DeGate

Security Assessment

April 20, 2023

Prepared for:
DeGate DAO
DeGate

Prepared by:
Devashish Tomar, Shaun Mirani, and Will Song

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 80+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and mutually agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

DeGate engaged Trail of Bits to review the security of its smart contracts. From February 7 to February 22, 2022, a team of three consultants conducted a security review of the client-provided source code, with six person-weeks of effort. On March 7, 2022, we started an additional week of review focused on the circuits and gadgets built using `ethsnarks`. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with partial knowledge of the target system, including access to the circuit source code, smart contract source code, and documentation. We performed automated analysis and a manual review of the code, in addition to running system elements.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. These include incorrect ERC20 token handling, undefined behavior in circuits, and unclear gadget limitations.

This report includes the findings of a December 2021 audit of the DeGate protocol. A summary of the findings of both audits is provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	4
Medium	3
Low	1
Informational	2
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Configuration	1
Data Validation	3
Cryptography	1
Undefined Behavior	6

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Shaun Mirani, Consultant
shaun.mirani@trailofbits.com

Devashish Tomar, Consultant
devashish.tomar@trailofbits.com

Will Song, Consultant
will.song@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
February 7, 2022	Pre-project kickoff call
February 11, 2022	Delivery of status report
February 22, 2022	Delivery of final report; report readout meeting
March 30, 2022	Delivery of public report
April 20, 2023	Delivery of updated public report

Project Goals

The engagement was scoped to provide a security assessment of the DeGate smart contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Do the system components have appropriate access controls?
- Is it possible to manipulate the system by front-running transactions?
- Is it possible for participants to steal or lose tokens or shares?
- Are there any circumstances under which arithmetic errors could affect the system?
- Are any of the system components vulnerable to denial-of-service attacks, and could any be used in phishing?
- Does the exchange bookkeeping arithmetic hold?
- Are critical events logged?
- Are the off-chain computations that use circuits implemented correctly?
- Is each circuit variable in `generate_r1cs_constraints` properly constrained?
- Does every circuit and gadget correctly implement the corresponding zero-knowledge statement?
- Do the circuits use the low-level gadgets correctly and safely?

Project Targets

The engagement involved a review and testing of the following target.

DeGate Protocol

Repository	https://github.com/degatedev/degate-protocols

Version	`7d5e66a1e22d7657c888463256e0175119745621`
	Commit used for auditing
	
	`e934d99da5a3bc7dd078c0d5c9831b3b31ea97c4`
	Chinese character removal for page 17

Type	Decentralized Exchange
Platforms	Solidity, C++

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- A review of the addition and handling of ERC20 tokens revealed several issues that could affect the DeGate exchange's behavior and users; these include a failure to ensure that tokens are registered to the appropriate list (i.e., the normal or reserved list) (TOB-DeGate-1) and the incorrect handling of non-standard tokens (TOB-DeGate-4).

- Analysis of the circuits revealed undefined behavior (TOB-DeGate-11), interoperability issues caused by the use of JSON integers (TOB-DeGate-7), and gadget limitations (TOB-DeGate-9). Detailed analysis of the circuits' correctness yielded a finding regarding the range checks of a gadget used in fee calculations (TOB-DeGate-2).

- A manual review of the processing of block deposits and withdrawals enabled us to assess the ways in which assets flow into the DeGate protocol and users interact on-chain; we did not find any issues.

- Validation of the external interactions did not reveal any reentrancy risks.

- A review of the functions for front-running opportunities did not reveal any critical concerns, although a user could front-run a call to an initialization function to disrupt a contract's deployment (TOB-DeGate-5).

- A review of the issues discovered in our previous DeGate audit determined that several circuit and smart contract issues remain unresolved (TOB-DeGate-4, TOB-DeGate-5, TOB-DeGate-6, TOB-DeGate-7, TOB-DeGate-8, TOB-DeGate-9, TOB-DeGate-10, and TOB-DeGate-11).

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of certain system elements, which may warrant further review. The following is a summary of the coverage limitations of the engagement.

- We performed a limited review of the base cryptography used by the `poseidon` and `ethsnarks` circuits and their dependencies as well as the user / validator key management system.

- The operator codebase was out of scope.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase generally uses `SafeMath` functions to perform calculations, and we did not identify any potential overflows in places in which those functions are not used.	**Satisfactory**
Auditing	Most of the functions emit events where appropriate. The events emitted by the code are sufficient for monitoring on-chain activity. However, we did not have access to an incident response plan or information on the use of off-chain components in behavior monitoring.	**Satisfactory**
Authentication / Access Controls	We did not identify any serious access control issues.	**Satisfactory**
Complexity Management	Many of the system's functionalities, especially the block-processing functionalities, are broken up into multiple functions. This makes the code less readable and more difficult to modify.	**Moderate**
Decentralization	Although the block submission process is centralized and the owner has the ability to shut down the exchange, users can force certain actions. For example, if a user's deposit into the `ExchangeV3` contract is never picked up by a validator and included in a block, the user can recover the funds by submitting an on-chain transaction. There is a similar mechanism for withdrawals: by executing a withdrawal that will not be timely picked up by a validator, a user could force `ExchangeV3` to enter	**Moderate**

	withdrawal mode, preventing the validator from submitting more blocks and allowing users to withdraw their funds.	
Documentation	The protocol is a fork of Loopring v3.6.1 and therefore has a significant amount of documentation. DeGate also provided documentation regarding changes to the original code; however, that documentation does not identify all of the changes or all of the features that were added or removed.	**Weak**
Front-Running Resistance	We found one issue related to front-running (TOB-DeGate-5). However, time constraints prevented us from exhaustively checking the protocol for front-running and unintended arbitrage opportunities.	**Further Investigation Required**
Low-Level Manipulation	A substantial amount of assembly is used throughout the math and utility libraries (e.g., `Poseidon`); however, time constraints prevented us from testing or verifying the libraries.	**Further Investigation Required**
Testing and Verification	We identified some failing unit tests. Certain of these tests fail because they rely on parts of the original Loopring code that have been removed by the DeGate team; other test cases were not properly set up or fail for other reasons.	**Weak**
Upgradeability	Contracts including `ExchangeV3` and `DefaultDepositContract` support proxy patterns for upgrades. `DefaultDepositContract` is used with a proxy pattern in some tests (`testExchangeUtil.ts`); however, during our first call, the DeGate team expressed that it intends to remove this feature. For the time being, though, it is possible to make certain of the contracts upgradeable.	**Further Investigation Required**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Lax boundaries between normalTokens and reservedTokens arrays	Undefined Behavior	Medium
2	Missing range checks in MulDivGadget	Cryptography	High
3	Poor code management practices	Undefined Behavior	Informational
4	Token management difficulties caused by the addition of arbitrary tokens	Data Validation	High
5	Initialization functions can be front-run	Configuration	Low
6	Circuit crashes when invalid blocks are generated by the operator	Undefined Behavior	High
7	Saving large JSON integers could result in interoperability issues	Undefined Behavior	Medium
8	Lack of contract existence check on delegatecall will result in unexpected behavior	Data Validation	Medium
9	Numerical comparison gadget does not support very large numbers	Data Validation	High
10	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
11	Circuits rely on undefined behavior in libff	Undefined Behavior	Undetermined

Detailed Findings

1. Lax boundaries between normalTokens and reservedTokens arrays	
Severity: **Medium**	Difficulty: **Medium**
Type: Undefined Behavior	Finding ID: TOB-DeGate-1
Target: `ExchangeV3.sol`, `ExchangeTokens.sol`	

Description

Any user can register a new token explicitly, by using the `registerToken` function, or implicitly, by making a deposit to a new token address. If the caller of `registerToken` is not the contract's owner, the token is registered in the `normalTokens` array; if the caller is the owner, it is registered in the `reservedTokens` array.

```
    function registerToken(
        address tokenAddress
        )
        external
        override
        nonReentrant
        returns (uint32)
    {
        return state.registerToken(tokenAddress, msg.sender == owner);
    }
```

Figure 1.1: The `registerToken` function in ExchangeV3

By front-running a contract owner's `registerToken` transaction, one could cause the token to be registered in the `normalTokens` array. Moreover, once a token has been registered in an array, it cannot be removed.

```
    function registerToken(
        ExchangeData.State storage S,
        address tokenAddress,
        bool isOwnerRegister
        )
        public
        returns (uint32 tokenID)
    {
        require(!S.isInWithdrawalMode(), "INVALID_MODE");
```

```
        require(S.tokenToTokenId[tokenAddress] == 0, "TOKEN_ALREADY_EXIST");

        if (isOwnerRegister) {
            require(S.reservedTokens.length < ExchangeData.MAX_NUM_RESERVED_TOKENS,
"TOKEN_REGISTRY_FULL");
        } else {
            require(S.normalTokens.length < ExchangeData.MAX_NUM_NORMAL_TOKENS,
"TOKEN_REGISTRY_FULL");
        }

        // Check if the deposit contract supports the new token
        if (S.depositContract != IDepositContract(0)) {
            require(S.depositContract.isTokenSupported(tokenAddress), "UNSUPPORTED_TOKEN");
        }

        // Assign a tokenID and store the token
        ExchangeData.Token memory token = ExchangeData.Token(tokenAddress);

        if (isOwnerRegister) {
            tokenID = uint32(S.reservedTokens.length);
            S.reservedTokens.push(token);
        } else {
            tokenID =
uint32(S.normalTokens.length.add(ExchangeData.MAX_NUM_RESERVED_TOKENS));
            S.normalTokens.push(token);
        }
        S.tokenToTokenId[tokenAddress] = tokenID + 1;
        S.tokenIdToToken[tokenID] = tokenAddress;

        S.tokenIdToDepositBalance[tokenID] = 0;

        emit TokenRegistered(tokenAddress, tokenID);
    }
```

Figure 1.2: The `registerToken` function in `ExchangeTokens`

This issue does not have a significant effect on the protocol; however, if the functionality of `reservedTokens` is expanded, it may pose a risk to the protocol.

Exploit Scenario

Alice, the owner of the `ExchangeV3` contract, sends a transaction to register token X as a reserved token. Bob, a griefer, notices Alice's transaction and front-runs it to register token X as a normal token.

Recommendations

Short term, change the approach to registering reserved tokens to remove the risk of front-running.

Long term, thoroughly document the purpose of reserved and normal tokens, and review their effects on the functionality of the protocol.

2. Missing range checks in MulDivGadget

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-DeGate-2
Target: `packages/loopring_v3/circuit/Gadgets/MathGadgets.h`	

Description

The zero-knowledge `MulDivGadget` is used in the `SpotTradeCircuit` and `BatchOrderGadget` to calculate fees. The gadget takes `numBitsValue`, `numBitsNumerator`, and `numBitsDenominator` arguments indicating the bit size of the value, numerator, and denominator but does not validate those arguments; nor are the values used to validate the bit size of the quotient, which should be equal to `numBitsValue + numBitsNumerator - numBitsDenominator`. This lack of validation could cause a fee calculation to overflow, which could have catastrophic consequences.

```
MulDivGadget(
  ProtoboardT &pb,
  const Constants &constants,
  const VariableT &_value,
  const VariableT &_numerator,
  const VariableT &_denominator,
  unsigned int numBitsValue,
  unsigned int numBitsNumerator,
  unsigned int numBitsDenominator,
  const std::string &prefix)
    : GadgetT(pb, prefix),
      value(_value),
      numerator(_numerator),
      denominator(_denominator),
      quotient(make_variable(pb, FMT(prefix, ".quotient"))),
      denominator_notZero(pb, denominator, FMT(prefix, ".denominator_notZero")),
      product(pb, value, numerator, FMT(prefix, ".product")),
      // Range limit the remainder. The comparison below is not guaranteed to
      // work for very large values.
      remainder(pb, numBitsDenominator, FMT(prefix, ".remainder")),
      remainder_lt_denominator(
        pb,
        remainder.packed,
        denominator,
        numBitsDenominator,
        FMT(prefix, ".remainder < denominator"))
{
    assert(numBitsValue + numBitsNumerator <= NUM_BITS_FIELD_CAPACITY);
}
```

Figure 2.1: The `MulDivGadget` constructor

The gadget should also check the assertion that the product is bounded. A motivated attacker could easily bypass the assertion by commenting it out, which would not change the final circuit but would prevent a crash.

Fee calculations currently use a constant denominator of 10,000 from the set of constants, but future updates could render it variable. Moreover, in the `SpotTradeCircuit`, the values of `fillS_A.value()` and `fillS_B.value()` appear to be relatively unconstrained, which is a cause for concern.

```
feeCalculatorA(
  pb,
  state.constants,
  fillS_B.value(),
  // split trading fee and gas fee
  // state.protocolTakerFeeBips,
  orderA.feeBips.packed,
  FMT(prefix, ".feeCalculatorA")),
feeCalculatorB(
  pb,
  state.constants,
  fillS_A.value(),
  // split trading fee and gas fee
  // state.protocolMakerFeeBips,
  orderB.feeBips.packed,
  FMT(prefix, ".feeCalculatorB")),
```

Figure 2.2: Fee calculations in the `SpotTradeCircuit`

In the `BatchOrderGadget`, these values appear to be properly constrained, as `deltaFilledB` is a `DualVariableGadget` of `NUM_BITS_AMOUNT` bits.

```
tradingFeeCalculator(
    pb,
    constants,
    deltaFilledB.packed,
    order.feeBips.packed,
    FMT(prefix, ".tradingFeeCalculator")),
```

Figure 2.3: A fee calculation in the `BatchOrderGadget`

Exploit Scenario

An attacker leverages a field multiplication overflow to reduce the fees charged for the attacker's spot trades. Because the quotient bit size is not checked, the attacker may be able to evade the fees entirely (and disrupt other arithmetic operations) by causing the system to set a negative fee amount.

Recommendations

Short term, redesign the `MulDivGadget` to use `DualVariableGadgets` to validate the bit size of its arguments, and use a `SafeMulGadget` to validate the bit size of the product.

Long term, carefully check each gadget and circuit to ensure that every field is properly validated. While obtaining an external audit of the SNARK code is a good step, it is not unheard of for bugs to slip through regardless; Zcash, for example, experienced a catastrophic SNARK failure a few years ago despite having been audited.

3. Poor code management practices	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-DeGate-3
Target: Throughout the codebase	

Description

The changes made to the DeGate protocol for version 0.2.0 were implemented through only a few commits. Most of the changes were consolidated and introduced under a single commit, resulting in very large unorganized code diffs.

A very large diff under one commit can cause confusion. It can also make it more difficult to review the code and to identify which functionality is most affected by a change. For example, one commit (`d111d7e19b1466dfdc68de508de3abd4faedf765`) introduced almost 20,000 changes across various functionalities of the platform.

These practices increase the likelihood of latent bugs in the codebase. If new bugs are introduced, which is also likely, the mishandled revision control will make them more difficult to identify.

Recommendations

Short term, make changes to the codebase in smaller more digestible commits with descriptive commit messages.

Detailed Findings (Previous Audit)

4. Token management difficulties caused by the addition of arbitrary tokens

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-DeGate-4
Target: `ExchangeV3.sol`, `ExchangeDeposits.sol`, `DefaultDepositContract.sol`	

Description

Transfers and balance-change operations involving certain tokens require special verification; if the owner of a contract does not carefully monitor the contract when one such token is added, those operations can unexpectedly fail.

Any user can register a new token explicitly, by using the `registerToken` function (figure 4.1.), or implicitly, by making a deposit to a new token address (figure 4.2).

```
function registerToken(
    address tokenAddress
    )
    external
    override
    nonReentrant
    returns (uint32)
{
    return state.registerToken(tokenAddress);
}
```

Figure 4.1: The `registerToken` function in ExchangeV3

```
function deposit(
    ExchangeData.State storage S,
    address from,
    address to,
    address tokenAddress,
    uint96  amount,                 // can be zero
    bytes   memory extraData
    )
    internal  // inline call
{
    require(to != address(0), "ZERO_ADDRESS");
```

```
        // Deposits are still possible when the exchange is being shutdown, or even in
withdrawal mode.
        // This is fine because the user can easily withdraw the deposited amounts again.
        // We don't want to make all deposits more expensive just to stop that from
happening.

        (uint32 tokenID, bool tokenFound) = S.findTokenID(tokenAddress);
        if(!tokenFound) {
            tokenID = S.registerToken(tokenAddress);
        }
        ...
```

Figure 4.2: The `deposit` function in `ExchangeDeposits`

Additionally, some tokens perform custom transfer logic and must be subject to a special check:

```
    function deposit(
        address from,
        address token,
        uint96  amount,
        bytes   calldata /*extraData*/
        )
        external
        override
        payable
        onlyExchange
        ifNotZero(amount)
        returns (uint96 amountReceived)
    {
        uint ethToReturn = 0;

        if (isETHInternal(token)) {
            ...
        } else {
            // When checkBalance is enabled for a token we check the balance change
            // on the contract instead of counting on transferFrom to transfer exactly
            // the amount of tokens that is specified in the transferFrom call.
            // This is to support non-standard tokens which do custom transfer logic.
            bool checkBalance = needCheckBalance[token];
            uint balanceBefore = checkBalance ? ERC20(token).balanceOf(address(this)) : 0;

            token.safeTransferFromAndVerify(from, address(this), uint(amount));

            uint balanceAfter = checkBalance ? ERC20(token).balanceOf(address(this)) :
amount;
            uint diff = balanceAfter.sub(balanceBefore);
            amountReceived = diff.toUint96();
```

```
            ethToReturn = msg.value;
        }
        ...
```

Figure 4.3: The `deposit` function in `DefaultDepositContract`

The owner of the deposit contract should explicitly enable this check by using the `setCheckBalance` function.

```
    function setCheckBalance(
        address token,
        bool    checkBalance
        )
        external
        onlyOwner
    {
        require(needCheckBalance[token] != checkBalance, "INVALID_VALUE");

        needCheckBalance[token] = checkBalance;
        emit CheckBalance(token, checkBalance);
    }
```

Figure 4.4: The `setCheckBalance` function in `DefaultDepositContract`

However, since any user can add any token at any time, it is virtually impossible for an owner to verify which tokens require such a check.

Exploit Scenario

Alice makes a deposit of a new deflationary token, the balance of which must be carefully checked. Unless the owner of the contract is able to front-run Alice's transaction to enable `checkBalance`, her deposit will be processed without that balance check, leading to unexpected behavior.

Recommendations

Short term, ensure that users are aware of this limitation or consider disallowing the addition of arbitrary tokens.

Long term, review our Token Integration Checklist before deciding which tokens should be added to the protocol.

5. Initialization functions can be front-run

Severity: **Low**	Difficulty: **High**
Type: Configuration	Finding ID: TOB-DeGate-5
Target: `ExchangeV3.sol`, `DefaultDepositContract.sol`	

Description

The `ExchangeV3` and `DefaultDepositContract` contracts have initialization functions that can be front-run, allowing an attacker to incorrectly initialize the contracts.

```
function initialize(
    address _loopring,
    address _owner,
    bytes32 _genesisMerkleRoot,
    bytes32 _genesisMerkleAssetRoot
    )
    external
    override
    nonReentrant
    onlyWhenUninitialized
{
    require(address(0) != _owner, "ZERO_ADDRESS");
    owner = _owner;
    loopringAddr = _loopring;

    state.initializeGenesisBlock(
        _loopring,
        _genesisMerkleRoot,
        _genesisMerkleAssetRoot,
        EIP712.hash(EIP712.Domain("DeGate Protocol", version(), address(this)))
    );
}
```

Figure 5.1: ExchangeV3's `initialize` function

```
function initialize(
    address _exchange
    )
    external
{
    require(
        exchange == address(0) && _exchange != address(0),
```

```
            "INVALID_EXCHANGE"
        );
        owner = msg.sender;
        exchange = _exchange;
    }
```

Figure 5.2: `DefaultDepositContract`'s `initialize` function

Neither of these functions is protected by access controls; this is because the functions are meant to set the initial state of the contracts, since the contracts both support a proxy architecture. As such, an attacker could front-run these functions and initialize the contracts with malicious values.

Exploit Scenario

Alice deploys the `ExchangeV3` contract. Eve front-runs the contract's initialization and sets her own address as the owner. As a result, she gains access to owner privileges and can perform actions such as changing the settings of the exchange and withdrawing fees to her own address.

Recommendations

Short term, add proper access controls to the initializer functions to ensure that they are callable only by contract owners.

Long term, carefully review the initializers across the codebase to ensure that they have proper access controls.

6. Circuit crashes when invalid blocks are generated by the operator	
Severity: **High**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-DeGate-6
Target: Circuits	

Description

Users can force the operator to produce invalid blocks that will crash the circuits.

To maintain strong security guarantees and high performance, DeGate requires that certain computations be performed from the `ethsnarks` circuits. The operator produces blocks in the form of JSON files, and validators process them using the `dex_circuit` program. However, the operator can produce invalid blocks under certain conditions.

For instance, if an order cancellation is generated with an invalid signature, the operator will produce an invalid block with a negative balance.

```
"rootBefore":
"16338061663996780208594963981079042825468976450837220348267387051031964921715",
"rootAfter":
"13168472117334653415272168230518893410152064571170296320262299419327863961379",
"before": {
    "balance": "0"
},
"after": {
    "balance": "-21000000000000"
}
```

Figure 6.1: Part of an invalid block

When the validator tries to parse the block (by using `dex_circuit`), the validator will crash. This crash is caused by an assertion failure when parsing negative numbers as big int values.

```
Dex_circuit: .../ethsnarks/depends/libsnark/depends/libff/libff/algebra/fields/bigint.tcc:33:
libff::bigint<4>::bigint(const char *) [n = 4]: Assertion `s[i] >= '0' && s[i] <= '9''
failed.
```

Figure 6.2: A validator crash

Exploit Scenario

Eve repeatedly submits invalid requests in DeGate to force the operator to generate invalid blocks. This degrades the validator's performance and may even cause it to crash.

Recommendations

Short term, ensure that the validator will never crash regardless of the values in JSON blocks.

Long term, review the testing approach to make sure that a crash in the validator produces a testing failure.

7. Saving large JSON integers could result in interoperability issues

Severity: **Medium**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-DeGate-7
Target: Circuits	

Description

The DeGate protocol's method of parsing user- and operator-generated JSON integers differs from that of mainstream implementations such as `NodeJS` and `jq`.

The JSON standard warns about certain "interoperability problems" in numeric types outside the range `[-(2**53)+1, (2**53)-1]`. These issues are caused by widely used JSON implementations that use `IEEE 754` (double-precision) numbers to implement integers.

For instance, if the operator saved the `amount` value `"1152921504606846976" (2**60)`, it would be serialized as the expected value `1152921504606846976`. However, web browsers, `NodeJS`, and `jq 1.5` would parse it as `1152921504606847000`.

```
[
  {
    ...
    "amount": 1152921504606846976,
    ...
  }
]
```

Figure 7.1: Part of a JSON file

This parsing affects fields such as `fFillS_A` and `fFillS_B`, which are parsed directly from JSON numbers, without the use of strings:

```
static void from_json(const json &j, SpotTrade &spotTrade)
{
    spotTrade.orderA = j.at("orderA").get<Order>();
    spotTrade.orderB = j.at("orderB").get<Order>();
    spotTrade.fillS_A = ethsnarks::FieldT(j["fFillS_A"]);
    spotTrade.fillS_B = ethsnarks::FieldT(j["fFillS_B"]);
}
```

Figure 7.2: The `from_json` function in `circuit/Utils/Data.h`

Exploit Scenario

Alice, a DeGate user, inputs a large number that will be processed by the operator and included in a JSON file block. The `dex_circuit` program reads Alice's JSON value and interprets it differently than Alice, resulting in behavior that she did not expect.

Recommendations

Short term, use strings instead of JSON numeric values to implement the `amount` field. This will prevent any ambiguity when parsing the numeric fields of JSON file blocks.

Long term, use a recommended JSON approach when interacting with JSON files generated by the operator. This will prevent any ambiguity when parsing the numeric fields of JSON files.

References

Numbers in JSON (RFC 8259, Section 6)

8. Lack of contract existence check on delegatecall will result in unexpected behavior

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-DeGate-8
Target: `thirdparty/proxies/Proxy.sol`	

Description

The `Proxy` contract uses the `delegatecall` proxy pattern. If the implementation contract is incorrectly set or is self-destructed, the `Proxy` contract may not be able to detect failed executions.

The `Proxy` contract includes a payable fallback function that is invoked when proxy calls are executed. This function lacks a contract existence check (figure 8.1).

```
function _fallback() private {
    address _impl = implementation();
    require(_impl != address(0));

    assembly {
        let ptr := mload(0x40)
        calldatacopy(ptr, 0, calldatasize())
        let result := delegatecall(gas(), _impl, ptr, calldatasize(), 0, 0)
        let size := returndatasize()
        returndatacopy(ptr, 0, returndatasize())

        switch result
        case 0 { revert(ptr, size) }
        default { return(ptr, size) }
    }
}
```

Figure 8.1: The `_fallback` function in `Proxy.sol`

A `delegatecall` to a destructed contract will return success (figure 8.2). Due to the lack of contract existence checks, a series of batched transactions may appear to be successful even if one of the transactions fails.

```
The low-level functions call, delegatecall and staticcall return true as their first
return value if the account called is non-existent, as part of the design of the
EVM. Account existence must be checked prior to calling if needed.
```

Figure 8.2: A snippet of the Solidity documentation detailing unexpected behavior related to `delegatecall`

Exploit Scenario

Eve upgrades the proxy to point to an incorrect new implementation. As a result, each `delegatecall` returns success without changing the state or executing code. Eve uses this defect to scam users.

Recommendations

Short term, implement a contract existence check before each `delegatecall`. Document the fact that using `suicide` and `selfdestruct` can lead to unexpected behavior, and prevent future upgrades from using these functions.

Long term, carefully review the Solidity documentation, especially the "Warnings" section, and the pitfalls of using the `delegatecall` proxy pattern.

References

- Contract Upgrade Anti-Patterns
- Breaking Aave Upgradeability

9. Numerical comparison gadget does not support very large numbers

Severity: **High**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-DeGate-9
Target: `circuit/Gadgets/MathGadgets.h`	

Description

The `LeqGadget` performs various comparison operations over numeric values but does not support the full range of `uint256` values:

```
// (A <(=) B)
class LeqGadget : public GadgetT
{
  ...
        // The comparison gadget is only guaranteed to work correctly on values in
        // the field capacity - 1
        ...

        ASSERT(n <= NUM_BITS_FIELD_CAPACITY - 1, prefix);
```

Figure 9.1: Part of the `LeqGadget` in `MathGadgets.h`

Since `NUM_BITS_FIELD_CAPACITY` is equal to 253, the use of numbers close to `2**256 - 1` will cause an assertion failure. Token deposit amounts are not affected by this limitation, since they are defined using the `uint248` type; however, other fields, such as the ones that track down the deposit state, could trigger a failure in the `LeqGadget`.

```
    struct BalanceLeaf
    {
        uint32   tokenID;
        uint248   balance;
    }

    ...

    struct DepositState {
        uint256 freeDepositMax;
        uint256 freeDepositRemained;
        uint256 lastDepositBlockNum;
        uint256 freeSlotPerBlock;
        uint256 depositFee;
    }
```

Figure 9.2: The `BalanceLeaf` and `DepositState` data structures in `ExchangeData.sol`

This problem was also raised in the Loopring repository.

Exploit Scenario

Alice triggers a circuit that uses the `LeqGadget` to perform a comparison with a large number that it does not support. This causes an assertion failure that crashes the validator.

Recommendations

Short term, disallow the use of numbers greater than the capacity of the bits of the field, and document this limitation for users.

Long term, review the limitations of each gadget to ensure that it will not block any user operations, and document those limitations.

10. Solidity compiler optimizations can be problematic

Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-DeGate-10
Target: `truffle.js`	

Description
The DeGate protocol has enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the DeGate protocol contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

11. Circuits rely on undefined behavior in libff

Severity: **Undetermined**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-DeGate-11
Target: Circuits	

Description

Running the circuits to validate blocks triggers undefined behavior in the `libff` library.

To maintain strong security guarantees and high performance, DeGate requires that certain computations be performed from the `ethsnarks` circuits. The operator produces blocks in the form of JSON files, and validators process them using the `dex_circuit` program. If the circuits are compiled using `UndefinedBehaviorSanitizer`, unit testing will trigger numerous warnings in a `libff` function:

```
.../ethsnarks/depends/libsnark/depends/libff/libff/algebra/scalar_multiplication/multiexp.tcc
:178:80: runtime error: shift exponent 64 is too large for 64-bit type 'mp_limb_t' (aka
'unsigned long')
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior
.../ethsnarks/depends/libsnark/depends/libff/libff/algebra/scalar_multiplication/multiexp.tcc
:178:80 in
.../loopring_v3/ethsnarks/depends/libsnark/depends/libff/libff/algebra/scalar_multiplication/
multiexp.tcc:178:107: runtime error: shift exponent 64 is too large for 64-bit type
'unsigned long'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior
.../ethsnarks/depends/libsnark/depends/libff/libff/algebra/scalar_multiplication/multiexp.tcc
:178:107 in
```

Figure 11.1: `UndefinedBehaviorSanitizer` warnings

These warnings are caused by the following code:

```
static inline size_t get_id(size_t c, size_t bitno, const mp_limb_t* data)
{
    static const mp_limb_t one = 1;
    const mp_limb_t mask = (one << c) - one;
    const size_t limb_num_bits = sizeof(mp_limb_t) * 8;

    const size_t part = bitno / limb_num_bits;
    const size_t bit = bitno % limb_num_bits;
```

```
    size_t id = (data[part] & (mask << bit)) >> bit;
    //const mp_limb_t next_data = (bit + c >= limb_num_bits && part < 3) ?
bn_exponents[i].data[part+1] : 0;
    //id |= (next_data & (mask >> (limb_num_bits - bit))) << (limb_num_bits - bit);
    id |= (((bit + c >= limb_num_bits && part < 3) ? data[part+1] : 0) & (mask >>
(limb_num_bits - bit))) << (limb_num_bits - bit);

    return id;
}
```

Figure 11.2: `get?id`

Additionally, CodeQL identified potential integer overflows in the computations of `k*c` as arguments of `get_id`:

```
template<typename T, typename FieldT, bool with_density, bool prefetch, unsigned int
prefetch_locality>
T multi_exp_inner_bellman_with_density(
    …
    unsigned int c,
    unsigned int k,
    …)
{
        …
        if (prefetch)
        {
            // prefetch next bucket
            if (i < length - look_ahead)
            {
                size_t next_id = get_id(c, k*c, &exponents[i+look_ahead].data[0]);
                …
            }
        }
        size_t id = get_id(c, k*c, &exponents[i].data[0]);
        …
```

Figure 11.3: Part of the `multi_exp_inner_bellman_with_density` function

The affected code was introduced by Loopring in a fork of `libff` that should be carefully reviewed.

Exploit Scenario

The compiler used to produce the `dex_circuit` binary is changed or updated. The change results in undefined behavior that may be compiled in different ways, producing different results on different machines.

Recommendations

Short term, refactor the affected `libff` code to avoid triggering undefined behavior.

Long term, enable code sanitizers (e.g., `AddressSanitizer` and `UndefinedBehaviorSanitizer`) to ensure that no undefined behavior is invoked during testing.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Upgradeability	Related to contract upgradeability

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. See `crytic/building-secure-contracts` for an up-to-date version of the checklist.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, use the below output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of
Echidna and Manticore
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.

- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.

- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.

- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

D. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General Recommendations

- Consider transforming the following abstract contracts into interfaces:
  - `IBlockReceiver.sol`
  - `IAgentRegistry.sol`
  - `ILoopringV3Partial.sol`
  - `BurnableERC20.sol`
  - `ERC20.sol`
  - `IChiToken.sol`
- Consider removing the unused agent code from the codebase.
  - Refactor the parts of the code that check whether `msg.sender` is an agent (e.g., the `isUserOrAgent` function).
  - Consider removing the function `approveTransactions` in `ExchangeV3.sol`; it will always revert, because it relies on commented agent code.
- Consider making the `withdraw` function in `DefaultDepositContract.sol` non-payable.
- Consider removing the following dead / commented code:
  - The commented code in `Poseidon.sol`
  - The commented `AuxiliaryData` struct field in `ExchangeData.sol`
  - The import of `IAgentRegistry` in `ExchangeGenesis.sol` and `FastWithdrawalAgent.sol`
  - The commented code in the `beforeBlockSubmission` function in `LoopringIOExchangeOwner.sol`

    - `bytes extraData` in the `Withdrawal` and `WithdrawalAuxiliaryData` structs of `WithdrawTransaction.sol`
    - The commented code in the `_beforeBlockSubmission` function of the `LoopringIOExchangeOwner.sol` contract

- There is a typo in the annotation prefix for the `SelectOneTokenAmountGadget`. The `tokenX_neq_tokenA` gadget is assigned the prefix "`.tokenX_eq_tokenA`", which is the same as that of the `tokenX_eq_tokenA` gadget. This prefix should be replaced with "`.tokenX_neq_tokenA`".

E. Fix Log

On March 11, 2022, Trail of Bits reviewed the DeGate team's responses to issues identified in this report. The DeGate team acknowledged all 11 issues but did not fix any. Various fix commits contained additional changes not related to the fixes. We did not comprehensively review these changes. For additional information, please refer to the Detailed Fix Log.

ID	Title	Severity	Fix Status
1	Lax boundaries between normalTokens and reservedTokens arrays	Medium	Risk accepted by the client
2	Missing range checks in MulDivGadget	High	Risk accepted by the client
3	Poor code management practices	Informational	Risk accepted by the client
4	Token management difficulties caused by the addition of arbitrary tokens	High	Risk accepted by the client
5	Initialization functions can be front-run	Low	Risk accepted by the client
6	Circuit crashes when invalid blocks are generated by the operator	High	Risk accepted by the client
7	Saving large JSON integers could result in interoperability issues	Medium	Risk accepted by the client
8	Lack of contract existence check on delegatecall will result in unexpected behavior	Medium	Risk accepted by the client

9	Numerical comparison gadget does not support very large numbers	High	Risk accepted by the client
10	Solidity compiler optimizations can be problematic	Informational	Risk accepted by the client
11	Circuits rely on undefined behavior in libff	Undetermined	Risk accepted by the client

Detailed Fix Log

TOB-DeGate-1: Lax boundaries between normalTokens and reservedTokens arrays
Risk accepted by the client. The DeGate team provided the following rationale for its acceptance of this risk:

> "We decide to register as many reserved tokens as possible during the deployment phase to avoid this issue. `ReservedTokens` are tokens that users are allowed to use to pay fees."

TOB-DeGate-2: Missing range checks in MulDivGadget
Risk accepted by the client. The DeGate team provided the following rationale for its acceptance of this risk:

> "Muldivgadget is used to calculate the handling charge of spottrade and batchspottrade.

```
product(pb, value, numerator, FMT(prefix, ".product")),
assert(numBitsValue + numBitsNumerator <= NUM_BITS_FIELD_CAPACITY)
```

Figure E.1: The relevant code

> The length of `numBitsNumerator` in `FeeCalculatorGadget` is no more than 6 bits, while `NUM_BITS_FIELD_CAPACITY` = 253, the length of amount `numBitsValue < = 247`.
>
> That is, the maximum value supported is `247bits`, while the maximum value supported here is `96bits`. Therefore, there will be no problem for the time being. If there is a problem, the circuit will crash."

TOB-DeGate-3: Poor code management practices
Risk accepted by the client. The DeGate team provided the following rationale for its acceptance of this risk:

> "In internal repository, code changes are submitted based on user stories or issues."

TOB-DeGate-4: Token management difficulties caused by the addition of arbitrary tokens
Risk accepted by the client. The DeGate team provided the following rationale for its acceptance of this risk:

> "We will add a warning about this limitation on front-end."

TOB-DeGate-5: Initialization functions can be front-run

Risk accepted by the client. The DeGate team provided the following rationale for its acceptance of this risk:

> "Ignored. We will ensure the deployment and initialization are correct."

TOB-DeGate-6: Circuit crashes when invalid blocks are generated by the operator
Risk accepted by the client. The DeGate team provided the following rationale for its acceptance of this risk:

> "The wrong data provided by the operator will indeed lead to circuit crash, which is an expected phenomenon. If not, there will be other problems."

TOB-DeGate-7: Saving large JSON integers could result in interoperability issues
Risk accepted by the client. The DeGate team provided the following rationale for its acceptance of this risk:

> "The amount mentioned here is the filled value in SpotTrade. The value had been handled by float32 format. so just 32bits. There will be no problem. If it exceeds 32, the circuit will crash."

TOB-DeGate-8: Lack of contract existence check on delegatecall will result in unexpected behavior
Risk accepted by the client. The DeGate team provided the following rationale for its acceptance of this risk:

> "Ignored. We will ensure the deployment and initialization are correct."

TOB-DeGate-9: Numerical comparison gadget does not support very large numbers
Risk accepted by the client. The DeGate team provided the following rationale for its acceptance of this risk:

> "Due to FTT, `libsnark` supports a maximum amount of 252 power of 2. Now it has supported `248bits` in balance change, recharge and withdrawal.
>
> The total balance which deposited by user in smart contract is limited by `248bits`, so there will be no problem in the circuit."

TOB-DeGate-10: Solidity compiler optimizations can be problematic
Risk accepted by the client. The DeGate team provided the following rationale for its acceptance of this risk:

> "We will keep the current optimization switch and evaluate it later."

TOB-DeGate-11: Circuits rely on undefined behavior in libff

Risk accepted by the client. The DeGate team provided the following rationale for its acceptance of this risk:

> "This issue is still under investigation."