Zellic

April 12, 2024

Prepared for
Jay Xu
Singularity

Prepared by
SeungHyeon Kim
Malte Leip
Sylvain Pelissier
Mohit Sharma
Zellic

Singularity

Zero Knowledge Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Singularity from March 4th to April 9th. During this engagement, Zellic reviewed Singularity's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. 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:

- Can funds be stolen from the asset pools?
- Is it ensured that on-chain changes reflect the user's intention for actions signed by the user?

1.3. 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

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

Lack of access to documentation for the external KeyringCredentials, PolicyManager, and WalletCheck contracts for the Keyring on-chain KYC provider integration prevented us from verifying correct usage of them in the KeyringManager contract.

Based on the number of severe findings uncovered during the audit, it is our opinion that the project is not yet ready for production. We strongly advise a comprehensive reassessment before deployment to help identify any potential issues or vulnerabilities introduced by necessary fixes or changes. We also recommend adopting a security-focused development workflow, including (but not limited to) augmenting the repository with comprehensive end-to-end tests that achieve 100% branch coverage using any common, maintainable testing framework, thoroughly documenting all function requirements, and training developers to have a security mindset while writing code.

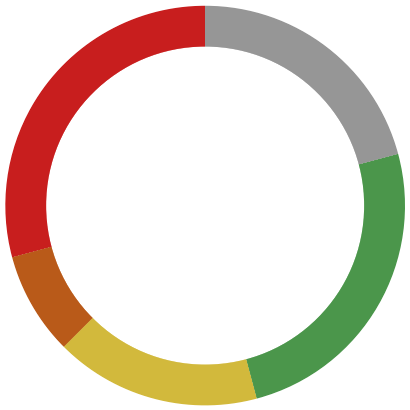
1.4. Results

During our assessment on the scoped Singularity contracts, we discovered 24 findings. Seven critical issues were found. Two were of high impact, four were of medium impact, six were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	7
🟧 High	2
🟨 Medium	4
🟩 Low	6
⬜ Informational	5



2. Introduction

2.1. About Singularity

Singularity contributed the following description of Singularity:

> Singularity is KYB/KYC permissioned institutional DeFi access layer that provides access to popular protocols for onchain participants to transact with commercial confidentiality.

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, upgradability 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:

Singularity Contracts

Repositories	https://github.com/portalgateme/darkpool-v1-zk-contracts ↗ https://github.com/portalgateme/darkpool-v1-proof ↗
Versions	darkpool-v1-zk-contracts: `b9f2c713310c7a85ad50c9de78bfeee86b77a0a4` darkpool-v1-proof: `608fa72d416ffa7fb4b4be15f24834937c203862`
Programs	• portalgateme-darkpool-v1-proof/src/utils/*.ts • portalgateme-darkpool-v1-proof/src/services/*.ts • portalgateme-darkpool-v1-zk-contracts/contracts/core/*.sol • portalgateme-darkpool-v1-zk-contracts/contracts/defi/*.sol • portalgateme-darkpool-v1-zk-contracts/circuits/*.nr
Type	Solidity, Noir, TypeScript
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with four consultants for a total of nine person-weeks. The assessment was conducted over the course of six 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:

SeungHyeon Kim
Engineer
seunghyeon@zellic.io ↗

Malte Leip
Engineer
malte@zellic.io ↗

Sylvain Pelissier
Engineer
sylvain@zellic.io ↗

Mohit Sharma
Engineer
mohit@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 4, 2024	Start of primary review period
March 7, 2024	Kick-off call
April 9, 2024	End of primary review period

3. Detailed Findings

3.1. Double spend possible within actions taking two notes as input

Target	DarkpoolAssetManager		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The DarkpoolAssetManager's `joinSplit`, `join`, and `swap` functions each invalidate two input notes by marking the respective nullifiers as used, preventing them from being spent a second time later. However, it is not ensured that the two nullifiers that are invalidated are distinct. Take the example of the `join` function:

```
function join(
    bytes32 _merkleRoot,
    bytes32 _nullifierIn1,
    bytes32 _nullifierIn2,
    bytes32 _noteOut1,
    bytes calldata _proof
) public payable {
    if (!_merkleTreeOperator.noteIsNotCreated(_noteOut1)) {
        revert NoteAlreadyCreated();
    }
    if (!_merkleTreeOperator.nullifierIsNotUsed(_nullifierIn1)) {
        revert NullifierUsed();
    }
    if (!_merkleTreeOperator.nullifierIsNotUsed(_nullifierIn2)) {
        revert NullifierUsed();
    }
    if (!_merkleTreeOperator.merkleRootIsAllowed(_merkleRoot)) {
        revert MerkleRootNotAllowed();
    }

    JoinRawInputs memory inputs = JoinRawInputs(
        _merkleRoot,
        _nullifierIn1,
        _nullifierIn2,
        _noteOut1
    );

    _verifyProof(_proof, _buildJoinInputs(inputs), "join");
    _postDeposit(_noteOut1);
```

```
        _postWithdraw(_nullifierIn1);
        _postWithdraw(_nullifierIn2);

        emit Join(_nullifierIn1, _nullifierIn2, _noteOut1);
    }
```

As can be seen in the snippet above, it is checked at the start of the function that `_nullifierIn1` and `_nullifierIn2` are not yet marked as used. At the end of the function, both are marked as used. The call to `_postWithdraw` will succeed even if the nullifier was already marked used, as it is a wrapper around the following function, which is idempotent:

```
function setNullifierUsed(bytes32 nullifier) external onlyAssetManager {
    if (nullifier != bytes32(0)) {
        nullifiersUsed[nullifier] = true;
    }
}
```

The `join` function will thus accept the same value for `_nullifierIn1` and `_nullifierIn2`, as long as that common nullifier has not been used yet. The situation is the same for the other two functions.

The respective `join_split`, `join`, and `swap` circuits do not ensure that the two input nullifiers are distinct either.

The Uniswap integration's `uniswapLiquidityProvision` also has two inputs. However, in this case, liquidity provision will ultimately revert when two identical inputs are used, as the assets will be the same (the NonfungiblePositionManager contract will indirectly call the PoolAddress libraries' `computeAddress` function, which requires the address of the first address to be smaller than the address of the second address; see here ↗).

The Curve integration's `curveAddLiquidity` functions in the CurveAddLiquidityAssetManager, CurveMPAddLiquidityAssetManager and CurveFSNAddLiquidityAssetManager contracts similarly have four input notes, but they are not checked to be distinct. They are all checked to be unused at the start of the function and only marked as used at the end of the function:

```
function curveAddLiquidity(
    bytes calldata _proof,
    AddLiquidityArgs calldata _args
) external payable {
    // ...
    for (uint256 i = 0; i < 4; i++) {
        _validateNullifierIsNotUsed(_args.nullifiers[i]);
    }

    // ...

    for (uint256 i = 0; i < 4; i++) {
        _postWithdraw(_args.nullifiers[i]);
    }
    // ...
}
```

Thus, from the perspective of the nullifier check, reuse of the same note in multiple inputs is possible. Doing so would, as in the Uniswap case, imply that multiple inputs are for the same assets. Whether there is any impact thus depends on whether there exist Curve pools with duplicate coins for tokens that the attacker is interested in double spending. We did not verify conclusively whether or not that is the case.

Impact

It is possible to use any one of the `joinSplit`, `join`, and `swap` functions to double one's balance by using the same note for both inputs. For example, for the `join` function, using a note with amount A for both inputs will allow one to obtain a joined note with amount $2 \cdot A$. Repeating this allows reaching arbitrary amounts, which can be followed by withdrawing to drain the darkpool's entire balance.

Recommendations

Ensure that the input notes / nullifiers must be distinct. This could be done by verifying the nullifiers are distinct in the circuit or alternatively by verifying this in the contract, for example by modifying the function `setNullifierUsed` of the MerkleTreeOperator contract to revert whenever the nullifier was already marked used:

```
function setNullifierUsed(bytes32 nullifier) external onlyAssetManager {
     require(nullifierIsNotUsed(nullifier), "Nullifier already used");
    if (nullifier != bytes32(0)) {
        nullifiersUsed[nullifier] = true;
    }
}
```

Remediation

This issue has been acknowledged by Singularity. The issue was fixed by the following commit 0228ebfc ↗. With the modifications, the `setNoteCommitmentCreated`, `setNullifierUsed` and `setNoteFooterUsed` functions revert in case the argument is already marked as used. Commit 0c5a5f28 ↗ also adds an in-circuit check that the two input notes to the join circuit are distinct.

3.2. Reentrancy in withdrawals leading to double-spend

Target	DarkpoolAssetManager, UniswapSwapAssetManager, UniswapLiquidityAssetManager		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Entry points that result in transfer of funds to a user or relayer do not follow the checks-effects-interactions pattern, transferring those funds to the relayer and/or user before marking the relevant nullifier as used. This results in possible double-spends.

There is some commonality of this finding with Finding 3.7. ↗. This finding discusses the case in which the attacker is the recipient of a transfer and can reenter on reception, and it results in double spending of the same asset that was used for reentry. Entry points are thus withdrawals for a normal user and any actions resulting in relayer fees for relayers. In contrast, Finding 3.7. ↗ discusses the case in which the attacker controls the contract of a fully fake token and can reenter on any call to that token (including transfers where the attacker is not the recipient). As the fake token is worthless, double-spending it does not gain anything, so as the entry point for the attack, the attacker needs to use actions in which two different assets are paid as input in order to double-spend the second, legitimate one while reentering on the first.

Case of DarkpoolAssetManager's `withdrawETH`

In the case of `withdrawETH`, the function is implemented as follows:

```
function withdrawETH(
    bytes calldata _proof,
    bytes32 _merkleRoot,
    bytes32 _nullifier,
    address payable _recipient,
    address payable _relayer,
    uint256 _relayerGasFee,
    uint256 _amount
) public {
    require(
        _complianceManager.isPermitted(address(this), _recipient),
        "BaseAssetManager:  invalid credential"
    );
    if (!_relayerHub.isRelayerRegistered(_relayer)) {
        revert RelayerNotRegistered();
    }
    if (!_merkleTreeOperator.nullifierIsNotUsed(_nullifier)) {
        revert NullifierUsed();
    }
    if (!_merkleTreeOperator.merkleRootIsAllowed(_merkleRoot)) {
        revert MerkleRootNotAllowed();
    }
    WithdrawRawInputs memory inputs = WithdrawRawInputs(
        _recipient,
        _merkleRoot,
        ETH_ADDRESS,
        _amount,
        _nullifier
    );

    _verifyProof(_proof, _buildWithdrawInputs(inputs), "withdraw");
    _releaseETHWithFee(_recipient, _relayer, _relayerGasFee, _amount);
    _postWithdraw(_nullifier);

    emit Withdraw(_nullifier, _amount, ETH_ADDRESS, _recipient);
}
```

An attacker can call this to withdraw a valid unspent note containing ETH. The funds will be transferred in the call `_releaseETHWithFee(_recipient, _relayer, _relayerGasFee, _amount);`, but only afterwards will the nullifier be marked as used with the call to `_postWithdraw(_nullifier);`, and thus the note becomes unusable. The function `_releaseETHWithFee` in the contract BaseAssetManager uses the ETH asset pool to transfer ETH to the relayer, fee manager, and the recipient specified as an argument to `withdrawETH`. Ultimately, the funds will be transferred using this function of the ETHAssetPool contract:

```
function release(
    address payable to,
    uint256 amount
) external onlyAssetManger nonReentrant transferNotLocked {
    require(amount > 0, "ETHAssetPool: amount must be greater than 0");

    require(
        address(this).balance >= amount,
        "ETHAssetPool: Insufficient balance"
    );

    (bool success, ) = to.call{value: amount}("");
    require(success, "ETHAssetPool: Failed to send Ether");
}
```

Control of execution is passed to the recipient's contract on `to.call{value: amount}("")`. Note that while the above function uses the `nonReentrant` modifier, this only prevents reentry into the `release` function of the ETHAssetPool contract. Reentry into functions of the DarkpoolAssetManager contract are still possible.

An attacker could reenter using, for example, the `transfer` function:

```
function transfer(
    bytes32 _merkleRoot,
    bytes32 _nullifierIn,
    bytes32 _noteOut,
    bytes calldata _proof
) public {
    if (!_merkleTreeOperator.noteIsNotCreated(_noteOut)) {
        revert NoteAlreadyCreated();
    }
    if (!_merkleTreeOperator.nullifierIsNotUsed(_nullifierIn)) {
        revert NullifierUsed();
    }
    if (!_merkleTreeOperator.merkleRootIsAllowed(_merkleRoot)) {
        revert MerkleRootNotAllowed();
    }

    TransferRawInputs memory inputs = TransferRawInputs(
        _merkleRoot,
        _nullifierIn,
        _noteOut
    );

    _verifyProof(_proof, _buildTransferInputs(inputs), "transfer");
    _postWithdraw(_nullifierIn);
    _postDeposit(_noteOut);

    emit Transfer(_nullifierIn, _noteOut);
}
```

All checks here would pass when using the same note as for the withdrawal. At the end, `_postWithdraw(_nullifierIn)` marks the spent note's nullifier as used, and `_postDeposit(_noteOut)` adds a new note holding the same amount of ETH that the spent note did.

When returning back from the call in the withdrawal, `withdrawETH` will call `_postWithdraw(_nullifier)` as well. However, as discussed in Finding 3.1. ↗, this is a idempotent operation, so marking the nullifier as used a second time does not fail.

At the end, the attacker would be left still having their original amount of ETH in a note in the darkpool but also having withdrawn that amount. The new note in the darkpool could still be withdrawn, thereby doubling the attacker's initial amount.

Case of DarkpoolAssetManager's `withdrawERC20`

In the case of `withdrawERC20`, the asset contract's `transfer` function is called to transfer the funds to the user. For normal ERC-20 tokens, these should not lead to reentrancy. However, reentrancy could occur for tokens that support receive hooks, such as ERC-777 tokens. As there are no restriction on which addresses can be used for ERC-20 token assets in the darkpool, such tokens could potentially be used.

While the attacker could also deploy their own (fake) token contract with receive hooks, this does not lead to an issue with respect to `withdrawERC20`, as double-spending a darkpool note denominated in their fake token does not advantage the attacker, as they could just mint their token directly anyway. Using a different entry point leveraging fake tokens is possible in a different manner, however; see Finding 3.7. ↗.

In the case of `withdrawERC20` with receive hooks, the attacker can proceed as in `withdrawETH` to drain the darkpool.

Case of DarkpoolAssetManager's `depositERC20`

In the case of `depositERC20`, the asset contract's `transferFrom` function is called to transfer the tokens from the user. ERC-777 also allows for send hooks, so for these tokens, the possibility of reentrancy exists for `depositERC20` as well. However, as no notes are spent in `depositERC20`, this can not lead to a double-spend vulnerability. The user could use this reentrancy to deposit the same amount twice and receive the same note for it (making one copy unspendable), which is otherwise not possible, but as this means the user loses half of their deposit, there is no advantage in doing so.

Case of relayer reentering

Whenever fees in are paid to the relayer, the relayer can reenter for ETH or ERC-20 with transfer hooks. They can thus carry out a similar attack to the one discussed in the `withdrawETH` case: the relayer acts as both relayer and user (using their own notes) to carry out an action causing fees to be paid to the relayer and to set the fee amount to the maximum possible. When receiving the fee, they reenter and transfer the input note to a fresh note so that they will retain the funds in the darkpool as well as having them paid out as relayer fees.

This type of issue occurs in all Darkpool and Uniswap entry points in which relayers get fees:

- DarkpoolAssetManager's `withdrawETH` and `withdrawERC20`
- UniswapSwapAssetManager's `uniswapSimpleSwap`
- UniswapLiquidityAssetManager's `uniswapLiquidityProvision`, `uniswapCollectFees`, and `uniswapRemoveLiquidity`

The issue does not appear to arise with the Curve integrations, as those mark nullfiers are used before transferring fees, as in this example of the CurveSingleExchangeAssetManager contract's `curveSingleExchange` functions:

```
function curveSingleExchange(
    bytes calldata _proof,
    ExchangeArgs calldata _args
) external payable {
    _validateRelayerIsRegistered(_args.relayer);
    _validateNullifierIsNotUsed(_args.nullifier);
    _validateMerkleRootIsAllowed(_args.merkleRoot);
    _validateNoteFooterIsNotUsed(_args.noteFooter);

    // ...

    if (_args.assetIn == ETH_ADDRESS) {
        _assetPoolETH.release(payable(address(this)), _args.amountIn);
        amountOut = IExchange(exchangeContract).exchange{value: msg.value}(
            _args.pool,
            _args.assetIn,
            _args.assetOut,
            _args.amountIn,
            amountOut
        );
    } else {
        // ...
    }

    _postWithdraw(_args.nullifier);

    // ...

    if (_args.assetOut == ETH_ADDRESS) {
        (bool success, ) = payable(address(_assetPoolETH)).call{
            value: noteAmount
        }("");
        (success, ) = payable(address(_feeManager)).call{value: serviceFee}(
            ""
        );
        (success, ) = payable(address(_args.relayer)).call{
            value: _args.gasRefund
        }("");
    } else {
        // ...
    }

    // ...
}
```

However, this function also does not follow the checks-effects-interactions pattern; to follow best practices, the nullifier should be marked used *before* releasing funds from the asset pool and calling `exchange`.

Impact

All ETH stored in the darkpool can be drained by the attacker. For ERC-20 assets, this is possible for those assets where the attacker can gain control of execution on `transfer` calls to the asset contract, such as in the case of ERC-777 receive hooks.

In the case of some entry points, the attack has to be carried out by a relayer.

Recommendations

Use the checks-effects-interactions pattern and mark the nullifier as used before interacting with external contracts, for example in the `withdrawETH` case:

```
function withdrawETH(
    bytes calldata _proof,
    bytes32 _merkleRoot,
    bytes32 _nullifier,
    address payable _recipient,
    address payable _relayer,
    uint256 _relayerGasFee,
    uint256 _amount
) public {
    require(
        _complianceManager.isPermitted(address(this), _recipient),
        "BaseAssetManager: invalid credential"
    );
    if (!_relayerHub.isRelayerRegistered(_relayer)) {
        revert RelayerNotRegistered();
    }
    if (!_merkleTreeOperator.nullifierIsNotUsed(_nullifier)) {
        revert NullifierUsed();
    }
    if (!_merkleTreeOperator.merkleRootIsAllowed(_merkleRoot)) {
        revert MerkleRootNotAllowed();
    }
    WithdrawRawInputs memory inputs = WithdrawRawInputs(
        _recipient,
        _merkleRoot,
        ETH_ADDRESS,
        _amount,
        _nullifier
    );
    _verifyProof(_proof, _buildWithdrawInputs(inputs), "withdraw");
    _releaseETHWithFee(_recipient, _relayer, _relayerGasFee, _amount);
    _postWithdraw(_nullifier);
    _releaseETHWithFee(_recipient, _relayer, _relayerGasFee, _amount);

    emit Withdraw(_nullifier, _amount, ETH_ADDRESS, _recipient);
}
```

For in-depth defense, we also recommend changing the `setNullifierUsed` function of the MerkleTreeOperator contract to revert whenever the nullifier was already marked used. This would prevent using the same nullifier twice in all cases. With this change, but without the change above, the attacker could still reenter, and the reentered call would succeed and mark the nullifier as used, but then in the original, outer call marking the nullifier as used would fail, causing a revert.

```
function setNullifierUsed(bytes32 nullifier) external onlyAssetManager {
    require(nullifierIsNotUsed(nullifier), "Nullifier already used");
    if (nullifier != bytes32(0)) {
        nullifiersUsed[nullifier] = true;
    }
}
```

Remediation

This issue has been acknowledged by Singularity. The issue was fixed by the following commit `0228ebfc` ↗ in which `setNullifierUsed` was changed to revert in case the argument is already marked as used.

3.3. Uniswap liquidity positions can get stolen

Target	UniswapLiquidityAssetManager		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Funds held by users in the darkpool can be in the form of ETH, ERC-20 tokens, or ERC-712 tokens. For bookkeeping of these funds, notes are used that consist of three fields:

1. The `asset` field, which is used for the address of the token contract, or a special value in the case of ETH.
2. The `amount` field, which stores the amount of ETH or ERC-20 token in those cases, or the NFT token ID in the case of ERC-712 tokens.
3. The `footer` field, which stores a hash of a random value `rho` and the public key. The usage of this field is not relevant for this finding.

These notes are stored on chain in a single Merkle tree. As the ETH/ERC-20 and ERC-712 notes are not domain separated, this opens the possibility of using actions intended for one note type for a note of the other type.

Concretely, this leads to the possibility of the following kind of attack in which an attacker, A, can steal a liquidity position from another user, B.

1. B provides liquidity to some Uniswap pool via the UniswapLiquidityAssetManager. They get minted a position NFT with a certain token ID, say token ID `b`, and a position note in the darkpool reflecting the NFT. Note that while the identity of B is protected, the token ID as well as the pool interacted with is visible on chain.
2. The attacker A also provides liquidity to a Uniswap pool while using the minimum amount of funds possible (thus, this step costs the attacker nearly nothing), receiving a position NFT with token ID `a` and a corresponding position note. In practice, on minting of new position NFTs, the NonfungiblePositionManager increments the IDs. So it will hold that `a > b`.
3. A now calls `split`, using their position note as input. The `split` action assumes a fungible type of note and interprets the second field as an amount. The attacker can thus use `split` to split the note into two, one where the `amount` field holds the value `b` and one in which it holds the value `b-a`.

4. The attacker now calls `uniswapRemoveLiquidity` with their position note with token ID `b`. They receive notes in return reflecting the liquidity that was originally provided by B. Should B try to remove the liquidity, they will get nothing. The attacker has thus successfully stolen B's liquidity position.

Impact

Anyone can steal Uniswap liquidity positions.

Recommendations

Ensure that non-fungible notes cannot be used as if they were fungible and vice-versa. This can be done by domain separating them. Currently, fungible note commitments are given by `hash(asset, amount, footer)` while non-fungible note commitments are given by `hash(asset, id, footer)`. These could be domain separated by using instead `hash(DOMAIN_SEPARATOR_FUNGIBLE, asset, amount, footer)` and `hash(DOMAIN_SEPARATOR_NON_FUNGIBLE, asset, id, footer)`, where `DOMAIN_SEPARATOR_FUNGIBLE` and `DOMAIN_SEPARATOR_NON_FUNGIBLE` are different constants. Proofs should then check that the new domain-separator field of notes is of the expected type. For example, the `split` circuit should fail for input notes that do not have `DOMAIN_SEPARATOR_FUNGIBLE` as the first field.

Remediation

This issue has been acknowledged by Singularity. The issue was fixed with commits `c2e68b4b` ↗, `b37a1eab` ↗, and `a8a9968d` ↗. The first commit adds domain separation functionality and the relevant checks to all relevant circuits except depoit and swap, the second commit adds the relevant check to the deposit circuit, and the third commit disables the swap entrypoint in the DarkpoolAssetManager.

3.4. Lack of wraparound protection in circuits

Target	join, join_split, and split circuits		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In the `join`, `join_split`, and `split` circuits, it is checked that the sum of the amounts of the input notes is equal to the sum of the amounts of the output notes, such as seen in this snippet from `join_split`:

```
assert(in_amount_1 + in_amount_2 == out_amount_1 + out_amount_2);
```

This check is done without protecting against overflows, so equality is only checked to hold modulo the characteristic `p` of the finite field that the witness variables are elements of. Thus, this check will pass, for example, for `in_amount_1 = 0`, `in_amount_2 = 0`, `out_amount_1 = a`, and `out_amount_2 = p - a`, for arbitrary `a`. An attacker can thus create themselves notes with arbitrary amounts.

This is possible with both the `join_split` and `split` circuits. In the case of the `join` circuit, overflows can only happen on the input side (as there is only one output note), hence a wraparound would be to the attacker's disadvantage (losing `p` of the asset).

Impact

An attacker can create notes with arbitrary amounts at no cost, thereby allowing them to drain the asset vaults of all ERC-20 and ETH.

Recommendations

Ensure in the circuits that no overflow happens on addition. As only two summands are added, one way to do this would be to check that each summand lies in the range `0, ..., (p-1) / 2`.

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `77df136c` ↗.

3.5. Front-runners of Uniswap swaps can steal most funds

Target	UniswapSwapAssetManager		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

With the `uniswapSimpleSwap` function, users can use Uniswap to swap one asset for another, paying with a note in the darkpool and obtaining a new note for the swapped asset. The function takes a proof as well as the following struct `UniswapSimpleSwapArgs` as arguments:

```
struct UniswapSimpleSwapArgs {
    // data of the note that will be used for swap
    UniswapNoteData inNoteData;
    // merkle root of the merkle tree that the commitment of the note is
    included
    bytes32 merkleRoot;
    // address of the asset that will be received after swap
    address assetOut;
    // address of the relayer
    address payable relayer;
    // minimum amount of the asset that will be received after swap
    uint256 amountOutMin;
    // note footer of the note created after swap
    bytes32 noteFooter;
    // gas fee of the relayer
    uint256 relayerGasFee;
    // pool fee of the swap (Uniswap)
    uint24 poolFee;
}

struct UniswapNoteData {
    address assetAddress;
    uint256 amount;
    bytes32 nullifier;
}
```

The proof checks that a note with the nullifier `inNoteDate.nullifier` is present in the Merkle tree with root hash `merkleRoot` with asset `inNoteDate.assetAddress` and amount `inNoteDate.amount`. It is also checked that `noteFooter` belongs to the same user and that the user signed the input note and nullifier as well as the output-note footer and asset.

However, `relayer`, `relayerGasFee`, `amountOutMin`, and `poolFee` are not used in the proof, and in particular, are not checked to be signed by the user owning the input note.

As the `uniswapSimpleSwap` function can be called by anyone, an attacker getting knowledge of the arguments to a call to `uniswapSimpleSwap` (for example by observing a transaction including it in the mempool or by acting as a relayer for the call) can then front-run the call with their own transaction, where they can choose the above arguments freely.

The impact of this regarding manipulation of `relayer` and `relayerGasFee` is discussed in a separate finding; see Finding 3.8. ↗.

An attacker can submit the transaction with `amountOutMin=0` and `poolFee` set to a value for which the Uniswap pool for the relevant asset pair has very low liquidity. By preceding the call to `uniswapSimpleSwap` with themselves adding the appropriate liquidity to the pool at the highest or lowest price possible, the swap carried out by `uniswapSimpleSwap` for the input note's owner will be done at an extremely unfavorable price. The attacker will thus have been able to buy the value of the input note (minus the service fee) for (next to) nothing, essentially stealing the input note's value.

Impact

Attackers front-running calls to `uniswapSimpleSwap` can steal the value of the input note.

Recommendations

Ensure that the user owning the input note approves of `relayer`, `relayerGasFee`, `amountOutMin`, and `poolFee` by including those values as public circuit variables in the `uniswap_swap` circuit and including them in the signature signed by the user's private key.

Remediation

This issue has been acknowledged by Singularity. In commit `c2e68b4b` ↗, `amountOutMin` and `poolFee` were added as public circuit variables in the `uniswap_swap` circuit and included in the signature signed by the user's private key. The analogous change for `relayer` was implemented in commit `b22be7df` ↗.

Singularity informed us that initially, only Singularity will run relayers, and that gas fees will be handled before third parties can run relayers.

3.6. Fund lock via dummy notes in curve_add_liquidity

Target	curve_add_liquidity circuit		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The CurveFSNAddLiquidityAssetManager contract's `curveAddLiquidity` entry point can be used to add liquidity to Curve pools. As part of its arguments, arrays of four assets and amounts as well as nullifiers are passed:

```
struct AddLiquidityArgs {
    bytes32 merkleRoot;
    bytes32[4] nullifiers;
    address[4] assets;
    uint256[4] amounts;
    address pool;
    address lpToken;
    bytes32 noteFooter;
    address payable relayer;
    uint256[4] gasRefund;
}
```

Not all of the four indexes are necessarily used, with the remaining amounting to dummy notes, which correspond to indexes `i` for which `assets[i]` and `amounts[i]` are zero. This finding concerns such dummy notes, so we will go through the code to discuss how the `nullifiers`, `assets`, and `amounts` arguments are used. Here is an extract from the implementation of the `curveAddLiquidity` function:

```
function curveAddLiquidity(
    bytes calldata _proof,
    AddLiquidityArgs calldata _args
) external payable {
    _validateRelayerIsRegistered(_args.relayer);
    for (uint256 i = 0; i < 4; i++) {
        _validateNullifierIsNotUsed(_args.nullifiers[i]);
    }

    // ...
    if (
        _args.amounts[0] +
            _args.amounts[1] +
            _args.amounts[2] +
            _args.amounts[3] ==
        0
    ) {
        revert AmountNotCorrect();
    }

    // ...

    _verifyProof(
        _proof,
        _buildLPInputs(
            LPRawInputs(
                _args.merkleRoot,
                _args.nullifiers,
                _args.assets,
                _args.amounts,
                _args.pool,
                _args.noteFooter
            )
        ),
        "curveAddLiquidity"
    );

    if (
        !_validateAssets(_getMetaFactoryCoins(_args.pool), _args.assets)
    ) {
        revert AssetNotInPool();
    }

    uint256[] memory actualAmounts;
    uint256[4] memory serviceFees;

    (actualAmounts, serviceFees, )
    = IFeeManager(_feeManager).calculateFeeForFSN(
        _args.amounts,
        _args.gasRefund
    );

    uint256 mintAmount = _addLiquidity(_args, actualAmounts);

    // ...
}
```

As can be seen here, all nullifiers are first checked to be unused so far. The check on the sum of `amounts` implies not all input notes can be dummy notes. The proof is then verified. In the proof circuit, a signature is checked that includes four input notes and nullifiers. Furthermore, for each of the four input notes, it is checked that they and the corresponding nullifier are correct, as long as the corresponding amount is nonzero. If the corresponding amount is zero, then verification is skipped:

```
if amount1 != 0 {
    fuzk::assert_note_with_membership(
        merkle_root,
        merkle_index1,
        merkle_path1,
        note1,
        asset1,
        amount1,
        rho1,
        pub_key
    );
    fuzk::assert_nullifier(nullifier1, rho1, pub_key);
}
```

This means that if for some `i` the argument `amounts[i]` is zero, then `nullifiers[i]` and `assets[i]` can be chosen arbitrarily by the caller in the proof.

The next check involving these three values in `curveAddLiquidity` is the call to `_validateAssets`, which is implemented as follows:

```
function _validateAssets(
    address[8] memory coins,
    address[4] memory assets
) internal pure returns (bool) {
    for (uint256 i = 0; i < 4; i++) {
        if (coins[i] == address(0)) {
            continue;
        }
        if (assets[i] != address(0)) {
            if (assets[i] != coins[i]) {
                if (
                    assets[i] == _ETH_ADDRESS && coins[i] == _WETH_ADDRESS
                ) {
                    continue;
                }
                return false;
            }
        }
    }
    return true;
}
```

As can be seen here, if `assets[i]` is zero, then this check passes, no matter what the value of `coins[i]` is. So far, we can thus conclude that the caller can choose `nullifier[i]` to have an arbitrary value that has not been marked used yet, as long as `assets[i]` and `amounts[i]` are both zero.

The `curveAddLiquidity` function will then ultimately call `_addLiquidity`, which is implemented as follows:

```
function _addLiquidity(
        AddLiquidityArgs memory _args,
        uint256[] memory actualAmounts
    ) private returns (uint256) {
        uint256 coinNum = _getMetaFactoryCoinNum(_args.pool);
        uint256 ethAmount = 0;
        uint256 mintAmount = 0;
        uint256 i = 0;
        if (coinNum <= 4) {

            // ...

            for (i = 0; i < 4; i++) {
                _postWithdraw(_args.nullifiers[i]);
            }
        } else {
            revert PoolNotSupported();
        }
        return mintAmount;
    }
```

Note that `_postWithdraw` is called for all nullifiers, including the dummy nullifiers corresponding to the cases in which `assets[i]` and `amounts[i]` are both zero and which the caller can choose arbitrarily (as long as they have not been used yet).

An attacker can thus call CurveFSNAddLiquidityAssetManager contract's `curveAddLiquidity` function with one to three legitimate inputs as well as at least one dummy note, for which the attacker can choose an arbitrary unused nullifier. That nullifier will then be marked used and thus cannot be used anymore afterwards.

The analogous issue exists for the CurveAddLiquidityAssetManager and CurveMPAddLiquidityAssetManager contracts.

Impact

An attacker that sees a pending transaction in the mempool can front-run it with a call to one of the `curveAddLiquidity` functions with a dummy note's nullifier set to the nullifier from the front-run transaction, thereby making the victim's note unspendable.

Recommendations

Consider ensuring that dummy notes have a zero nullifier. For example, if `amount1 == 0`, assert that `nullifier1 == 0` as well.

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `fe268599` ↗.

3.7. Reentrancy for actions involving multiple assets allows draining the vault

Target	UniswapLiquidityAssetManager, CurveAddLiquidityAssetManager, CurveFSNAddLiquidityAssetManager, CurveMPAddLiquidityAssetManager		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Several actions will cause calls to token contracts to move funds, for example from or to the asset pool. The implementations tend to not follow the checks-effects-interactions pattern, making external calls to the asset contracts before relevant nullifiers are marked as used. An attacker can thus double-spend tokens with the following generic steps:

1. The attacker creates a contract that supports relevant ERC-20 functions and deposits some of this token into the darkpool, obtaining a note denominated in the attacker's token.
2. The attacker calls an entry point in which they spend at least two notes. One of them is the note denominated in the attacker's token, and the rest are notes for legitimate assets. We assume that the attacker's contract will be called after the nullifiers for all notes were checked to be unused but before they are marked used.
3. On the call to the attacker's contract, the attacker reenters, spending the input notes for legitimate assets another time. As the nullifiers are not yet marked used, this will not revert.
4. Back in the original call, the original spend will not revert either, as the nullifiers are not checked a second time.

This issue occurs with the following entry points:

- UniswapLiquidityAssetManager's function `uniswapLiquidityProvision`
- CurveAddLiquidityAssetManager's function `curveAddLiquidity`
- CurveFSNAddLiquidityAssetManager's function `curveAddLiquidity`
- CurveMPAddLiquidityAssetManager's function `curveAddLiquidity`

We will explain the detailed steps of the attack with an example in the case of liquidity provision on Uniswap. The attack works very similarly in the case of the three liquidity provision entry points for Curve.

1. First, the attacker creates a contract with the relevant functionality for the later steps, in particular implementing the relevant parts of the ERC-20 interface. We will call this token FAKE.
2. The attacker then deploys a new Uniswap pool for FAKE and the target token they want to drain. Let us call that token REAL.
3. The attacker deposits 10 REAL and 10 FAKE into the darkpool.
4. The attacker calls `uniswapLiquidityProvision` to add liquidity to the REAL-FAKE pool, using the 10 FAKE and 10 REAL notes as input.

The following is an extract of that function's implementation:

```
function uniswapLiquidityProvision(
        UniswapLiquidityProvisionArgs memory args,
        bytes calldata proof
    ) public returns (FeesDetails memory feesDetails) {
        _validateLiquidityProvisionArgs(args);

        // ...

        (
            mintParams,
            feesDetails,
            originalIndices
        ) = _releaseFundsAndPrepareLiquidityProvisionArgs(args);

        // ...
    }
```

That the nullifiers of the two notes are not marked as used yet is only checked by `_validateLiquid-ityProvisionArgs`, which is called right at the start.

```
function _validateLiquidityProvisionArgs(
    UniswapLiquidityProvisionArgs memory args
) internal view {
    _validateMerkleRootIsAllowed(args.merkleRoot);
    _validateNullifierIsNotUsed(args.noteData1.nullifier);
    _validateNullifierIsNotUsed(args.noteData2.nullifier);
    // ...
}
```

The call to `_releaseFundsAndPrepareLiquidityProvisionArgs` will then release funds for each asset from the vault and then mark the relevant nullifier as used. As the attacker uses FAKE as the first of the two assets, when releasing FAKE funds from the vault (which calls the `transfer` function of the attacker's FAKE contract), none of the two nullifiers is marked used yet.

```
function _releaseFundsAndPrepareLiquidityProvisionArgs(
    UniswapLiquidityProvisionArgs memory args
)
    internal
    returns (
        INonfungiblePositionManager.MintParams memory mintParams,
        FeesDetails memory feesDetails,
        uint8[2] memory originalIndices
    )
{
    /**
     * * Release funds for token 1
     */

    FundReleaseDetails memory fundReleaseDetails1;

    fundReleaseDetails1.assetAddress = args.noteData1.assetAddress;
    fundReleaseDetails1.recipient = payable(address(this));
    fundReleaseDetails1.relayer = args.relayer;
    fundReleaseDetails1.relayerGasFee = args.relayerGasFees[0];
    fundReleaseDetails1.amount = args.noteData1.amount;

    (
        uint256 actualReleasedAmountToken1,
        FeesDetails memory feesDetails1
    ) = _releaseAndPackDetails(fundReleaseDetails1);

    _postWithdraw(bytes32(args.noteData1.nullifier));

    /**
     * * Release funds for token 2
     */

    // ...
}
```

5. In the call to the attacker's FAKE contract when releasing the 10 FAKE from the asset pool, the attacker calls the DarkpoolAssetManager's `transfer` function to transfer the 10 REAL note to a new note. The old 10 REAL note's nullifier will be marked used by `transfer`.

6. When returning to the original `uniswapLiquidityProvision` call, the 10 REAL-note nullifier is now marked used. However, execution is already past the check for this, so this does not cause an issue. The nullifier will get marked as used again by `_postWithdraw`. However, as discussed in Finding 3.1. ↗, this is a idempotent operation, so marking the nullifier as used a second time does not fail.

7. The attacker will thus be left with a liquidity position reflecting 10 REAL as well as the new 10 REAL note from the transfer. Removing the Uniswap liquidity position, they will be left with 20 REAL, having doubled their original investment. Repeating this process, the attacker can drain the REAL asset pool completely.

Impact

Attackers can completely drain ERC-20 and ETH asset pools.

Recommendations

Ensure all functions follow the checks-effects-interactions pattern and mark nullifiers as used before interacting with external contracts.

For in-depth defense, we also recommend changing the `setNullifierUsed` function of the MerkleTreeOperator contract to revert whenever the nullifier is already marked used. This would prevent using the same nullifier twice in all cases.

```
function setNullifierUsed(bytes32 nullifier) external onlyAssetManager {
    require(nullifierIsNotUsed(nullifier), "Nullifier already used");
    if (nullifier != bytes32(0)) {
        nullifiersUsed[nullifier] = true;
    }
}
```

Remediation

This issue has been acknowledged by Singularity. The issue was fixed in commit `0228ebfc` ↗ where `setNullifierUsed` was changed to revert in case the argument is already marked as used.

3.8. Relayers can steal funds by manipulating intended relayer and relayer fee

Target	DarkpoolAssetManager, All Uniswap integration contracts, All Curve integration contracts		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	Medium	**Impact**	High

Description

For many transactions, relayers are sent a fee for their gas costs and other costs. As an example, let us consider the function `withdrawERC20` of the DarkpoolAssetManager, which is implemented as follows:

```
function withdrawERC20(
        address _asset,
        bytes calldata _proof,
        bytes32 _merkleRoot,
        bytes32 _nullifier,
        address _recipient,
        address _relayer,
        uint256 _amount,
        uint256 _relayerGasFee
    ) public {
        require(
            _complianceManager.isPermitted(address(this), _recipient),
            "BaseAssetManager: invalid credential"
        );
        if (!_relayerHub.isRelayerRegistered(_relayer)) {
            revert RelayerNotRegistered();
        }
        if (!_merkleTreeOperator.nullifierIsNotUsed(_nullifier)) {
            revert NullifierUsed();
        }
        if (!_merkleTreeOperator.merkleRootIsAllowed(_merkleRoot)) {
            revert MerkleRootNotAllowed();
        }
        WithdrawRawInputs memory inputs = WithdrawRawInputs(
            _recipient,
            _merkleRoot,
            _asset,
            _amount,
            _nullifier
        );

        _verifyProof(_proof, _buildWithdrawInputs(inputs), "withdraw");

        _releaseERC20WithFee(
            _asset,
            _recipient,
            _relayer,
            _relayerGasFee,
            _amount
        );

        _postWithdraw(_nullifier);

        emit Withdraw(_nullifier, _amount, _asset, _recipient);
    }
```

Note that this function is public and `msg.sender` is not checked against anything, so this function is callable by anyone. We focus on the two arguments `_relayer` and `_relayerGasFee`. They only occur in the following two calls:

```
if (!_relayerHub.isRelayerRegistered(_relayer)) {
    revert RelayerNotRegistered();
}

_releaseERC20WithFee(
    _asset,
    _recipient,
    _relayer,
    _relayerGasFee,
    _amount
);
```

The first call checks that `_relayer` is a registered relayer. Relayers can be registered by the owner of the relayer hub.

The second call will transfer a certain service fee to the fee manager, transfer an amount of `_relay-erGasFee` of `asset` to `relayer`, and transfer the rest to `_recipient`.

As long as the service fee `serviceFee` plus `_relayerGasFee` is less than `_amount`, this will not revert. The argument `_relayerGasFee` can thus be freely chosen by the caller as long as it satisfies `_re-layerGasFee < _amount - serviceFee`. Similarly, `_relayer` can be freely chosen as long as the address is registered as a relayer.

Relayers are registered by the owner of the relayer hub. Logic deciding who will get registered is not in scope for this audit. Singularity informed us that the plan is to decentralize relayers, with anyone staking certain tokens having the possibility to become relayers. With that context, we judge that relayers cannot be considered trusted, though the staking aspect may introduce certain hurdles for potential attackers. Because of this we rate likelihood as Medium.

Impact

A user may expect a certain relayer A to submit the transaction for a certain small fee. Relayer A could instead submit the transaction with `_relayerGasFee` set to `_amount - serviceFee - 1`, pocketing (nearly) the entire withdrawal (minus the service fee), leaving the user with (next to) nothing.

Another relayer B might also see the transaction including a call to, for example, `withdrawERC20` in the mempool and front-run it with a call to `withdrawERC20`, in which they replace `_relayer` with their own address and `_relayerGasFee` with `_amount - serviceFee - 1`.

The following entry points are vulnerable to this issue:

- DarkpoolAssetManager's `withdrawETH` and `withdrawERC20`
- UniswapSwapAssetManager's `uniswapSimpleSwap`
- UniswapLiquidityAssetManager's `uniswapLiquidityProvision`, `uniswapCollectFees`, and `uniswapRemoveLiquidity`
- CurveAddLiquidityAssetManager's `curveAddLiquidity`
- CurveFSNAddLiquidityAssetManager's `curveAddLiquidity`
- CurveFSNRemoveLiquidityAssetManager's `curveRemoveLiquidity`
- CurveMPAddLiquidityAssetManager's `curveAddLiquidity`
- CurveMPRemoveLiquidityAssetManager's `curveRemoveLiquidity`
- CurveMultiExchangeAssetManager's `curveMultiExchange`
- CurveRemoveLiquidityAssetManager's `curveRemoveLiquidity`
- CurveSingleExchangeAssetManager's `curveSingleExchange`

Recommendations

Ensure that the user owning the funds from which the fee is paid approves of the relayer and relayer fee by including those two values as public circuit variables in the relevant circuit and including them in the signature signed by the user's private key.

Remediation

This issue has been acknowledged by Singularity. In commit `b22be7df` ↗, `relayer` was added as public circuit variables in the relevant circuits and included in the signature signed by the user's private key.

Singularity informed us that initially, only Singularity will run relayers, and that gas fees will be handled before third parties can run relayers.

3.9. Note footers not checked for Uniswap fee collection

Target	UniswapLiquidityAssetManager		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

Verification of arguments to the `uniswapCollectFees` function are handled by `_validateCollectFeesArgs`, which is implemented as follows.

```
function _validateCollectFeesArgs(
    UniswapCollectFeesArgs memory args
) internal view {
    _validateMerkleRootIsAllowed(args.merkleRoot);
    _validateNullifierIsNotUsed(args.feeNoteFooters[0]);
    _validateNullifierIsNotUsed(args.feeNoteFooters[1]);
    _validateRelayerIsRegistered(args.relayer);
}
```

Here the two note footers should be checked to not have been used before using `_validateNoteFooterIsNotUsed`. However, the incorrect function `_validateNullifierIsNotUsed` is used instead, making the check ineffective.

Impact

Note footers can be reused, making all but one of the resulting notes with the same nullifier unspendable.

As position notes are not spent when collecting fees, the missing check allows replaying calls to `uniswapCollectFees`, with the same proof. This means that an attacker could grief a position's fees after the first time fees were collected; they can periodically replay the call, thereby transferring the position's fees to notes with duplicate nullifiers that thus cannot all be spent.

On each replay, the relayer will still get their fee. Hence, the relayer would be incentivized to replay the call each time the Uniswap fees are high enough to cover the relayer fee, thereby stealing most of the Uniswap fees after the very first fee collection for themselves.

Relayers are registered by the owner of the relayer hub. Logic deciding who will get registered is not in scope for this audit. Singularity informed us that the plan is to decentralize relayers, with anyone staking certain tokens having the possibility to become relayers. With that context, we judge that relayers cannot be considered trusted, though the staking aspect may introduce certain hurdles for potential attackers. Because of this, we rate the likelihood as Medium.

Other impact depends on the front-end used.

Recommendations

Call the `_validateNoteFooterIsNotUsed` function instead:

```
function _validateCollectFeesArgs(
    UniswapCollectFeesArgs memory args
) internal view {
    _validateMerkleRootIsAllowed(args.merkleRoot);
    _validateNullifierIsNotUsed(args.feeNoteFooters[0]);
    _validateNullifierIsNotUsed(args.feeNoteFooters[1]);
    _validateNoteFooterIsNotUsed(args.feeNoteFooters[0]);
    _validateNoteFooterIsNotUsed(args.feeNoteFooters[1]);
    _validateRelayerIsRegistered(args.relayer);
}
```

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `f8e86c3e` ↗.

3.10. Output-note footers can be reused within the same action

Target	All manager contracts with multiple output notes		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

This issue is similar to Finding 3.1. ↗, but it is for note footers instead of for nullifiers. In actions where multiple note footers are involved for outputs, checking whether these note footers have already been used is done first, and only later are both marked used. This allows for the same note footer to be used multiple times in such actions. For example, the UniswapLiquidityAssetManager contract's `uniswapRemoveLiquidity` function looks as follows:

```
function uniswapRemoveLiquidity(
    UniswapRemoveLiquidityArgs memory args,
    bytes calldata proof
)
    public
    returns (
        TransferFundsToVaultWithFeesAndCreateNoteData memory dataToken1,
        TransferFundsToVaultWithFeesAndCreateNoteData memory dataToken2
    )
{
    _validateRemoveLiquidityArgs(args);

    // ...

    _registerNoteFooter(args.outNoteFooters[0]);
    _registerNoteFooter(args.outNoteFooters[1]);

    // ...
}
```

Here, `_validateRemoveLiquidityArgs` checks that `args.outNoteFooters[0]` and `args.outNoteFooters[1]` have not been marked as used yet. The `_registerNoteFooter` function is a wrapper around the following function in the MerkleTreeOperator contract:

```
function setNoteFooterUsed(bytes32 noteFooter) external onlyAssetManager {
    if (noteFooter != bytes32(0)) {
        noteFootersUsed[noteFooter] = true;
    }
}
```

This is idempotent, so calling it twice in a row for the same footer marks that footer as used but does not fail.

The situation is similar for other actions and in other contracts wherever more than one footer is checked.

An analogous issue arises when checking for the entire note already having appeared, for example in DarkpoolAssetManager's `split` function:

```
function split(
    bytes32 _merkleRoot,
    bytes32 _nullifierIn1,
    bytes32 _noteOut1,
    bytes32 _noteOut2,
    bytes calldata _proof
) public payable {
    if (!_merkleTreeOperator.noteIsNotCreated(_noteOut1)) {
        revert NoteAlreadyCreated();
    }
    if (!_merkleTreeOperator.noteIsNotCreated(_noteOut2)) {
        revert NoteAlreadyCreated();
    }

    // ...

    _postDeposit(_noteOut1);
    _postDeposit(_noteOut2);

    // ...
}
```

The `_postDeposit` function is implemented as follows:

```
function _postDeposit(bytes32 _noteCommitment) internal {
    _merkleTreeOperator.setNoteCommitmentCreated(_noteCommitment);
    _merkleTreeOperator.appendMerkleLeaf(bytes32(_noteCommitment));
}
```

Neither `setNoteCommitmentCreated` nor `appendMerkleLeaf` revert when called again with the same argument. Using the same note for `_noteOut1` and `_noteOut2` will thus cause that note to appear twice in the Merkle tree, with the note commitment marked as used.

Impact

Note footers can be reused,[1] making all but one of the resulting notes with the same nullifier unspendable. Precise impact depends on the front-end used. Singularity informed us that the intention for note-footer–reuse checks is to prevent user error, in particular copy-pasting the same footer twice and thereby accidentally making funds unusable. This type of user error appears likely to appear particularly within the same transaction. As user error is needed, we still rate likelihood as Low.

Recommendations

Ensure that the output footers must be distinct. This could be done by verifying the footers are distinct in the circuit or alternatively by verifying this in the contract, for example by modifying the function `setNoteFooterUsed` of the MerkleTreeOperator contract to revert whenever the note footer was already marked used:

```
function setNoteFooterUsed(bytes32 noteFooter) external onlyAssetManager {
     require(noteFooterIsNotUsed(noteFooter));
    if (noteFooter != bytes32(0)) {
        noteFootersUsed[noteFooter] = true;
    }
}
```

For those cases in which the footers are not public (such as in the DarkpoolAssetManager's `split` function) and the note commitment is checked instead, the analogous change should be done. When checking for reuse in the contract rather than the circuit, this could be done by changing `setNoteCommitmentCreated` of the MerkleTreeOperator contract to revert whenever the note commitment was already marked used:

```
function setNoteCommitmentCreated(
    bytes32 commitment
) external onlyAssetManager {
     require(noteIsNotCreated(commitment));
    noteCommitmentsCreated[commitment] = true;
}
```

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `0228ebfc` ↗.

[1] The case of reuse of note commitments implies reuse of note footers as well, which is why both variants are subsumed under the description of note-footer reuse.

3.11. Note footers can be reused in DarkpoolAssetManager functions

Target	DarkpoolAssetManager		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

To prevent generation of more than one note with the same nullifier, functions in contracts other than DarkpoolAssetManager check that the output-note footers have not been used yet and mark them as used. However, functions in DarkpoolAssetManager only check that the *note* has not been posted so far. It is thus possible to generate multiple notes with the same footer and hence, absent hash collisions, also the same nullifier, as long as the asset or amount differs.

Impact

Via the actions offered by the DarkpoolAssetManager, multiple notes with the same nullifier can be generated as outputs — also across actions, making all but one of the resulting notes unspendable. Precise impact depends on the front-end used. Singularity informed us that the intention for note-footer–reuse checks is to prevent user error, in particular copy-pasting the same footer twice and thereby accidentally making funds unusable. How likely such a user error is to happen with regards to footer reuse across multiple actions when generating proofs for DarkpoolAssetManager function depends on the front-end used and workflows that users follow.

Recommendations

If ensuring on chain that unspendable notes with the the same nullifier cannot be generated by the system is important, consider calculating and exposing the output-note footers in the circuits, and then verifying that they have not been marked used yet, and then marking them used. Note that this will make the output-note footers public where they were not before, so it should be considered whether there are usages in which this impacts privacy.

Remediation

This issue has been acknowledged by Singularity, and fixes were implemented in the following commits:

- `1bdfe703` ↗
- `a8a9968d` ↗

3.12. Front-runners of `uniswapLiquidityProvision` can cause loss of fees

Target	UniswapLiquidityAssetManager		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

With the `uniswapLiquidityProvision` function, users can use two assets they own on the darkpool in order to provide liquidity on a Uniswap pool. The advantage to a user for doing this is earning (parts of) the fees that users of Uniswap are paying to trade in the position's range. Uniswap pools can have different fees, from 0.01% to 1%.

The `uniswapLiquidityProvision` function takes a proof as well as the following struct `UniswapLiquidityProvisionArgs` as arguments:

```
struct UniswapLiquidityProvisionArgs {
    // data of the note that will be used for liquidity provision as first token
    UniswapNoteData noteData1;
    // data of the note that will be used for liquidity provision as second
    token
    UniswapNoteData noteData2;
    // address of the relayer
    address payable relayer;
    // gas fees of the relayer
    uint256[2] relayerGasFees;
    // merkle root of the merkle tree that the commitment of the note is
    included
    bytes32 merkleRoot;
    // note footer of the note that will be created after mint (both tokens)
    bytes32[2] changeNoteFooters;
    // tick min of the liquidity provision
    int24 tickMin;
    // tick max of the liquidity provision
    int24 tickMax;
    // note footer of the NFT position note that will be created after
    liquidity provision
    bytes32 positionNoteFooter;
    // pool fee of the liquidity provision (Uniswap)
    uint24 poolFee;
}
```

```
struct UniswapNoteData {
    address assetAddress;
    uint256 amount;
    bytes32 nullifier;
}
```

The proof checks that notes with the nullifiers `noteData1.nullifier` and `noteData2.nullifier` are present in the Merkle tree with root hash `merkleRoot`, with assets `noteData1.assetAddress` and `noteData2.assetAddress` and amounts `noteData1.amount` and `noteData2.amount` belonging to the same user. It is also checked that the three note footers are well-formed and belong to the same user and that the user signed the two input notes and nullifiers, the three output footers, and `tickMin` and `tickMax`.

However, `relayer`, `relayerGasFee`, and `poolFee` are not used in the proof, and in particular, are not checked to be signed by the user owning the input note.

As the `uniswapLiquidityProvision` function can be called by anyone, an attacker getting knowledge of the arguments to a call to `uniswapLiquidityProvision` (for example by observing a transaction including it in the mempool or by acting as a relayer for the call) can then front-run the call with their own transaction, where they can choose the above arguments freely.

The impact of this regarding manipulation of `relayer` and `relayerGasFee` is discussed in a separate finding; see Finding 3.8. ↗.

An attacker can submit the transaction with `poolFee` set to a lower value than in the original transaction. The user will then possibly receive a lower amount of fees than they expected (e.g., if the liquidity is not provided to a pool with 0.01% fees instead of 1%). Furthermore, a front-runner is incentivized to carry out this attack if they themselves wish to swap their asset in a direction and at a price that is made possible by this position. By submitting the `uniswapLiquidityProvision` call with a lower fee, they can within the same transaction swap against that position at a possibly much lower fee than otherwise.

Impact

Attackers front-running calls to `uniswapLiquidityProvision` can lower the Uniswap fee level for which liquidity will be provided, benefitting the attacker by being able to trade at lower fees and causing loss of potential fee profits to the original user.

Recommendations

Ensure that the user owning the input notes approves of `relayer`, `relayerGasFee`, and `poolFee` by including those values as public circuit variables in the `uniswap_lp` circuit and including them in the signature signed by the user's private key.

Remediation

This issue has been acknowledged by Singularity. In commit `b22be7df` ↗, `relayer` was added as public circuit variables in the `uniswap_lp` circuit and included in the signature signed by the user's private key. In commit `4df0140f` ↗, `poolFee` was analogously added.

Singularity informed us that initially, only Singularity will run relayers, and that gas fees will be handled before third parties can run relayers.

3.13. No slippage limit for Uniswap liquidity provision and removal

Target	UniswapLiquidityAssetManager		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In the `_releaseFundsAndPrepareLiquidityProvisionArgs` function, the parameters for the minimum minted amounts are hardcoded to zero:

```
/**
* * Prepare mint parameters
*/

mintParams.token0 = tokens[0];
mintParams.token1 = tokens[1];

mintParams.fee = args.poolFee;
mintParams.tickLower = args.tickMin;
mintParams.tickUpper = args.tickMax;
mintParams.amount0Desired = amounts[0];
mintParams.amount1Desired = amounts[1];
mintParams.amount0Min = 0;
mintParams.amount1Min = 0;
mintParams.recipient = payable(address(this));
mintParams.deadline = block.timestamp + 2 minutes;
```

The minting will succeed for any amount, even zero.

Similarly, `_prepareRemoveLiquidityParams` preparing liquidity removal hardcodes the minimum output amounts to zero:

```
function _prepareRemoveLiquidityParams(
    UniswapRemoveLiquidityArgs memory args,
    uint128 liquidity
)
    internal
    view
    returns (
        INonfungiblePositionManager.DecreaseLiquidityParams
```

```
            memory decreaseLiquidityParams
    )
{
    decreaseLiquidityParams.tokenId = args.positionNote.amount;
    decreaseLiquidityParams.liquidity = liquidity;
    decreaseLiquidityParams.amount0Min = 0;
    decreaseLiquidityParams.amount1Min = 0;
    decreaseLiquidityParams.deadline = block.timestamp;
}
```

Impact

An attacker could front-run a Uniswap liquidity provision with a transaction that manipulates the pool price. The liquidity provision would then occur at an incorrect price, resulting in a lower amount of liquidity for the provider than would have been the case at the correct price. The attacker might profit from this by being able to trade at an advantageous price immediately afterwards. For example, consider a pool containing tokens X and Y:

1. The attacker trades on the Uniswap pool to a new price for X higher than the current (not manipulated) price P_{real}. The price is the amount of tokens Y that the attacker has to pay to obtain a token X. As the attacker buys more and more, the price of X increases.
2. The front-run user's liquidity position from P_{low} to P_{high} is added, with $P_{\text{low}} < P_{\text{real}} < P_{\text{high}}$ but the current price above P_{high}. This liquidity position thus only consists of tokens Y, which are now cheaper than at the unmanipulated price. The liquidity provider obtains less liquidity for their tokens than expected.
3. The attacker now trades back to the original price. This involves trading back through the other liquidity provider's liquidity. Additionally, there is extra liquidity by the victim between P_{real} and P_{high}. This allows the attacker to sell additional tokens X at a price between P_{real} and P_{high}. They thus obtain more tokens Y for their tokens X than they would have at the unmanipulated price.

However, the attacker also has to pay fees for their trades, including for the liquidity between P_{real} and P_{high} by all the other providers. Thus, this attack is profitable only if the victim's position is wide enough (so that P_{high} differs significantly from P_{real}) as well as large enough compared to the other liquidity providers in that range. (Otherwise, the fees paid to trade through the other liquidity provider's position would exceed any possible profits from trading with the victim at a profitable price.)

For the case of liquidity removal, front-runners might cause an unexpected mix of the two tokens to be returned, but this should not lead to a loss of value for the user.

Recommendations

In order to use the `amount0Min` and `amount1Min` parameters to prevent slippage, we recommend setting those parameters to values provided by the user, which are exposed as public circuit variables in the relevant circuit and included in the data signed by the user.

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `4df0140f` ↗.

3.14. Relayers can drain Curve asset managers

Target	FeeManager		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

For several types of actions, fees are paid to a relayer and the fee manager (as a service fee). In most cases, the fees are calculated by the following function:

```
function calculateFee(
    uint256 amount,
    uint256 relayerRefund
) external view returns (uint256, uint256, uint256) {
    uint256 serviceFeePercent = _serviceFeePercent;
    uint256 serviceCharge = (amount * serviceFeePercent) / _PRECISION;

    require(
        amount > serviceCharge + relayerRefund,
        "FeeManager: amount must be greater than fees"
    );

    return (
        amount - serviceCharge - relayerRefund,
        serviceCharge,
        relayerRefund
    );
}
```

Note that the `serviceCharge` is calculated from `amount` (as a percentage), while `relayerRefund` is an absolute amount passed as an argument. The function checks that `amount` is sufficient to cover both fees.

In some instances, multiple assets are involved and the following function is used instead:

```
function calculateFee(
    uint256[4] calldata amount,
    uint256[4] calldata relayerRefund
)
```

```
    external
    view
    returns (uint256[4] memory, uint256[4] memory, uint256[4] memory)
{
    uint256 serviceFeePercent = _serviceFeePercent;
    uint256[4] memory serviceCharge;
    uint256[4] memory actualAmount;
    for (uint256 i = 0; i < 4; i++) {
        if (amount[i] != 0) {
            serviceCharge[i] = (amount[i] * serviceFeePercent) / _PRECISION;
            require(
                amount[i] > serviceCharge[i] + relayerRefund[i],
                "FeeManager:  amount must be greater than fees"
            );
            actualAmount[i] =
                amount[i] -
                serviceCharge[i] -
                relayerRefund[i];
        }
    }

    return (actualAmount, serviceCharge, relayerRefund);
}
```

While the `amount[i] > serviceCharge[i] + relayerRefund[i]` requirement is present in this variant of the function as well, it is skipped if `amount[i] == 0`. Thus, if `amount[i] = 0` and `relayerRefund[i]` is an arbitrary value, then calling this variant, `calculateFee`, will not revert.

A function `calculateFeeForFSN` also exists with identical behavior, having the same issue.

The two fee calculation functions with the above bug are used in several Curve integration asset managers.

As an example, we consider the `curveAddLiquidity` function of the CurveAddLiquidityAssetManager contract. That function receives an `AddLiquidityArgs` struct as one argument, containing a field `uint256[4] gasRefund`. The following extract of the implementations shows the only two places where `gasRefund` is used:

```
function curveAddLiquidity(
    bytes calldata _proof,
    AddLiquidityArgs calldata _args
) external payable {

    // ...

    uint256[4] memory actualAmounts;
    uint256[4] memory serviceFees;
    (actualAmounts, serviceFees, ) = IFeeManager(_feeManager).calculateFee(
        _args.amounts,
        _args.gasRefund
    );

    uint256 mintAmount = _addLiquidity(_args, actualAmounts);

    IERC20(_args.lpToken).safeTransfer(
        address(_assetPoolERC20),
        mintAmount
    );

    _transferFees(
        _args.assets,
        serviceFees,
        _args.gasRefund,
        address(_feeManager),
        _args.relayer
    );

    // ...
}
```

If a relayer calls `curveAddLiquidity` with arguments that satisfy the following,

1. The relayer is set as `_args.relayer`.
2. For some `i`, it holds that `_args.amounts[i] = 0`, and `_args.gasRefund[i] = a` for some arbitrary `a`.
3. The arguments are otherwise valid.

then the call will not revert, and the relayer will receive `a` of `_args.assets[i]` as fees, as long as the asset manager held a sufficient amount of these tokens. Relayers can thus use this to steal all tokens that are held by the asset manager. In normal operation, the asset manager should not hold any tokens at rest however, so practical impact is limited to any tokens that may have been stuck by user error or other bugs.

Impact

Relayers can drain several Curve asset managers of any tokens or Ether they hold. As the asset managers usually do not hold any assets themselves, impact is limited unless attackers can combine this with another bug that leads to tokens being stuck in the asset manager (rather than being held by the relevant asset pool).

Recommendations

As future code changes that are seemingly unrelated to the FeeManager contract (e.g., making relayer fees paid directly by the asset pools rather than transferring first to the asset manager and then having the asset manager pay the relayer) could easily result in a critical vulnerability where relayers can drain asset pools of all ERC-20 tokens and ETH, we recommend to make sure that in `calculateFee`, the `amount[i] > serviceCharge[i] + relayerRefund[i]` check is carried out in all cases, including when `amount[i]` is zero. This can be done, for example, with the following change:

```
function calculateFee(
    uint256[4] calldata amount,
    uint256[4] calldata relayerRefund
)
    external
    view
    returns (uint256[4] memory, uint256[4] memory, uint256[4] memory)
{
    uint256 serviceFeePercent = _serviceFeePercent;
    uint256[4] memory serviceCharge;
    uint256[4] memory actualAmount;
    for (uint256 i = 0; i < 4; i++) {
        if (amount[i] != 0) {
            serviceCharge[i] = (amount[i] * serviceFeePercent) / _PRECISION;
            require(
                amount[i] > serviceCharge[i] + relayerRefund[i],
                "FeeManager: amount must be greater than fees"
            );
            actualAmount[i] =
                amount[i] -
                serviceCharge[i] -
                relayerRefund[i];
        }
    }

    return (actualAmount, serviceCharge, relayerRefund);
}
```

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit f9784a0c ↗.

3.15. Both transfer and approval done by `_transferERC20`, thus sending twice the intended amount

Target	UniswapCoreAssetManager		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The UniswapCoreAssetManager contract's function `_transferERC20` both approves and transfers the amount:

```
function _transferERC20(
    address token,
    address to,
    uint256 amount
) internal {
    if (amount > 0) {
        IERC20(token).forceApprove(to, amount);
        IERC20(token).safeTransfer(to, amount);
    }
}
```

This means that the receiver will obtain both the funds transferred directly as well as an allowance that they can use to later transfer themselves the same amount again. As long as there are not other approvals overriding the earlier one in the meantime and the asset manager has a sufficient balance, the receiver will thus be able to obtain twice the intended amount.

The function `_transferERC20` is only called by `_transferAsset` in the same contract. That function is in turn called by `_transferAssetToVault` and `_chargeFees`. The former of the two only transfers to the vault, which is trusted and whose current code does make use of the approvals; hence, impact is limited. The latter transfers to the relayer and fee manager. The function is called by `_transferFundsToVaultWithFeesAndCreateNote`, which is in turn called by `uniswapCollectFees` and `uniswapRemoveLiquidity`.

Impact

Relayers for calls to `uniswapCollectFees` and `uniswapRemoveLiquidity` obtain up to twice the intended amount of fees. However, any amount beyond the intended fee must come from tokens stuck in the respective Uniswap asset-manager contracts, where no tokens should stay at rest under normal conditions.

Recommendations

Remove the approval from `_transferERC20`:

```
function _transferERC20(
    address token,
    address to,
    uint256 amount
) internal {
    if (amount > 0) {
        IERC20(token).forceApprove(to, amount);
        IERC20(token).safeTransfer(to, amount);
    }
}
```

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `fe268599` ↗.

3.16. Low-entropy note generation

Target	noteService.ts		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

If there are multiple notes for the same public key with identical `rho`, only one of those notes will be spendable. The function `generateRho` generates this value only with 64 bits of entropy according to its implementation:

```
export const generateRho = () => {
  let ab = new ArrayBuffer(32);
  return bn_to_0xhex(crypto.getRandomValues(Buffer.from(ab)).readBigUInt64BE()
    % p);
};
```

‘

Impact

If a note is generated by calling this function, as it is done by the `createNote` function, then the entropy for such value is not sufficient to avoid a situation similar to the Faerie Gold Attack ↗. Then, if a note is created with the same value of `rho` as a previous one, then it would not be usable even though it is valid.

Recommendations

The value of `rho` should be generated uniformly over the range $[0, p)$.

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit 0972c4dd ↗. The note is generated over 47 bytes and then reduced modulo p to avoid modular bias.

3.17. Pool parameters are not verified in Curve multiexchange

Target	CurveMultiExchangeAssetManager		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

In the function `curveMultiExchange`, the route and the swap parameters are verified within the route-hash computation and included in the proof verification. The pool argument defines the pool addresses, and the array is passed as an argument in `_args.pools`. This argument is not verified in the proof. This argument can be front-run and changed.

Impact

The pool address needs to be a valid pool address for the given token, but nevertheless, it may be changed for another pool the user did not choose. There is not a big margin for exploitation.

Recommendations

It can be added to the route-hash computation to enforce its verification.

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `1c9f23e1` ↗.

3.18. Signatures in circuits are not domain separated

Target	All circuits		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The circuits contain and verify a Schnorr signature of the relevant data (such as input and output notes) for the action, as in this snippet from the `join` circuit.

```
let m = std::hash::mimc::mimc_bn254([
    in_note_1,
    in_note_2,
    in_nullifier_1,
    in_nullifier_2,
    out_note_1,
]);
let m_bytes = fuzk::to_bytes(m);

let v = std::schnorr::verify_signature(
    pub_key[0],
    pub_key[1],
    signature,
    m_bytes,
);

assert(v);
```

The data that is signed does not however contain a domain separator that specifies *which* action is being signed. This means that an attacker (for example, a service constructing the relevant proof as a service to the user) that obtains one such signature might be able to use it to construct a proof for a different action than the user intended.

Doing so would require the length of the data being hashed to be equal between these two circuits. There are some clashes like this that could be used:

```
4:
split: [Field; 4] : [note, nullifier, out note, out note]
uniswap_swap: [Field; 4]: [note, nullifier, out footer, out asset]
transfer: [Field; 4]: [note, nullifier, out note, out footer]
```

```
5:
join: [Field; 5]: [note, note, nullifier, nullifier, out note]
curve_exchange: [Field; 5]: [note, nullifier, pool, out footer, out asset]
uniswap_collect_fees: [Field; 5]: [pos note, pos addr, pos token, note footer,
    note footer]
curve_multi_exchange: [Field; 5]: [note, nullifier, routeHash, out footer, out
    asset]

6:
join_split: [Field; 6]: [note, note, nullifier, nullifier, out note, out note]
swap: [Field; 6]: [note, note, nullifier, nullifier, out note, out note]
uniswap_remove_liquidity: [Field; 6]: [note, address, token, nullifier, out
    footer, out footer]
```

As the attacker would not be able to control the data being signed, attempting to prove the “wrong” circuit might mean to have to (say using a `curve_exchange` signature for `join`) prove that something is a well-formed note commitment, even though that piece of data was constructed as a nullifier of another note, or having to prove that something is a nullifier for a specific note even though it arose from a completely different piece of data. For many of the pairs above, it will thus be impossible in practice to use a signature for the wrong circuit. In some cases it might be possible, such as using a `join_split` signature for a `swap` proof. We have not found a way for an attacker to profit from any such proof for an incorrect circuit, however.

Note that, should Schnorr signatures of `mimc_bn254` hashed data be used elsewhere as well (i.e., not just in these circuits), then possible issues of using such signatures elsewhere, or using such other signatures within the circuit, could arise as well.

Impact

The signatures in the circuits are not domain separated, so it is not ensured that such a signature was intended by the signer to be used for this particular circuit. No practical impact was identified at this time, but we still recommend proper domain separation.

Recommendations

Add a domain separator to the data to be signed. This domain separator should be a different constant for each circuit, and it should be unlikely that the same domain separator is used accidentally for a different purpose.

For example, as the domain separator for the `join` circuit, one might use `DOMAIN_SEPARATOR_JOIN=mimc_bn254(“SINGULARITY_CIRCUIT_JOIN”)` (pseudocode, the string would need to be cast to an array of field elements) as a domain separator and add that constant as a prefix to the data to be signed:

```
let m = std::hash::mimc::mimc_bn254([
    DOMAIN_SEPARATOR_JOIN,
    in_note_1,
    in_note_2,
    in_nullifier_1,
    in_nullifier_2,
    out_note_1,
]);
let m_bytes = fuzk::to_bytes(m);

let v = std::schnorr::verify_signature(
    pub_key[0],
    pub_key[1],
    signature,
    m_bytes,
);

assert(v);
```

Remediation

This issue has been acknowledged by Singularity and a fix for all circuits except swap was implemented in commit `0094b4e1` ↗. The swap entrypoint was disabled in commit `a8a9968d` ↗.

3.19. Incomplete asset-validation logic

Target	CurveAssetManagerHelper		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `_validateAssets` function is defined in CurveAssetManagerHelper as follows.

```
function _validateAssets(
    address[8] memory coins,
    address[4] memory assets
) internal pure returns (bool) {
    for (uint256 i = 0; i < 4; i++) {
        if (coins[i] == address(0)) {
            continue;
        }
        if (assets[i] != address(0)) {
            if (assets[i] != coins[i]) {
                if (
                    assets[i] == _ETH_ADDRESS && coins[i] == _WETH_ADDRESS
                ) {
                    continue;
                }
                return false;
            }
        }
    }
    return true;
}
```

The following snippet makes the validation logic insufficient.

```
if (coins[i] == address(0)) {
    continue;
}
```

Consider a Curve pool with two tokens, poolToken1 and poolToken2. If a user initiates a request where the assets are [poolToken1, poolToken2, extraToken1, extraToken2], this will still pass the `_validateAssets` check.

Impact

One possible impact is a fund lock in `addLiquidity` functions. For example, in CurveAddLiquidityAssetManager, `_addLiquidity` contains the following snippets:

```
for (i = 0; i < 4; i++) {
    if (actualAmounts[i] > 0) {
        if (_args.assets[i] == ETH_ADDRESS) {
            if (ethAmount > 0) {
                revert AmountNotCorrect();
            }
            _assetPoolETH.release(
                payable(address(this)),
                _args.amounts[i]
            );
            ethAmount = actualAmounts[i];
            // 01 or 11
            if (_args.isLegacy & 1 == 1) {
                _wrapEth(_args.amounts[i]);
                IERC20(_WETH_ADDRESS).forceApprove(
                    _args.pool,
                    actualAmounts[i]
                );
            }
        } else {
            _assetPoolERC20.release(
                _args.assets[i],
                address(this),
                _args.amounts[i]
            );
            IERC20(_args.assets[i]).forceApprove(
                _args.pool,
                actualAmounts[i]
            );
        }
    }
}
```

```
for (i = 0; i < 4; i++) {
    _postWithdraw(_args.nullifiers[i]);
}
```

This pulls assets from the asset pools for all the provided assets. All the corresponding notes are also marked as spent.

But the processing only happens for the first `coinNum` tokens; the ownership of the remaining two tokens, extraToken1 and extraToken2, is therefore never transferred from the asset-manager contract and gets locked.

Recommendations

Modify the `_validateAssets` functions to handle extra tokens.

```
function _validateAssets(
    address[8] memory coins,
    address[4] memory assets
) internal pure returns (bool) {
    for (uint256 i = 0; i < 4; i++) {
        if (assets[i] != address(0)) {
            if (assets[i] != coins[i]) {
                if (
                    assets[i] == _ETH_ADDRESS && coins[i] == _WETH_ADDRESS
                ) {
                    continue;
                }
                return false;
            }
        }
    }
    return true;
}
```

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `55c3df63` ↗.

3.20. Second preimage attack in Merkle circuit

Target	fuzk/src/lib.nr		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The Merkle verification circuit is vulnerable to a second preimage attack. The circuit is defined as follows.

```
pub fn compute_merkle_root(
    leaf: Field,
    merkle_index: [u1; 32],
    merkle_path: [Field; 32],
) -> Field {

    let mut merkle_root = leaf;
    for i in 0..32 {
        let left = if merkle_index[i] == 0 { merkle_root } else {
    merkle_path[i] };
        let right = if merkle_index[i] == 1 { merkle_root } else {
    merkle_path[i] };

        let next_merkle_root = std::hash::mimc::mimc_bn254([left, right]);
        if merkle_path[i] != 0 {
            merkle_root = next_merkle_root;
        }
    }
    merkle_root
}
```

Impact

The circuit allows variable length paths, and there is no domain separation for node hashes and leaf hashes. Therefore, an attacker can successfully prove an intermediate note as a leaf in a Merkle tree.

We did not discover a way to exploit this in practice.

Recommendations

There are generally two ways to mitigate a second preimage attack in Merkle root computation.

1. Make the verification path a fixed length.
2. Add domain separation for hashing leaves and hashing internal nodes.

The first solution is not a possibility for Singularity as the on-chain Merkle store allows variable depth leaves. We would recommend adding domain separation as per the second solution.

A straightforward way to implement domain separation would be to add different prefix elements to leaf and node hashes.

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `064c573c` ↗.

3.21. The two Curve exchange proofs are interchangable

Target	curve_exchange and curve_multi_exchange circuits		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The CurveMultiExchangeAssetManager contract's `curveMultiExchange` function verifies a proof for the `curveMultiExchange` circuit, while the CurveSingleExchangeAssetManager contract's `curveSingleExchange` function verifies a proof for the `curveExchange` circuit.

However, those two circuits are identical up to variable names, and the verifier contracts are precisely identical. This means that proofs for one of the two circuits will also verify by the other circuit's verifier contract. Proofs can thus be used for an unintended action, such as using a `curveSingle-Exchange` proof for `curveMultiExchange` instead.

The different `curveAddLiquidity` entry points (in the CurveAddLiquidityAssetManager, CurveFS-NAddLiquidityAssetManager, and CurveMPAddLiquidityAssetManager contracts) all use the same `curveAddLiquidity` circuit. So here, too, can a proof that was intended to be used with one of those entry points be used for another entry point — analogous for the `curveRemoveLiquidity` entry points.

Impact

It is not ensured that proofs are only usable in the entry point that the user intended them for. No practical impact was identified at this time.

Recommendations

The suggested fix for Finding 3.18. ↗ would make the `curveMultiExchange` and `curveExchange` circuits distinct.

As there is no practical impact of using the same circuit for the three `curveAddLiquidity` (and three `curveRemoveLiquidity`) entry points, no change is necessary. If making sure that the proofs cannot be used for an unintended variant is desired, then this can of course be done by creating three variants of the circuit that use a different domain separator as in Finding 3.18. ↗. An alternative that would not require duplicating the circuit (which comes with the need to deploy more contracts and possible other practical disadvantages) would be to include a variant identifier in the data that is signed inside the circuit and expose that variant identifier as a public circuit variable, checking it in the contract, for example like this:

```
let m = std::hash::mimc::mimc_bn254(
    [
     DOMAIN_SEPARATOR_CURVE_ADD_LIQUIDITY,
     add_liquidity_variant, // this is a new public input to the
         circuit.
    note1,
    note2,
    note3,
    note4,
    nullifier1,
    nullifier2,
    nullifier3,
    nullifier4,
    pool,
    out_note_footer
]
);

let m_bytes = fuzk::to_bytes(m);

let v = std::schnorr::verify_signature(pub_key[0], pub_key[1], signature,
    m_bytes);
```

In CurveAddLiquidityAssetManager, CurveFSNAddLiquidityAssetManager, and CurveMPAddLiquidityAssetManager, it might then be checked that `add_liquidity_variant` is equal to a constant that differs between these three variants.

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit 0094b4e1 ↗.

3.22. PoolAddress uses incorrect constant

Target	PoolAddress		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The PoolAddress library included in `contracts/defi/uniswap/libraries/PoolAddress.sol` uses an incorrect value for the `POOL_INIT_CODE_HASH` constant, at least for mainnet pools. Issue #348 ↗ on the Uniswap v3-perihphery repository points this out. We also verified that the correct value is

```
bytes32 internal constant POOL_INIT_CODE_HASH =
    0xe34f199b19b2b4f47f68442619d555527d244f78a3297ea89325f843f87b8b54;
```

using the following Python snippet with the contract-creation code from one of the Uniswap pools on mainnet.

```
#!/usr/bin/env python3
# took this from
    https://etherscan.io/address/0x4e68ccd3e89f51c3074ca5072bbac773960dfa36#code
text = '6101...'
code = bytes.fromhex(text)

from Cryptodome.Hash import keccak
keccak_hash = keccak.new(digest_bits=256)
keccak_hash.update(code)
print(keccak_hash.hexdigest())
```

Impact

Results of `computeAddress` will be incorrect for mainnet pools. However, it appears that the PoolAddress library is not used at all, so there is no impact in practice.

Recommendations

As the PoolAddress appears to be unused, we recommend removing it if it is not needed anymore. Otherwise, update the constant to the correct value:

```
bytes32 internal constant POOL_INIT_CODE_HASH =
    0xa598dd2fba360510c5a8f02f44423a4468e902df5857dbce3ca162a43a3a31ff;
    0xe34f199b19b2b4f47f68442619d555527d244f78a3297ea89325f843f87b8b54;
```

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `fe268599` ↗.

3.23. Uniswap liquidity provision has unnecessary deadline

Target	UniswapLiquidityAssetManager		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The UniswapLiquidityAssetManager contract's `uniswapLiquidityProvision` function first calls the `_releaseFundsAndPrepareLiquidityProvisionArgs` to prepare arguments for liquidity provision, and then it calls `_executeLPMint`, which calls the Uniswap NonfungiblePositionManager's `mint` function to execute the mint. The arguments for this prepared by `_releaseFundsAndPrepareLiquidityProvisionArgs` include the following deadline:

```
mintParams.deadline = block.timestamp + 2 minutes;
```

As the call to `nonfungiblePositionManager.mint` happens in the same transaction, the time of execution of the mint will always be the value of `block.timestamp` at the moment `mintParams.deadline` is set. As long as `mintParams.deadline` is set to a value of at least `block.timestamp`, the deadline will always be respected, and the `2 minutes` is unnecessary.

The purpose of the `deadline` argument is for transactions that have been prepared off chain, which may be delayed from the moment the user sets the parameters (for example in the mempool or due to using a hardware device to sign the transaction) until they land on-chain. If such usage of the deadline functionality was intended, then the deadline should be set by the user (or their client/front-end), included as a public value in the circuit, and be part of the signed data in the circuit. For example, the user's client could set the deadline to the timestamp that is two minutes in the future at the time that the user generates the proof.

Impact

The addition of `2 minutes` is unnecessary but has no negative impact in itself. The deadline functionality is not used to protect against slippage, however.

Recommendations

If the Uniswap deadline functionality is intended to be used to protect against slippage, then the value of `deadline` should be obtained from a public value in the circuit, and that value should be part of the signed data in the circuit.

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `4df0140f` ↗.

3.24. Wrong nullifier computation

Target	curveService.ts		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the `generateCurveAddLiquidityProof` function, if the value of `rho` happens to be null, then the nullifier is also set to zero:

```
const nullifier1 = param.note1.rho === 0n ? 0n :
    calcNullifier(param.note1.rho,
    fuzkPubKey);
const nullifier2 = param.note2.rho === 0n ? 0n :
    calcNullifier(param.note2.rho,
    fuzkPubKey);
const nullifier3 = param.note3.rho === 0n ? 0n :
    calcNullifier(param.note3.rho,
    fuzkPubKey);
const nullifier4 = param.note4.rho === 0n ? 0n :
    calcNullifier(param.note4.rho,
    fuzkPubKey);
```

A zero nullifier will not be verified properly by the verifier.

Impact

In the unlikely case where `rho` is zero, the nullifier is set to zero and will not verify, later making the corresponding note unspendable.

Recommendations

The nullifier should be computed such that the verifier is able to verify it correctly.

Remediation

This issue has been acknowledged by Singularity, and a fix was implemented in commit `2b464a25` ↗. The verification has been to change to check the `amount` value instead of `rho`.

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. Some natspec comments and interfaces are inaccurate

During review of the codebase, we noticed some places where behavior of the code deviate from the documented behavior.

For example, the NatSpec documentation for the `_convertToEthIfNecessary` function in the UniswapCoreAssetManager contract documents it as returning `address(0)` in the WETH conversion case. However, the return value in that case is actually `ETH_ADDRESS=0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`.

As another example, the IAssetPool interface contains the following signatures for the `setAssetManager` and `getAssetManager` functions:

```
function setAssetManager(address assetManager) external;

function getAssetManager() external view returns (address);
```

However, in the implementation in the BaseAssetPool contract, the signatures, and also the name of the function to retrieve asset-manager information is different:

```
function setAssetManager(
    address assetManager,
    bool registered
) external onlyOwner {
    _assetManagerRegistered[assetManager] = registered;
}

function getAssetManagerRegistration(
    address assetManager
) external view returns (bool) {
    return _assetManagerRegistered[assetManager];
}
```

This divergence likely arose after a change in the implementation to allow multiple assset managers at the same time.

We recommend ensuring that any specification and documentation stays up to date with the functionality implemented.

Singularity made the changes related to the two examples above in commits `fe268599` ↗ and

3a70d36f ↗, and informed us that they plan to go through all NatSpec comments before launch to ensure accuracy.

4.2. BN254 security level

The default proving backend of Noir is Aztec's Barretenberg, which uses the curve BN254 together with the Grumpkin curve as a cycle of curves.

With the recent advance in optimizing the special tower number field sieve (STNFS), the BN254 curve is not anymore 128-bit security but estimated to be around 102 bits ↗. It means that the security level of the proofs generated by the Barretenberg backend have the same level of security. Zcash has switched to BLS12-381 ↗ curve for this reason.

The proof system is not directly at risk at the moment of writing the report, but it may not be excluded that future advances in the ongoing research would lower the security level of this curve. A long-term project would be to use a different proving backend with a larger security margin.

4.3. Interpretability of `uniswapLiquidityProvision` return value

The UniswapLiquidityAssetManager contract's `uniswapLiquidityProvision` function returns a value `FeesDetails memory feesDetails`, which is constructed by the `_releaseFundsAndPrepareLiquidityProvisionArgs` function. The relevant code is the following:

```
feesDetails.serviceFee =
    feesDetails1.serviceFee +
    feesDetails2.serviceFee;

feesDetails.relayerRefund =
    args.relayerGasFees[0] +
    args.relayerGasFees[1];
```

We can see that the two types of fees are set by summing up fees arising from the two assets involved in liquidity provision. Thus for both of these sums, the two summands reflect different units so that their sum does not have any immediate interpretation. For example, one summand might be `5 USDC`, and the other summand might be `1 ETH`. Interpretability of the meaning of the sum 6 of the two numerical values is then questionable.

Singularity removed `feesDetails` from the codebase in fe268599 ↗

4.4. Assumption on non-fungible position identifiers

Uniswap V3 positions are represented by a token following EIP-721 ↗. Those tokens are minted at the Uniswap side by the NonfungiblePositionManager ↗ contract. Each token has a a unique `uint256` identifier. EIP-721 explicitly warns that "callers SHALL NOT assume that ID numbers have any specific pattern to them, and MUST treat the ID as a 'black box'". The token ID is passed as a field element to the collecting fee circuit. It would make the circuit fail in the case the ID is not in the field-element range; the ID value is above the prime number defining the field. A user would not be able to collect fees of such NFT. It is not an issue in practice in this particular usage since, in the Uniswap contract, the IDs are generated consecutively as it is filed. At the timing of writing, the current ID is about 700000 ↗. But this is a nonstandard usage of NFT token IDs and may be problematic if used for other use cases.

4.5. Removed Uniswap positions are kept in the ERC-721 pool

When a Uniswap liquidity position is removed by the user, the Uniswap NonfungiblePositionManager NFT for that (now empty) position is not burned but transferred back to the ERC-721 asset pool, as can be seen in the UniswapLiquidityAssetManager contract's `uniswapRemoveLiquidity` function:

```
function uniswapRemoveLiquidity(
    UniswapRemoveLiquidityArgs memory args,
    bytes calldata proof
)
    public
    returns (
        TransferFundsToVaultWithFeesAndCreateNoteData memory dataToken1,
        TransferFundsToVaultWithFeesAndCreateNoteData memory dataToken2
    )
{

    // ...

    nonfungiblePositionManager.decreaseLiquidity(
        _prepareRemoveLiquidityParams(args, liquidity)
    );

    (uint256 amount0, uint256 amount1) = nonfungiblePositionManager.collect(
        _prepareCollectFeeParams(args.positionNote.amount)
    );

    _transferERC721PositixonToVault(args.positionNote.amount);

    // ...
}
```

As there will be no notes associated to this NFT — and as even if there were, there would be no way to add liquidity to an existing position — it is unnecessary to keep the NonfungiblePositionManager NFT around. It might thus be cleaner to burn it instead:

```
nonfungiblePositionManager.decreaseLiquidity(
    _prepareRemoveLiquidityParams(args, liquidity)
);

(uint256 amount0, uint256 amount1) = nonfungiblePositionManager.collect(
    _prepareCollectFeeParams(args.positionNote.amount)
);

_transferERC721PositixonToVault(args.positionNote.amount);
nonfungiblePositionManager.burn(args.positionNote.amount);
```

Singularity changed the contract to instead burn NFTs when a position is removed. The changes were implemented in `daf2b4a8` ↗

4.6. Gas saving in MerkleTreeOperator by reusing `merkleRoots`

The MerkleTreeOperator contract has a state variable that is a dynamic array of `bytes32` containing all previous Merkle roots: `bytes32[] public merkleRoots;`. This array is not used to check which Merkle roots have already appeared, however; this is done with the `merkleRootIsAllowed` function, which uses the `merkleRootsAllowed` mapping instead:

```
function merkleRootIsAllowed(
    bytes32 _merkleRoot
) external view returns (bool) {
    return merkleRootsAllowed[_merkleRoot];
}
```

From the `merkleRoots` array, only the last component is ever used to get the most recent Merkle root. It would thus be possible to make `merkleRoots` of type `bytes32` instead and update it with `merkleRoots = leaf;` instead of `merkleRoots.push(leaf);`, replacing the access `merkleRoots[merkleRoots.length - 1]` with merely `merkleRoots`. This would save a decent amount of gas, as writing in previously used storage locations is cheaper than writing to previously unused locations (using the array also requires additional reads and writes to access and update the length).

Singularity implemented the suggested changes in `1fd85c02` ↗

4.7. Gas savings and specification of Mimc254 implementation

There are two distinct implementations of the MiMC hash function in the Mimc254 contract: `_mimc` and `_mimcForTree`. The main difference is that the latter one has hardcoded the exponent to be 7 while the former takes a 32-bit exponent as an argument and uses the `_pow32` function for exponentiation. However, `_mimc` is only used in `mimcBn254ForNote` and `mimcBn254ForRoute`, where the exponent is always 7. Using the specialized variant in those functions as well could save gas.

It should also be considered whether the functions in the Mimc254 contract

1. may assume inputs are smaller than `P`,
2. could be passed inputs bigger than or equal to `P`, but it should revert if that happens, and
3. should handle inputs bigger than or equal to `P` by reducing them modulo `P`.

Option 3 would make the relevant MiMC functions into hash functions where it is easy to obtain second preimages. A hash function is second-preimage resistant if, given `a`, it is intractible to find a `b != a` such that `hash(a) = hash(b)`, and with option 3, both `a` and `a+P` would yield the same hash. We thus recommend options 1 or 2 as well as documenting which behavior the functions should follow with comments. In the case of option 1, it would be the responsibility of caller functions to ensure only inputs smaller than `P` are passed.

The `mimcBn254ForTree` function has a line `r = (((r + elem) % P) + h) % P;` that would revert due to arithmetic overflow for some large values of `elem` (this variable is obtained from the arguments passed to the function). If `elem` can be assumed to be smaller than `P` at this point, then it would be more gas efficient to replace this line by `r = (r + elem + h) % P;`, reducing the number of reductions modulo `P`. This is as `P < 2^256 / 3`, so the addition of three summands that are smaller than `P` will not cause an arithmetic wraparound modulo `2^256`.

There are also some further unnecessary reductions modulo `P`. For example, the value of `r` will already be reduced modulo `P` after the loops in the functions `mimcBn254ForNote`, `mimcBn254ForTree`, and `mimcBn254ForRoute`. Thus `return r % P;` could be replaced by `return r`. Similarly, the return value of `_pow32` is already reduced mod `P`, so the line `uint256 h = _pow32(t, _exp) % P;` in `_mimc` could be `uint256 h = _pow32(t, _exp);`.

For such changes, we recommend adding comments to the code documenting why it is not necessary to reduce modulo `P` or why additions will not overflow.

In addition, the constant `P` is defined in the Mimc254 contract and redefined in the MerkleTreeOperator and BaseAssetManager contracts but not used there.

Singularity made the suggested changes in commits `fc9f1d01` ↗ and `1fd85c02` ↗.

4.8. Confusing argument order in ERC721AssetPool `approve`

The ERC721AssetPool contract's `approve` function is implemented as follows:

```
function approve(
    address nft,
    uint256 tokenId,
    address to
) external onlyAssetManger transferNotLocked {
    require(
        IERC721(nft).ownerOf(tokenId) == address(this),
        "ERC721AssetPool: NFT does not belong to asset pool"
    );

    IERC721(nft).approve(to, tokenId);
}
```

The argument order of `tokenId` and `to` is unusual here, as normally, as can be seen in the last line of the function body, the recipient is listed before the token ID or amount. This could cause confusion in code calling this function. In the current codebase, this function does not appear to be used so far.

Singularity removed this function in `fe268599` ↗.

4.9. Centralization risks

There are several types of privileged accounts for the Singularity contracts, as owners of different types of contracts:

- Verifier hub
- Asset pools
- Asset managers
- Fee manager
- Keyring manager
- Merkle tree operator
- Relayer hub

An attacker gaining access to most of these privileged accounts could use them to drain the asset pools of funds or do other damage:

- The owner of a verifier hub can change the verifier contract for the different circuit names. This allows replacing the verifier with one that passes for the attacker's proofs, so that the attacker can, for example, mint notes with arbitrary amounts.

- The owner of an asset pool can register asset managers for it, and asset managers can release funds from the asset pool directly.
- The owner of an asset manager can change the address used for various other contracts, such as the verifier hub. By changing the verifier hub to a contract the attacker controls, they can again make any of their proofs succeed, thereby for example minting arbitrary notes.
- The owner of the fee manager can withdraw collected service fees. Furthermore, they can set the service fee. By setting it to 100%, they will capture all of the value in withdrawals until this is fixed.
- The owner of the keyring manager can disable keyring checks and change the keyring policy ID.
- The owner of the Merkle tree operator can set the enabled asset managers, which in turn can add notes to the Merkle tree. Using this, an attacker could add notes for arbitrary amounts.
- The owner of the relayer hub can add and remove relayers. Removing all relayers would make it impossible to withdraw funds from the darkpool.

Singularity informed us that the above contract owners will be multi-sig wallets. We simply note the above centralization risks here as the verification of the multi-sig mechanism and security of key custody is beyond the scope of this audit.

4.10. MiMC hash-length extension attack

The MiMC hash implementation in Noir stdlib is prone to a possible hash-length extension attack. Someone is able to create a hash of a secret value concatenated with values they control without knowing the secret values themself.

If they know the hash $h_1 = H(\rho)$ of a secret value ρ, they can compute a new hash $h_2 = H(\rho, x)$ for any value x just knowing the value of h_1.

Here is an example in JavaScript using the TypeScript implementation of Singularity:

```
> h1 = singularity.mimc_bn254([secret])
6330754968575099758421459477990008027487643103853686899483588894031458107821n
> ht = (h1 + 2n + singularity.mimc(2n, h1, singularity.constants, 7n))
    % singularity.P
14068749358199635716244899654758139174543955001947899909971002288403713001091n
> h2 = singularity.mimc_bn254([secret, 2n])
14068749358199635716244899654758139174543955001947899909971002288403713001091n
> h2 == ht
true
```

This primitive is also implemented in Mimc254 to be compatible with the Noir language circuit calls to this primitive in the Noir standard library. Therefore, the Noir language implementation is impacted and the Noir developers have been contacted.

No practical exploitation in the circuits has been found in the current implementation. Nevertheless, this needs to be taken into account for further development of the project. The same issue has been reported to the gnark-crypto ↗ project.

The documentation and code comments should warn about this property to avoid misusage. To avoid this threat, the Poseidon hash function, implemented in Noir, can be used.

Singularity added a warning to the code in commit `064c573c` ↗.

5. Assessment Results

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

During our assessment on the scoped Singularity contracts, we discovered 24 findings. Seven critical issues were found. Two were of high impact, four were of medium impact, six were of low impact, and the remaining findings were informational in nature.

5.1. 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.