Verification

Resolved.

Issue B: DeGate is Not Censorship Resistant

Location

`loopring_v3/DESIGN.md#operators`

`loopring_v3/DESIGN.md#exchanges`

Synopsis

The DeGate prover relies on a single central operator that processes transactions from users and sends them to the prover in the form of blocks. As a result, the operator has the power to censor transactions, e.g. for specific users or token types. This undermines the trustlessness assumptions of Ethereum and contradicts the design goal of the DeGate system, as stated in the DeGate Whitepaper.

Impact

The operator can use its central position to exclude specific users, tokens, or transactions. Only the forced withdrawal transaction functionality prevents user funds from being locked up in the system.

Mitigation

We recommend that the security implications of a central operator model be clearly and explicitly communicated to the users.

Remediation

The design of the DeGate system is based on a central operator model. As such, no direct remediative action is possible. This would require reconsidering the design of the system and re-implementing significant critical functionality. In order to remediate this issue, we recommend the system allow a registration mechanism that permits arbitrary users to become operators or provers. Multiple operators could then compete concurrently in order to get their proof onto the chain.

Status

The DeGate team has responded that at the current stage of development, they expect the system to be trustless but not necessarily permissionless. The decentralization of the operator is on the roadmap.

Verification

Unresolved.

Issue C: Missing Check Allows Users to Pay Lower Fees

Location

`Gadgets/BatchOrderGadgets.h`

`Gadgets/MatchingGadgets.h#L183`

`DESIGN.md#batch-spot-trade`

Synopsis

The function `BatchSpotTrade` requires that operator fees be paid as a percentage of the amount of the bought token. However, there is no constraint that enforces `tokenB_ID` to be equal to `feeToken_ID`. Hence, a user could exploit this by paying in tokenS instead of tokenB, resulting in the loss of fees for the operator.

Impact

The operator loses fees through this attack.

Feasibility

The attack is always feasible but it only makes sense economically in the case that tokenB is more valuable than tokenS.

Technical Details

If tokenB is more valuable than tokenS, the fee can be paid in tokenS but as a percentage of the amount of tokenB. Assume amountB = 10 and amountS = 100 in the order and that the fee is 10%. If the user sends as `feeTokenID` the ID of tokenS, then the circuit calculates the amount to pay based on amountB, which would give a fee of 1. But since the fee is paid in tokenS, the user would pay only 1% in tokenS.

With larger differences in value between tokens, the user can exploit this issue to save operator fees.

Remediation

We recommend implementing a check on the feeTokenID in `FeeCalculatorGadget`.

Status

The DeGate team has responded that the feeTokenID is for the gas fee and can be in tokenS or tokenB. As such, we determined this is not a valid issue.

Verification

Resolved.

Issue D: Code Comments Coverage Critically Insufficient

Synopsis

We found that the code comments are insufficient in a considerable number of areas in the code and that the existing descriptive comments require further clarification. DeGate is a highly complex system and the current lack of comments and explanations introduces a level of "security by obfuscation" that we consider dangerous. This issue greatly blocks the ability to audit the code in all its details because an unnecessary and disproportionate amount of time has to be spent on understanding the functionality itself rather than searching for, and identifying, security vulnerabilities.

Comments on larger circuits should reference the associated detailed description in the specification. Additionally, gadgets should have comments that describe the required assumptions, guarantees, and algorithms used. In most parts, variables are used that are never explained or specified outside of the code.

Code comments within the codebase are critical for developers and reviewers, as they help to define and explain the purpose of each gadget and provide a description of their intended functionality.

Impact

The lack of code comments and explanation in DeGate severely inhibits the ability to efficiently and effectively audit the codebase.

Remediation

We recommend that the DeGate team not just expand code comment coverage but also edit existing comments for clarity and update the gadget comments such that they describe the intended behavior. The goal must be to have comments that clearly explain the functionality implemented in the codebase.

Status

The DeGate team has responded that they will implement the remediation in future releases.

Verification

Unresolved.

Issue E: zk-SNARK Statement Specification is Erroneous and Missing Components

Location

`circuit/statements.md`

Synopsis

zk-SNARKS are short and computationally sound proofs for the existence of witnesses to given *statements*. For effective review of the security of zk-SNARK engineering, it is fundamental to start with a clear and formal definition of the zk-SNARK statement. Without this definition, there is no foundation to compare the implementation against.

The current statement of the DeGate circuit is incomplete and several sub-gadgets are missing completely. Additionally, we found the statement to be erroneous in some instances, for example, the false signature verification in `AccountUpdate`. Implementing these erroneous statements into the code would have severe consequences for the security of the circuit.

Impact

Without a complete and accurate statement, there is no meaningful frame of reference with which to reason about the security of any implementation. As such, the severity of the security implications are unknown.

Technical Details

We identified subtle errors in the statement specification that could cause considerable harm. For example, we found the following errors (non-exhaustive):

- The AccountUpdate statement specification enforces either no signature or both on-chain and off-chain verification. The code, instead, is implemented correctly and leads to either on-chain or off-chain verification of signatures. Also, `nonce` is missing in the statement specification;
- The Transaction statement specification uses a `FromBitsGadget` for `accountA` but the codebase does not;
- The Transaction statement specification misses the input `state.oper.account.storageRoot` for `root_new`;
- There are inconsistent numbers in the statement specification for `Float32Encoding`; and
- The OrderCancel statement specification has `tokenID` and `owner` as input (in `hash` and `output.DA`) but the codebase does not. Additionally, `maxFee` and `useAppKey` are missing in the `OrderCancel` statement specification.

Many sub-gadgets are missing completely in the statement specification. Others exist in the statement specification but are not updated according to the changes implemented after the Loopring fork. For example, we identified the following out-of-date and missing statement specifications (non-exhaustive):

- Missing statement specifications for sub-gadgets for `BatchSpotTrade` circuit;
- Missing statement specification for `SignatureVerifier` and `BatchSignatureVerifier` sub-gadgets;
- Missing statement specification for constants, e.g. `NUM_BITS_AMOUNT_MAX`, `NUM_BITS_BATCH_SPOTRADE_TOKEN_TYPE_PAD`, and `Float 31 Encoding`; and
- Missing statement specification for `OrderCancelledNonceGadget` sub-gadget for `OrderCancel` circuit.

Remediation

We recommend the DeGate team update the statement specification to have an extensive, rigorous, and correct statement definition, adhering to the best practices of zk-SNARK engineering. In addition, institute regular documentation reviews to ensure the documentation remains up-to-date and consistent with the implementation.

Status

The DeGate team has responded that they will implement the remediation in future releases.

Verification

Unresolved.

Suggestions

Suggestion 1: Remove Redundant Code

Location

`loopring_v3/DESIGN.md#proof-generation-cost`

Synopsis

The DeGate zk-SNARK circuit is a fork of Loopring 3.6, implementing new functionalities and removing old ones. There are several places in the code that are artifacts from the removed Loopring code. According to the documentation, prover time is an estimated 7 minutes. Redundant code leads to further inefficiency in prover time. An example is described in Issue A where redundant code in `TransferCircuit` can be used to attack the system.

Mitigation

We recommend removing redundant code to improve the efficiency of the circuit and reduce the number of redundant circuits. In addition, this improves code readability and further strengthens the security of the DeGate zk-SNARK circuit.

Status

The DeGate team has responded that the redundant code has little impact on the circuit and they want to avoid code modifications. As such, the mitigation remains unresolved at the time of the verification.

Verification

Unresolved.

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.