Zellic

Nocturne

Smart Contract Security Assessment

October 20, 2023

Prepared for:

Luke Tchang

Nocturne

Prepared by:

Malte Leip, Kuilin Li, and Mohit Sharma

Zellic Inc.

Contents

About Zellic 3

1 Executive Summary 4

1.1 Goals of the Assessment 4

1.2 Non-goals and Limitations 4

1.3 Results 4

2 Introduction 6

2.1 About Nocturne 6

2.2 Methodology 6

2.3 Scope 7

2.4 Project Overview 9

2.5 Project Timeline 9

3 Detailed Findings 10

3.1 Attacker-deployed ERC-20s cause reentrancy and unverified deposits 10

3.2 Arbitrage opportunities bypass deposit limits 12

3.3 Bundler calls can be identified by MEV bots and front-run 14

3.4 `Operation` with zero joinsplits can be tampered with 17

3.5 Note encryption is unconstrained 18

4 Discussion 19

4.1 Gas tokens can be identified by index in `Operation` 19

4.2 Schnorr signature documentation typo 20

4.3 The `StealthAddrOwnership` implements redundant order-l check 20

5 Threat Model **22**

5.1 Module: BalanceManager.sol 22

5.2 Module: CommitmentTreeManager.sol 23

5.3 Module: DepositManager.sol 25

5.4 Module: Handler.sol 29

5.5 Module: Teller.sol 31

5.6 Analysis of the JoinSplit circuit 32

5.7 Analysis of the util circuits 34

6 Assessment Results **35**

6.1 Disclaimer 35

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for Nocturne from October 2nd to October 20th, 2023. During this engagement, Zellic reviewed Nocturne's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is there any way funds entrusted to the protocol can be lost?
- Can an on-chain attacker bypass the deposit limits set in DepositManager?
- Could a malicious operation trigger a denial of service (i.e., bricking the protocol)?
- Do the privacy guarantees, such as an external observer not being able to deanonymize users, hold?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody
- Storage of ERC-721 or ERC-1155 assets — only ERC-20 assets are enabled as of the audit

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3 Results

During our assessment on the scoped Nocturne contracts, we discovered five findings. No critical issues were found. Two findings were of high impact, two were of low impact, and the remaining finding was informational in nature.

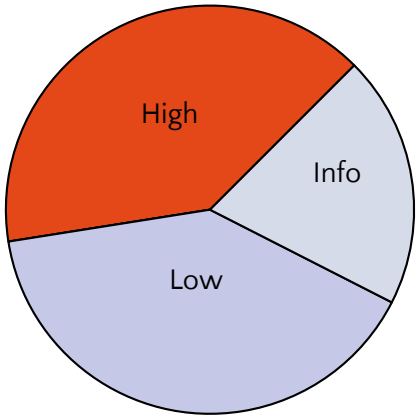
During the audit remediation phase, Nocturne requested Zellic to assess PR27, merged

in commit c9a4d4f6. This alteration allows users to perform an atomic swap and transfer (e.g., sending DAI by swapping from ETH). No security vulnerabilities were detected.

Additionally, Zellic recorded its notes and observations from the assessment for Nocturne's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
Medium	0
Low	2
Informational	1



2 Introduction

2.1 About Nocturne

Nocturne is protocol for private accounts on Ethereum. It combines a gas-optimized shielded pool with a stealth-address scheme to enable users to deposit, receive payments, and arbitrarily transact out of the Nocturne contracts, all with built-in asset privacy.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general.

We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3 Scope

The engagement involved a review of the following targets:

Nocturne Contracts

Repository	https://github.com/nocturne-xyz/protocol
Version	protocol: `bfcbfcfcb8cf1a2209c874700a43ecac28e18fe7`

Programs	bitifyBE.circom canonAddrSigCheck.circom joinsplit.circom lib.circom scalarMulWitGen.circom subtreeupdate.circom tree.circom AssetUtils BalanceManager CanonAddrRegistryEntryEIP712 CanonAddrSigCheckVerifier CanonicalAddressRegistry CommitmentTreeManager DepositManager DepositRequestEIP712 EthTransferAdapter Groth16 Handler JoinSplitVerifier NocturneReentrancyGuard OffchainMerkleTree OperationEIP712 OperationUtils Pairing ProxyAdmin Queue RethAdapter SubtreeUpdateVerifier Teller TransparentUpgradeableProxy TreeUtils Types Utils Validation Versioned WstethAdapter
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of three and a half person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Malte Leip, Engineer
malte@zellic.io

Kuilin Li, Engineer
kuilin@zellic.io

Mohit Sharma, Engineer
mohit@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

October 2, 2023	Kick-off call
October 2, 2023	Start of primary review period
October 20, 2023	End of primary review period

3 Detailed Findings

3.1 Attacker-deployed ERC-20s cause reentrancy and unverified deposits

- **Target**: Handler
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

The deposit-limit feature limits the rate at which ERC-20 tokens can enter and be mixed by Nocturne. Another feature, the screener-signature requirement, allows Nocturne to identify the owner of funds entering the protocol in order to ensure legal compliance.

One way that funds can enter Nocturne without being subject to the deposit limit or the screener-signature requirement is in the form of refund notes. At any time, the owner of previously deposited notes can issue an operation that unwraps the notes, does a series of actions (whitelisted external calls) with them, and then any assets resulting from those calls are rewrapped into new notes. The example use case for this feature is to enable Uniswap swaps of hidden assets.

Nocturne takes many steps to prevent these actions from introducing new value into the protocol or introducing any reentrancy hazards, including ensuring that the tracked assets have zero balances before running the actions, requiring the top-level bundle submission call to be made from an EOA, and strictly whitelisting the calls an action can be. In the version of the configuration we audited, the only swap methods whitelisted were Uniswap's `swapExactInput` and `swapExactInputSingle`.

However, a check that was missed is the tokens specified in the Uniswap swap path, including the `tokenIn` and `path` parameters. If the `tokenIn` parameter or any token in the `path` parameter is an attacker-deployed ERC-20, Uniswap will call `ERC20.transfer` on that token, which means the attacker can execute arbitrary code in the middle of the action.

Impact

An attacker can cause arbitrary calls to be done in the middle of an action through an attacker-deployed ERC-20 token's `ERC20.transfer` function called by Uniswap during a swap.

These arbitrary calls can transfer funds into the Handler, which bypasses deposit limits and screener checks; reenter Nocturne functions not gated by a reentrancy guard; and execute attacks on other protocols in order to immediately deposit the proceeds from such exploitation into Nocturne.

Recommendations

The Handler must ensure that all tokens Uniswap calls `transfer` on are legitimate tokens, tokens that do not cause attacker-specified behavior when called.

For `exactInputSingle`, this means checking the `tokenIn` and `tokenOut` parameters, and for `exactInput`, this means deserializing the `path` parameter and checking each token in it.

This issue is difficult to remediate because many tokens would need to be whitelisted for the purpose of being on a Uniswap path. (This could be a separate, more lax whitelist than the whitelist of tokens that Nocturne is willing to store.) If the best execution price for a swap that a nonmalicious user wishes to execute has a path that contains a token that is not on the whitelist, that user will have to get a suboptimal execution price for the swap.

Remediation

This issue has been acknowledged by Nocturne, and a fix was implemented in commits 50fe52a9 and 84f712da.

3.2 Arbitrage opportunities bypass deposit limits

- **Target**: Handler
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: High

Description

See Finding 3.1 for a description of the security guarantees around the external calls an action in an operation can make.

One logical consequence of allowing actions to execute swaps is that they can turn a profit by finding arbitrage opportunities between cycles of Uniswap pools. This is normally alright, but attackers can create larger-than-usual arbitrage opportunities by spending money outside Nocturne. If they do that and then resolve that arbitrage opportunity inside the protocol using an action, they have effectively made a deposit that bypasses the deposit limits and the screener-signature requirement.

Impact

If an attacker works with an Ethereum block builder, they can create an arbitrage opportunity immediately before the bundle gets processed by intentionally imbalancing a chosen cycle of Uniswap pools.

For example, if they choose tokens A, B, and C, they can use the A/B pool to trade A for B, and then use the B/C and C/A pools together to trade B for A. The former pool will have an inflated quantity of A and a scarcity of B, and the latter pair of pools will have an inflated quantity of B and a scarcity of A. The process can be repeated until all the funds have been spent on imbalancing the pool (or, a sufficiently large flash loan can be taken out so that all the funds the attacker wishes to "deposit" are spent imbalancing the pool in one or a few cycles — this saves gas).

Then, after the arbitrage opportunity is set up outside Nocturne, they execute a swap inside Nocturne rebalancing that cycle and extracting most of the funds they spent on imbalancing the pool, minus fees. Those funds are then added as refund notes, bypassing deposit limits and the screener-signature requirement.

An attacker must work with a block builder to execute this type of deposit because otherwise there is a significant risk of losing the funds to an arbitrage bot.

Recommendations

Safely check the total value of the assets before and after an action that does a swap, and reject the swap as unsafe if the increase in total value exceeds a threshold. If this

check is done on-chain (and bundle submission is still permissionless), care must be taken so that the oracle cannot also be manipulated.

Remediation

This issue has been acknowledged by Nocturne.

3.3 Bundler calls can be identified by MEV bots and front-run

- **Target**: Teller
- **Category**: Protocol Risks
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

Since being a bundler is permissionless, anyone can call `Teller.processBundle` to submit any valid bundle of operations. When submitting a bundle, the bundler pays for the gas spent in both verifying the proofs and executing the actions via Ethereum transaction fees. During the transaction, it then gets reimbursed for that gas via a transfer of unwrapped assets earmarked for gas.

However, this presents a perverse economic incentive for MEV-aware Ethereum block builders. Before including a `processBundle` transaction from a benign bundler in an Ethereum block, if a block builder simulates the transaction, they will find that if they front-run the transaction with an identical transaction sent from their own address instead, the transaction will happen in the same way, except they pay the gas cost and then they are paid the gas refund instead of the bundler. Doing this would cause the real bundler's transaction to revert, but the real bundler still pays the gas for verifying the proofs.

In `processBundle`,

```
function processBundle(
    Bundle calldata bundle
)
    external
    override
    whenNotPaused
    nonReentrant
    onlyEoa
    returns (uint256[] memory opDigests, OperationResult[]
    memory opResults)
{
    Operation[] calldata ops = bundle.operations;

// --- snip ---

    (bool success, uint256 perJoinSplitVerifyGas)
    = _verifyAllProofsMetered(
```

```
        ops,
        opDigests
    );
    require(success, "Batch JoinSplit verify failed");

    uint256 numOps = ops.length;
    opResults = new OperationResult[](numOps);
    for (uint256 i = 0; i < numOps; i++) {
        try
            _handler.handleOperation(
                ops[i],
                perJoinSplitVerifyGas,
                msg.sender
            )
        returns (OperationResult memory result) {
// --- snip ---
```

Note that first, a call to `_verifyAllProofsMetered` occurs, which expensively verifies the proofs and measures the gas required, setting `perJoinSplitVerifyGas`. Next, the call to `handleOperation` calls `_processJoinSplitsReservingFee`, which checks the nullifiers. This is what reverts in a second call, because the nullifiers will already have been used.

This means that, from a MEV-seeking block builder's perspective, if they front-run the bundler's transaction, they will still be paid for the gas price of verifying the proof. They need to pay it in their transaction, but the real bundler's reverted transaction will repay them about the same amount. So, they profit if they execute this front-run, and the real bundler is not repaid for the gas they spend on the proof verification.

Impact

Block builders are perversely incentivized to front-run the submission of bundles by bundlers. In a perfect economy, this means all bundlers must work with block builders or else their transactions will be reverted, front-run by the block builder issuing the same transaction, and they will pay for the gas for the verification circuit without any reimbursement. This disincentivizes block builders from building blocks.

Recommendations

Check the nullifiers of the joinsplits before checking the proofs so that a repeat submission of the same `Operation` fails much more cheaply, rendering the front-running of bundle submissions economically unviable.

Remediation

This issue has been acknowledged by Nocturne. Nocturne will ensure that bundlers submit their transactions through Flashbots Protect, which protects against front-running.

3.4 `Operation` with zero joinsplits can be tampered with

- **Target**: Handler
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

When an operation is processed in a transaction submitted by a bundler, it can specify an arbitrary sequence of external calls to do. These calls are checked by calculating the digest of the `Operation` struct and then supplying that digest as a public input into the joinsplit circuits. However, if an operation has zero joinsplits, no joinsplit circuits are verified, and so a bundler can freely change the calls executed.

Impact

There is not much impact, because if an operation has no joinsplits, no assets are unwrapped, and so the external calls only have access to the assets present in the contract before the operation (in the typical case, no assets).

Additionally, if an operation has no joinsplits, there is no way to repay the bundler for gas, so a bundler is disincentivized from including it in the bundle in the first place.

However, a user can still submit such an operation, and if they do, the bundler can modify it at will.

Recommendations

Disallow operations with no joinsplits.

Remediation

This issue has been acknowledged by Nocturne, and a fix was implemented in commit 50fe52a9.

3.5 Note encryption is unconstrained

- **Target**: joinsplit.circom
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Low
- **Impact**: Informational

Description

The note ciphertexts included as part of the joinsplit operation are unconstrained inputs into the circuit. This means that the note decrypted by the receiver may not correspond properly to the note added to the commitment tree.

Impact

The encrypted note ciphertext is responsible for communicating the value of the note to the receiver. Since in order to spend a note, the owner needs to know the value in the note, having a note where the real value in the commitment is unknown to the owner can make the note unspendable, causing the funds to become locked.

However, incorrectly computed ciphertexts can be identified by the receiver immediately upon the completion of the transaction by computing a commitment from the decrypted note and checking it against the Merkle tree entry.

This reduces the impact as the receiver can simply refuse to acknowledge transactions with malformed note ciphertexts and locked sender funds, eliminating the incentive for malformed ciphertexts as an attack.

Recommendations

We recommend client-side or on-chain checks for receivers to be able to verify note ciphertexts against note commitments.

Remediation

This issue has been acknowledged by Nocturne.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Gas tokens can be identified by index in `Operation`

In an `Operation`, the token used to pay back the bundler for gas is identified as an `EncodedAsset encodedGasAsset`, which must match one of the `trackedAssets` because it will be unwrapped after the actions run:

```
struct Operation {
    PublicJoinSplit[] pubJoinSplits;
    JoinSplit[] confJoinSplits;
    CompressedStealthAddress refundAddr;
    TrackedAsset[] trackedAssets;
    Action[] actions;
    EncodedAsset encodedGasAsset;
    uint256 gasAssetRefundThreshold;
    uint256 executionGasLimit;
    uint256 gasPrice;
    uint256 deadline;
    bool atomicActions;
}
```

In a `PublicJoinSplit`, the asset that the joinsplit points at is referred to using an index `uint8 assetIndex`:

```
struct PublicJoinSplit {
    JoinSplit joinSplit;
    uint8 assetIndex; // Index in op.joinSplitAssets
    uint256 publicSpend;
}
```

Instead of having to decode the encoded gas asset and compare that against each tracked asset, it would simplify the `Operation` struct and save gas to just specify the index of the gas asset.

4.2 Schnorr signature documentation typo

In the provided documentation, the steps to implement Nocturne's variant of Schnorr signatures over the Baby Jubjub curve are described as follows:

> 1. $h \leftarrow \text{SHA512}(sk)$
> 2. $s \leftarrow h[0:32]$ (derive the signing secret key from the spending key)
> 3. $x \leftarrow h[32:64]$ (extract 32 bytes of entropy from the spending key)
> 4. $v \leftarrow \text{randomBytes}(32)$ (sample another 32 random bytes)
> 5. $n \leftarrow \text{SHA512}(x \mid\mid v \mid\mid m) \mod r \in \mathbb{F}_r$ ("reduce" hash output into $\mathbb{F}_r$, using extra entropy from sk addition to the rng)
> 6. $R \leftarrow n \times G$ (the rest is a "standard" Schnorr signature)
> 7. $c \leftarrow H(\text{PK.X} \mid\mid R.X \mid\mid R.Y \mid\mid m)$
> 8. $z \leftarrow n - \text{sk} \cdot c$
> 9. The signature is the pair (c, z)

In step 8, z in the signature is generated using sk; however, it should be $z \leftarrow n - s \cdot c$ generated using s because the spending key sk is only used to derive the signing secret key.

4.3 The `StealthAddrOwnership` implements redundant order-l check

In lib.circom, `StealthAddrOwnership` computes `GG = vk * H1 - H2` and then applies the following constraints on `GG`:

```
(GG2X, GG2Y) <== BabyDbl()(GGX, GGY);
(GG4X, GG4Y) <== BabyDbl()(GG2X, GG2Y);
(GG8X, GG8Y) <== BabyDbl()(GG4X, GG4Y);

GG8X === 0;
GG8Y === 1;
```

But, since the input points to `StealthAddrOwnership`, `H1` and `H2` are already range

checked to be in the order-l subgroup of the curve, these constraints are redundant and can be removed.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Module: BalanceManager.sol

Function: `_processJoinSplitsReservingFee(Operation op, uint256 perJoinSplitVerifyGas)`

This is called by `handleOperation` to unwrap assets for the operation (nullifying unwrapped assets), transferring assets minus the amount reserved for the gas to the Handler (this).

Inputs

- `op`
  - **Control**: Arbitrary — proof must be valid. Tracked assets are whitelisted.
  - **Constraints**: None.
  - **Impact**: Operation to run.
- `perJoinSplitVerifyGas`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Amount to reserve (withhold from the Handler) to pay for gas.

Branches and code coverage

Intended branches

- Assets unwrap successfully.
  ☑ Test coverage

Negative behavior

- Tree root not past tree root.
  ☑ Negative test

- Note to be unwrapped already used (nullifier set).
  ☑ Negative test
- Not enough assets to cover gas.
  ☑ Negative test

5.2 Module: CommitmentTreeManager.sol

Function: `applySubtreeUpdate(uint256 newRoot, uint256[8] proof)`

This updates the root of the offline Merkle tree given a valid proof that the new tree is the old tree with the expected notes inserted.

Inputs

- `newRoot`
  - **Control**: Arbitrary.
  - **Constraints**: Must pass `subtreeupdate` circuit. Must not be past root hash.
  - **Impact**: New root.
- `proof`
  - **Control**: Arbitrary.
  - **Constraints**: Must pass `subtreeupdate` circuit.
  - **Impact**: None.

Branches and code coverage

Intended branches

- Subtree update succeeds.
  ☑ Test coverage

Negative behavior

- Subtree update fails due `newRoot` being a past root.
  ☐ Negative test
- Subtree update fails due to circuit-verification failure.
  ☐ Negative test

Function: `_handleJoinSplits(Operation op)`

This checks that a joinsplit is valid and nullifies assets unwrapped for it.

Inputs

- `op`
  - **Control**: Arbitrary — proof must be valid.
  - **Constraints**: See negative behavior.
  - **Impact**: Operation to check the joinsplits of.

Branches and code coverage

Intended branches

- Operation's joinsplits are valid.
  ☑ Test coverage

Negative behavior

- Commitment tree root is not a past root.
  ☑ Negative test
- Nullifier already used.
  ☑ Negative test
- Notes to join are the same note with same nullifier.
  ☑ Negative test

Function: `_handleRefundNote(EncodedAsset encodedAsset, CompressedStealthAddress refundAddr, uint256 value)`

This creates an encoded note out and adds refund note to queue to be added to off-chain Merkle tree by subtree update.

Inputs

- `encodedAsset`
  - **Control**: Arbitrarily set by authorized deposit source calling `Teller.depositFunds`.
  - **Constraints**: After decoding, address must be on whitelist. ERC-20 transfer of this value later must succeed.
  - **Impact**: Stored in enqueued note. Emitted in the `RefundProcessed` event.
- `refundAddr`
  - **Control**: Arbitrarily set by authorized deposit source calling `Teller.depositFunds`.
  - **Constraints**: The `encodedAsset.encodedAssetAddr` must be a valid field element and not have bits set outside `ENCODED_ASSET_ADDR_MASK`. The `encodedAsset.encodedAssetId` must be a valid field element and less than `MAX_AS`

SET_ID. The refundAddr.h1 and refundAddr.h2 must be valid field elements if the X-sign bit is unset.
  - **Impact**: Stored in enqueued note. Emitted in the RefundProcessed event.
- value
  - **Control**: Arbitrarily set by authorized deposit source calling Teller.depositFunds.
  - **Constraints**: Must be less than MAX_NOTE_VALUE. ERC-20 transfer of this value later must succeed.
  - **Impact**: Stored in enqueued note. Emitted in the RefundProcessed event.

Branches and code coverage

Intended branches

- Inserts note into queue.
  - ☑ Test coverage

Negative behavior

- Note fails validation.
  - ☑ Negative test (validation unit test)

5.3 Module: DepositManager.sol

Function: `completeErc20Deposit(DepositRequest req, byte[] signature)`

This completes ERC-20 deposit given a valid signature.

Inputs

- req
  - **Control**: Arbitrarily set by caller.
  - **Constraints**: Digest must be in outstanding deposit requests. Must pass checks done in Teller.depositFunds (see section 5.5).
  - **Impact**: Calls Teller.depositFunds with data.
- signature
  - **Control**: Arbitrarily set by caller.
  - **Constraints**: Must be a valid signature made by screener of digest of req.
  - **Impact**: None.

Branches and code coverage

Intended branches

- Calls `Teller.depositFunds`.
  ☑ Test coverage
- Actual gas compensation made.
  ☑ Test coverage
- Remaining gas compensation made.
  ☑ Test coverage

Negative behavior

- Deposit limits exceeded.
  ☑ Negative test
- Signature of `req` digest is not a screener.
  ☑ Negative test
- Deposit digest is not outstanding deposit.
  ☑ Negative test
- The `depositFunds` reverts.
  ☑ Negative test

Function: `instantiateETHMultiDeposit(uint256[] values, CompressedStealthAddress depositAddr)`

This instantiates a series of ETH deposit requests.

Inputs

- `values`
  - **Control**: Arbitrary.
  - **Constraints**: Transfer of sum of values must succeed. Deposit-size maximum enforced.
  - **Impact**: Transfers values and issues deposit requests for values.
- `depositAddr`
  - **Control**: Arbitrary.
  - **Constraints**: None. (If not a valid address, complete will fail.)
  - **Impact**: Copied to deposit request.

Branches and code coverage

Intended branches

- Instantiates deposit.
  ☑ Test coverage

Negative behavior

- Deposit size exceeds maximum.
  ☐ Negative test
- Not enough ETH sent with call.
  ☑ Negative test
- Gas compensation split uneven.
  ☐ Negative test

Function: `instantiateErc20MultiDeposit(address token, uint256[] values, CompressedStealthAddress depositAddr)`

This instantiates a series of ERC-20 deposit requests.

Inputs

- `token`
  - **Control**: Arbitrary.
  - **Constraints**: Must be whitelisted asset.
  - **Impact**: Token to be transferred.
- `values`
  - **Control**: Arbitrary.
  - **Constraints**: Transfer of sum of values must succeed. Deposit-size maximum enforced.
  - **Impact**: Transfers values and issues deposit requests for values.
- `depositAddr`
  - **Control**: Arbitrary.
  - **Constraints**: None. (If not a valid address, complete will fail.)
  - **Impact**: Copied to deposit request.

Branches and code coverage

Intended branches

- Instantiates deposit.
  ☑ Test coverage

Negative behavior

- Deposit size exceeds maximum.

☑ Negative test

- Gas compensation split is uneven.
  ☐ Negative test
- Transfer of token fails.
  ☐ Negative test

Function: `retrieveDeposit(DepositRequest req)`

This cancels and retrieves pending ERC-20 deposits.

Inputs

- `req`
  - **Control**: Arbitrarily set by caller.
  - **Constraints**: Digest must be in outstanding deposit requests.
  - **Impact**: Outstanding deposit request cleared and sends back tokens.

Branches and code coverage

Intended branches

- Successfully retrieves deposit.
  ☑ Test coverage

Negative behavior

- Caller is not spender.
  ☑ Negative test
- Deposit was not in outstanding deposit requests.
  ☑ Negative test

Function: `retrieveETHDeposit(DepositRequest req)`

This cancels and retrieves pending ETH deposit.

Inputs

- `req`
  - **Control**: Arbitrarily set by caller.
  - **Constraints**: Digest must be in outstanding deposit requests.
  - **Impact**: Outstanding deposit request cleared and sends back ETH.

Branches and code coverage

Intended branches

- Successfully retrieves ETH deposit.
  - ☑ Test coverage

Negative behavior

- Caller is not spender.
  - ☑ Negative test
- Deposit type was not ETH.
  - ☐ Negative test
- Deposit was not in outstanding deposit requests.
  - ☐ Negative test

5.4 Module: Handler.sol

Function: `handleDeposit(Deposit deposit)`

This verifies a deposit note and adds it to the Merkle tree. Called only by Teller.

Inputs

- `deposit`
  - **Control**: Arbitrarily set by any deposit source calling into `Teller.depositFunds`.
  - **Constraints**: The `deposit.encodedAsset` must be a supported contract. Note must pass checks (see `CommitmentTreeManager._handleRefundNote` in section 5.2 for more info).
  - **Impact**: Note details added to Merkle tree.

Branches and code coverage

Intended branches

- Call into `CommitmentTreeManager._handleRefundNote`.
  - ☑ Test coverage

Negative behavior

- Contract is paused.
  - ☑ Negative test
- Caller is not Teller.

☑ Negative test

- Asset is not whitelisted.
  ☑ Negative test
- The call to `CommitmentTreeManager._handleRefundNote` reverts.
  ☑ Negative test

Function: `handleOperation(Operation op, uint256 perJoinSplitVerifyGas, address bundler)`

This is called by Teller to handle one `Operation`. Ensures zeroed balances on all of the tracked assets of the operation, unwraps (transfers from Teller) assets, executes actions, pays bundler gas-compensation tokens, and creates refund tokens for tracked assets obtained during the actions.

Inputs

- `op`
  - **Control**: Arbitrary — proof must be valid.
  - **Constraints**: All assets must be tracked.
  - **Impact**: Operation to run.
- `perJoinSplitVerifyGas`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Reserves this amount to pay for gas from unwrapped assets.
- `bundler`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Address of bundler for refund.

Branches and code coverage

Intended branches

- Actions run successfully.
  ☑ Test coverage
- Balance not zeroed; actions run successfully after transfer.
  ☐ Test coverage
- An Action reverts during execution.
  ☑ Test coverage

Negative behavior

- Contract paused.
  ☑ Negative test
- Not called by Teller.
  ☑ Negative test
- Tracked assets contains unsupported asset.
  ☐ Negative test

5.5 Module: Teller.sol

Function: `depositFunds(Deposit deposit)`

This is called by an authorized deposit source to deposit funds. Inserts the refund note into the Merkle tree if valid.

Inputs

- `deposit`
  - **Control**: Arbitrarily set by caller.
  - **Constraints**: None checked by this contract (see `Handler.handleDeposit` in section 5.4 for more details).
  - **Impact**: Calls `Handler.handleDeposit` with `deposit` and transfers `deposit.encodedAsset` (assumed to be an ERC-20 token) from sender to Teller.

Branches and code coverage

Intended branches

- Call into `Handler.handleDeposit`.
  ☑ Test coverage

Negative behavior

- Contract is paused.
  ☑ Negative test
- Caller is not deposit source.
  ☑ Negative test
- `handleDeposit` reverts.
  ☑ Negative test

Function: `processBundle(Bundle bundle)`

This processes a bundle of operations.

Inputs

- `bundle`
  - **Control**: Arbitrary.
  - **Constraints**: All operations must be valid and pass circuit checks.
  - **Impact**: Actions in operations are executed, and refund notes are inserted into the Merkle tree. Gas is compensated back to caller.

Branches and code coverage

Intended branches

- Bundle with valid operations succeeds.
  ☑ Test coverage

Negative behavior

- Operation fails to be verified by circuit.
  ☐ Negative test
- Operation reverts during action execution.
  ☐ Negative test

5.6 Analysis of the JoinSplit circuit

JoinSplit circuit inputs

- `operationDigest`
- `pubEncodedAssetId`
- `pubEncodedAssetAddrWithSignBits`
- `refundAddrH1CompressedY`
- `refundAddrH2CompressedY`
- `vk` (viewing key)
- `spendPubkey`
- `vkNonce`
- `c`, `z` (operation signature)
- `encodedAssetId`
- `encodedAssetAddr`
- `refundAddrH1X`
- `refundAddrH1Y`
- `refundAddrH2X`
- `refundAddrH2Y`

- `oldNoteAOwnerH1X`
- `oldNoteAOwnerH1Y`
- `oldNoteAOwnerH2X`
- `oldNoteAOwnerH2Y`
- `oldNoteANonce`
- `oldNoteAValue`
- `pathA`
- `siblingsA`
- `oldNoteBOwnerH1X`
- `oldNoteBOwnerH1Y`
- `oldNoteBOwnerH2X`
- `oldNoteBOwnerH2Y`
- `oldNoteBNonce`
- `oldNoteBValue`
- `pathB`
- `siblingsB`
- `newNoteAValue`
- `receiverCanonAddr`
- `newNoteBValue`

Circuit outputs

- `newNoteACommitment`
- `newNoteBCommitment`
- `commitmentTreeRoot`
- `publicSpend`
- `nullifierA`
- `nullifierB`
- `senderCommitment`
- `joinSplitInfoCommitment`

Constraints

- The `spendPubkey` is a valid Baby Jubjub curve point of order l.
- The `vk` is derived correctly from `spendPubkey` and `vkNonce`.
- `oldNoteA.owner.H1` and `oldNoteA.owner.H2` are valid BabyJubJub curve points, and H1 is an order-l point.
- The `oldNoteB.owner.H1` and `oldNoteB.owner.H2` are valid BabyJubJub curve

points, and H1 is an order-l point.

- Constrain that H2 = [vk]H1 for `oldNoteA` and `oldNoteB`.
- Range check note values to account for arithmetic overflows.
- Compute and constrain public spend.
- Constrain note commitments for both old notes.
- Check Merkle inclusion proof for `oldNoteA`.
- Check Merkle inclusion proof for `oldNoteB` if and only if it holds nonzero value.
- Constrain nullifier derivation for `nullifierA` and `nullifierB`.
- Constrain new note commitments.

5.7 Analysis of the util circuits

`StealthAddrOwnership`

- Compute and constrain `G = vk*H1 - H2`.
- Constrain that `8*G == 0`.

`VKDerivation`

- Constrain that `vk = Poseidon(spendPubKey, vkNonce)`.
- Constrain that `vkbits` is the bit decomposition of `vk`.
- constrain that `vk` is less than the Baby Jubjub scalar field order.

`IsOrderL`

- Compute `Q = inv8*P` where `inv8 = (inv(8) mod l)`.
- Constrain that `8*Q` $\equiv$ `P`. This guarantees that `ord(P) = ord(Q)/GCD(ord(Q), 8)` so `ord(P)` is either l or 1.
- Constrain that P is not the identity point.

6 Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Nocturne contracts, we discovered five findings. No critical issues were found. Two findings were of high impact, two were of low impact, and the remaining finding was informational in nature. Nocturne acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.