SyncSwap

Smart Contract Security Assessment

May 4, 2023

Prepared for:

Nakato Konata

SyncSwap Labs

Prepared by:

Filippo Cremonese and Varun Verma

Zellic Inc.

Contents

About Zellic

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

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

1 Executive Summary

Zellic conducted a security assessment for SyncSwap Labs from April 17th to April 20th, 2023. During this engagement, Zellic reviewed SyncSwap's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

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

- Are assets deposited in the vault contract safe?
- Is the flash loan functionality implemented safely?
- Does the stable pool implement the Stableswap curve correctly?
- Does the classic pool implement the constant product invariant correctly?
- Is the unilateral burn feature implemented correctly?
- Is the unbalanced mint feature implemented correctly?
- Is the logic for computing fees implemented correctly?

1.2 Non-goals and Limitations

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

- Contracts not in scope, including the router contract
- 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. In particular, the lack of documentation specific to Stableswap and of a comprehensive test suite limited our efficiency.

1.3 Results

During our assessment of the SyncSwap contracts, we identified one issue of low impact. No critical issues were found.

Additionally, we recorded notes and observations taken during the assessment for SyncSwap Labs's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

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

2 Introduction

2.1 About SyncSwap

Powered by zero-knowledge technology, SyncSwap brings more people easy-to-use and low-cost DeFi with complete Ethereum security. It aims to provide a seamless and efficient decentralized exchange on the zkSync Era network.

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

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

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

2.3 Scope

The engagement involved a review of the following targets:

SyncSwap Contracts

Repository	https://github.com/syncswap/core-contracts
Version	core-contracts: `937645e5f751eb1c33e66bac2b4113094de6b673`
Programs	• SyncSwapStablePool • SyncSwapClassicPool • SyncSwapVault
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six person-days. 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:

Filippo Cremonese, Engineer
fcremo@zellic.io

Varun Verma, Engineer
varun@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

April 14, 2023	Kick-off call
April 17, 2023	Start of primary review period
April 20, 2023	End of primary review period

3 Detailed Findings

3.1 Lack of input validation leading to potentially dangerous calls

- **Target**: SyncSwap Pools
- **Category**: Code Maturity
- **Likelihood**: N/A
- **Severity**: High
- **Impact**: Low

Description

Multiple functions contain commented-out assertions that originally performed input validation. Due to the resulting lack of validation, it is possible to cause the SyncSwap pool contracts to invoke another contract with incorrect parameters via the `ICallback` interface.

For instance, the `burnSingle` function of both SyncSwapStablePool and SyncSwap-ClassicPool might invoke the callback contract as well as return an unexpected value, since `params.tokenOut` is not constrained to be either `token0` or `token1`.

```
function burnSingle(/* [ ... ] */) external override nonReentrant returns
    (TokenAmount memory _tokenAmount)
    {
    ICallback.BaseBurnSingleCallbackParams memory params;
    (params.tokenOut, params.to, params.withdrawMode) =
     abi.decode(_data, (address, address, uint8));
    // [ ... ]
    if (_callback ≠ address(0)) {
        // Fills additional values for callback params.
        params.sender = _sender;
        params.callbackData = _callbackData;
        ICallback(_callback).syncSwapBaseBurnSingleCallback(params);
    }
    // [ ... ]
    _tokenAmount = TokenAmount(params.tokenOut, params.amountOut);
    emit Burn(msg.sender, params.amount0, params.amount1,
    params.liquidity, params.to);
}
```

This issue is also present in `swap`, which does not properly check `params.tokenIn`:

```
function swap(/* [ ... ] */) external override nonReentrant returns
    (TokenAmount memory _tokenAmount)
    {
    ICallback.BaseSwapCallbackParams memory params;
    (params.tokenIn, params.to, params.withdrawMode) =
    abi.decode(_data, (address, address, uint8));
    // [ ... ]
    if (_callback != address(0)) {
        // Fills additional values for callback params.
        params.sender = _sender;
        params.callbackData = _callbackData;
        ICallback(_callback).syncSwapBaseSwapCallback(params);
    }
    // Updates reserves with up-to-date balances (updated above).
    _updateReserves(params.balance0, params.balance1);
    _tokenAmount.token = params.tokenOut;
    _tokenAmount.amount = params.amountOut;
}
```

Impact

This does not directly impact the contracts in scope of this audit. However, it does allow an attacker to manipulate the pool contracts into invoking a third party contract with unexpected parameters. The consequences of this issue would depend on the implementation of the contract being invoked.

Recommendations

We generally recommend including conservative assertions validating all the inputs. If these checks are deemed too expensive, we suggest removing the commented-out code to improve readability and inserting a comment stating why such a check is not needed.

Since SyncSwap contracts cannot be upgraded, we recommend updating the documentation of the contracts to clearly communicate to users of the contract how a callback invocation can be verified. Implementors of the `ICallback` interface should carefully analyze how the potentially untrusted parameters could cause issues in their contracts.

In general, depending on the usage, the two following strategies should suffice to guard against exploitation:

- if the callback is only supposed to be invoked when the Pool contract is called by a trusted address, check that the `sender` field of the callback parameters is in the set of trusted addresses
- otherwise (or in addition), ensure that `((tokenIn == pool.token0() && tokenOut == pool.token1()) || (tokenIn == pool.token1() && tokenOut == pool.token0()))`

Remediation

The development team has acknowledged this issue and added a notice warning about the untrusted nature of `tokenIn` and `tokenOut` in commit `5285a3a7`.

4 Discussion

This section contains notes taken during the assessment with the purpose of understanding the AMM behavior; SyncSwap bears heavy resemblance with Sushi's Trident — specifically, the classic pool is similar to Trident `ConstantProductPool`, and the stable pool is similar to Trident `HybridPool`. We spent part of the engagement comparing SyncSwap against other AMM implementations. We performed a comparative analysis against Trident since it has received extensive scrutiny from multiple security firms, which makes it a good candidate to highlight potentially dangerous differences in behavior. However, we note that despite implementing largely equivalent logic, the code is not a simple fork; additional features and gas efficiency optimizations complicated the comparative analysis. At the time of our review, no SyncSwap-specific documentation was available regarding the AMM pricing mechanisms used by the pools.

General differences between Trident and SyncSwap include the support for variable fees (obtained from the pool master, potentially varying on the asset pair being swapped) as well as for callbacks invoked when an operation is performed and the addition of a unilateral LP token burn functionality.

Definitions

Some variables recur multiple times throughout this section. Unless otherwise stated, these definitions should be assumed:

- b_0 and b_1: The actual balance of the assets of the pool at the start of the call.
- r_0 and r_1: The cached balance of the assets of the pool; they differ from the actual balances if assets were sent to the pool before the contract is called and had the possibility to update the cached value.
- a_0 and a_1: The amount of assets sent to the pool, computed as $b_i - r_i$.

Fees

SyncSwap has two types of fees: swap fees and protocol fees. Swap fees are used to compensate liquidity providers and are generally taken by virtually reducing the amount used as input to a swap by some percentage, which in turn reduces the amount received as output of the swap. The difference contributes to growing the total liquidity available in the pool, which increases the payout that is obtained by liquidity providers when redeeming their LP tokens. The pools also support protocol fees; these are generally paid by minting new LP tokens to an address configured to receive these fees.

Swap fees are configurable and obtained by invoking the pool master contract `getSw` apFee function; they can vary depending on the asset pair being swapped as well as the direction of the swap and the sender of the transaction:

$$fee_{Swap} = getSwapFee(sender, asset_0, asset_1)$$

Protocol fees ($fee_{Protocol}$) can also vary depending on the value returned by the pool master getProtocolFee function.

Both fees are computed by taking a percentage off of the input of the various operations; for the purposes of simplifying this exposition, fee_{Swap} and $fee_{Protocol}$ will be considered real numbers between 0 and 1. In reality, they are expressed as integers representing the numerator of a fraction with denominator `MAX_FEE == 10000`.

4.1 Classic pool

SyncSwap classic pool is based on this well-known constant product invariant:

$$\sqrt{b_0 * b_1} = K$$

Swap

To perform a swap, users must send the amount to be used as input and call `swap` (since this must be performed in a single transaction, it is normally delegated to the router contract).

The following calculations assume a swap from the first to the second asset of the pair that makes up the pool. The calculations in the reverse direction are specular. The amount of tokens resulting from the swap is computed as

$$\begin{aligned} amount_{InWithFee} &= a_0 * (1 - fee_{Swap}) \\ amount_{Out} &= \frac{amount_{InWithFee} * r_1}{r_0 + amount_{InWithFee}} \end{aligned}$$

Mint

The liquidity used to perform swaps is provided by liquidity providers who deposit liquidity in exchange for LP tokens. Liquidity providers must send the liquidity they want to provide to the pool and call `mint`. This is normally performed by the router contract.

The amount of LP tokens (liquidity) minted to the user is equal to

$$liquidity = \frac{(K_{New} - K_{Old}) * totalSupply}{K_{Old}}$$

where

$$K_{Old} = \sqrt{(r_0 + fee_0) * (r_1 + fee_1)}$$
$$K_{New} = \sqrt{b_0 * b_1}$$

The fee parameters are calculated to account for unbalanced minting: if the user providing liquidity does not provide both assets in a balanced way, the appropriate amount of the asset in excess is converted to the other asset to keep the balance, incurring in swap fees.

The $totalSupply$ parameter represents the total supply of LP tokens, including the newly minted LP tokens minted as protocol fees. If enabled, protocol fees cause the minting of additional LP tokens that are sent to an address responsible of collecting them. The amount to be minted is computed as

$$K_{Last} = K_{New}\ from\ previous\ execution = \sqrt{r_0 * r_1}$$
$$poolFeeLPTokens = \frac{totalSupply * (K_{Old} - K_{Last}) * fee_{Protocol}}{(1 - fee_{Protocol}) * K_{Old} + fee_{Protocol} * K_{Last}}$$

Burn

Opposite to minting, burning allows to get back liquidity in exchange for LP tokens. Users must send LP tokens to the pool before calling `burn` (within the same transaction), normally by using the router contract.

The amount of liquidity returned to the user is calculated by

$$a_0 = \frac{liquidity * b_0}{totalSupply}$$
$$a_1 = \frac{liquidity * b_1}{totalSupply}$$

where $liquidity$ is the amount of LP tokens being burned by the user and $totalSupply$ is the amount of LP tokens in circulation.

We noticed a difference between SyncSwap and Trident: where SyncSwap uses the current balance to compute the amounts returned to the user, Trident uses the cached reserves values. According to the SyncSwap team, this change was made to improve gas efficiency. While the balance can be manipulated upwards before calling `burn` without affecting the cached reserves values (by transferring tokens to the pool), we do not believe this behavior can be used profitably by an attacker.

Unilateral burn

SyncSwap also allows to burn LP tokens in exchange for one of the two assets of a pool. This is equivalent to burning LP tokens for the pair and swapping the unwanted asset for the desired one.

The amounts of assets gained by burning LP tokens is computed identically to the regular burn functionality, and the conversion rate between the pair matches the amount that would be obtained by a regular swap operation.

4.2 Stable pool

SyncSwap stable pool implements the Stableswap AMM curve. A significant amount of time of our review was dedicated to comparing SyncSwap against other Stableswap implementations used as reference. The code bears much similarity to Trident `HybridPool`; we found SyncSwap to be an equivalent but generally more gas-efficient implementation.

Stableswap primer

The Stableswap AMM curve was initially introduced by Curve Finance and is designed for increasing capital efficiency in pools containing assets that are intended to maintain a stable price. This is achieved by reducing price slippage compared to other AMM models such as constant product, thus encouraging more and bigger swaps and collecting more fees overall.

The Stableswap invariant is as follows:

$$An^n \sum x_i + D = ADn^n + \frac{D^{n+1}}{n^n \prod x_i}$$

Where x_i are the amount of assets in the pool, n is the number of assets in the pool, and D is a constant needed to balance both sides of the equation.

The curve behavior is a hybrid between the well-known constant product and the constant sum curves, which can be tuned via the A parameter. For increasing values of A, the behavior of Stableswap approaches that of the constant sum curve, whereas for $A = 0$, the equation is equivalent to the constant product curve.

SyncSwap implementation of Stableswap is optimized for a pool with two assets and a hardcoded $A = 1000$, reducing the invariant to

$$1000 * 2^2 * (x_0 + x_1) + D = 1000 * D * 2^2 + \frac{D^3}{2^2 * x_0 * x_1}$$
$$4000 * (x_0 + x_1) + D = 4000 * D + \frac{D^3}{4 * x_0 * x_1}$$

For more information on Stableswap, refer to the original paper describing the curve at https://classic.curve.fi/files/stableswap-paper.pdf.

Note: To simplify the exposition, all amounts will be considered normalized to the same amount of decimal digits.

Swap

To perform a swap, users must send the amount to be used as input and call `swap` (since this must be performed in a single transaction, it is normally delegated to the router contract).

The following calculations assume a swap from the first to the second asset of the pair that makes up the pool. The calculations in the reverse direction are specular and omitted.

In order to compute the amount of tokens obtained from the swap, the D constant is first obtained from the curve equation via iterative approximation. The recurrence relation is defined as

$$s = r_0 + r_1, p = r_0 r_1$$
$$\delta P_i = \frac{D_i^3}{4p}$$
$$D_0 = s$$
$$D_{i+1} = \frac{(nsA + 2\delta P_i)D}{(nA - 1)D + 3\delta P_i} = \frac{(2000s + 2\delta P_i)D_i}{1999D_i + 3\delta P_i}$$

Once D has been determined, the net amount of input tokens x is computed by subtracting the protocol fees from the input amount. The amount of output tokens y is then computed by another iterative approximation process using the recurrence relation

$$
\begin{aligned}
c &= \frac{D^2}{2x} * \frac{D}{2nA} = \frac{D^3}{8000x} \\
b &= x + \frac{D}{nA} = x + \frac{D}{2000} \\
y_0 &= D \\
y_{i+1} &= \frac{y_i^2 + c}{2y_i + b - D}
\end{aligned}
$$

Recurrence relation analysis

The above recurrence relations appear to implement a slight variant of Newton's root finding method. Both iterative computations are continued until they converge (e.g., $D_{i+1} - D_i <= 1$), and up to a maximum of 256 iterations.

During our review we observed that the recurrence relation used to calculate D appears to be slightly different from the one that would be obtained by applying the textbook Newton's method. Given the invariant expressed as a function of D,

$$
\begin{aligned}
p &= x_0 x_1 \\
s &= x_0 + x_1 \\
I(D_i) &= \frac{D_i^3}{4p} + 3999D_i - 4000s
\end{aligned}
$$

Newton's method would define D_{i+1} as

$$
D_{i+1} = D_i - \frac{I(D_i)}{I'(D_i)} = D_i - \frac{\frac{D_i^3}{4p} + 3999D_i - 4000s}{\frac{3D_i^2}{4p} + 3999} = \frac{4000s + \frac{D_i^3}{2p}}{3999 + \frac{3D_i^2}{4p}}
$$

which is slightly different from the recurrence relation used by the implementation that is equivalent to

$$
D_{i+1} = \frac{2000s + \frac{D_i^3}{2p}}{1999 + \frac{3D_i^2}{4p}}
$$

Numerical experimentation showed that the two equations converge to the same value.

Further investigation also confirmed that the equation used by SyncSwap is equivalent to the implementation used by Trident and even to the original Curve Finance implementation of Stableswap. To the best of our knowledge, no official documentation exists regarding the derivation of the formulas used by the original Curve Finance implementation. Since all implementations exhibit the same behavior, we consider it the de facto standard and the desired behavior.

Mint

The liquidity used to perform swaps is provided by liquidity providers who deposit liquidity in exchange for LP tokens. Liquidity providers must send the liquidity they want to provide to the pool and call `mint`. This is normally performed by the router contract.

The amount of LP tokens minted to the user is equal to

$$\begin{aligned}
D_r &= D(r_0 + Fee_0, r_1 + Fee_1) \\
D_b &= D(b_0, b_1) \\
MintedLiquidity &= \frac{(D_b - D_r) * LPSupply}{D_r}
\end{aligned}$$

where $D(x, y)$ computes the D parameter by iterative approximation as described in the previous section.

The Fee_i parameters represent swap fees charged to the user if the liquidity sent to the pool was imbalanced with respect to the current exchange rate, calculated with the same formula used by the swap process.

The $LPSupply$ parameter represents the total supply of LP tokens, including the newly minted LP tokens minted as protocol fees. Protocol fees can vary depending on the value returned by the pool master `getProtocolFee()` function, which for the purposes of this exposition we will treat as a number between zero and one. If enabled, protocol fees cause the minting of additional LP tokens that are sent to an address responsible of collecting them. The amount to be minted is computed as

$$Fee_{Protocol} = \frac{LPSupply * (D_r - D_{Last}) * FeePct}{(1 - FeePct) * D_r + FeePct * D_{Last}}$$

where D_{Last} is the value of the invariant $D_r = D$ saved from the previous execution, and $FeePct = getProtocolFee()$. Note that protocol fees are minted only if $D_r - D_{Last} > 0$

Burn

Opposite to minting, burning allows to get back liquidity in exchange for LP tokens. Users must send LP tokens to the pool before calling `burn` (within the same transaction), normally by using the router contract.

The amount of assets returned to the user is calculated by

$$amount_0 = \frac{BurnedLP * b_0}{LPSupply}$$
$$amount_1 = \frac{BurnedLP * b_1}{LPSupply}$$

Unilateral burn

SyncSwap also allows to burn LP tokens in exchange for one of the two assets of a pool. This is equivalent to burning LP tokens for the pair and swapping the unwanted asset for the desired one.

The amounts of assets gained by burning LP tokens is computed identically to the regular burn functionality, and the conversion rate between the pair matches the amount that would be obtained by a regular swap operation.

4.3 Attack surface evaluation

In this section we discuss the posture of SyncSwap with respect to some of the most common threats faced by AMMs.

Reentrancy

The pool contracts are protected against reentrancy exploits by using an appropriate modifier on all state-changing functions.

Vaults are also protected in the same way, applying the `nonReentrant` modifier on the core vault functionality (e.g., deposit, transfer, withdraw) as well as the flash loan functionality.

Being independent contracts, nothing prevents execution flow from performing a pool operation nested within a vault operation; for example, it is possible to take a flash loan, execute a swap, and repay the flash loan. The possible interactions between the two contracts do not appear to be exploitable for profit.

Arithmetic over/underflows

The contracts perform several arithmetic operations inside `unchecked` blocks, in which the compiler does not insert checks to guard against over or underflows; these unchecked math operations are usually commented with a rationale justifying why `unckecked` code can be used.

MEV

The possibility to extract value from swap transactions by reordering them or performing a sandwich attack is a common attack vector for AMM pools. This attack vector exploits the automatic pricing behavior inherent to the working of an AMM, and as such, mitigations cannot exclusively be implemented directly in the AMM. Nevertheless, usage of curves such as Stableswap certainly help in reducing the impact of a MEV attack.

The most common mitigation against MEV is the implementation of slippage checks in the contracts that invoke the pool where the swap is performed. The router contract that does this for SyncSwap was not in scope of this audit, but a limited inspection appears to confirm anti-slippage checks are performed correctly.

5 Threat Model

This section provides a threat model description for various functions. As time permitted, we analyzed each function in the smart 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: SyncSwapVault.sol

Function: `depositETH(address to)`

Allows the caller to deposit ETH.

Inputs

- `to`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the recipient of the deposit.

Branches and code coverage (including function calls)

Intended branches

Deposits the balance of the transaction to the recipient.

- [x] Test coverage

Function: `deposit(address token, address to)`

Allows to deposit assets (both native ETH and other ERC20 tokens) into the vault. The caller should first transfer the assets to the vault and then call `deposit`. This must be accomplished whithin the same transaction to prevent others from stealing the deposit.

Inputs

- `token`

  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the asset that is being deposited.
- `to`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the recipient of the deposit.

Branches and code coverage (including function calls)

Intended branches

Allows to deposit native ETH.

- [x] Test coverage

Allows to deposit an ERC20 token.

- [x] Test coverage

Negative behavior

ETH sent to the contract when depositing an ERC20 token.

- [ ] Negative test

Function call analysis

- `rootFunction → IERC20(wETH).balanceOf(address(this))`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `rootFunction → IWETH(wETH).withdraw(amount)`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** Not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

Function: `transferAndDeposit(address token, address to, uint256 amount)`

Allows to transfer and deposit an ERC20 token in a single call. Also allows to deposit ETH.

Inputs

- `token`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the asset to be deposited.
- `to`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the recipient of the deposit.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None if operating on ERC20; `amount == msg.value` if operating on ETH.
  - **Impact**: Determines the amount to be transferred.

Branches and code coverage (including function calls)

Intended branches

Allows to deposit an ERC20: transfers and deposits the requested amount.

☑ Test coverage

Allows to deposit ETH.

☐ Test coverage

Negative behavior

ERC20 balance insufficient to cover the deposit.

☑ Negative test

Function call analysis

- `rootFunction` → `IWETH(wETH).transferFrom(msg.sender, address(this), amount)`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.

  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is not a concern.
- `rootFunction → IWETH(wETH).withdraw(amount)`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Cannot revert or reenter.
- `rootFunction → TransferHelper.safeTransferFrom(token, msg.sender, address(this), amount)`
  - **What is controllable?** `token`, `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented via the `nonReentrant` modifier.

Function: `transfer(address token, address to, uint256 amount)`

Allows to transfer assets from the caller balance to another recipient.

Inputs

- `token`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the asset to be transferred.
- `to`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the recipient of the transfer.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: `amount ≥ balances[token][msg.sender]`.
  - **Impact**: Determines the amount to be transferred.

Branches and code coverage (including function calls)

Intended branches

Transfers the given asset of the specified amount to the given recipient.

☑ Test coverage

Negative behavior

Balance of `msg.sender` is insufficient to cover the transfer.

- [x] Negative test

Function: `withdrawAlternative(address token, address to, uint256 amount , uint8 mode)`

Allows to withdraw assets from the vault; in case an ETH/WETH withdrawal is requested, allows to specify whether the output should be ETH or WETH.

Inputs

- `token`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the asset to be transferred.
- `to`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the recipient of the withdrawal.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: `balances[token][msg.sender] ≥ amount`.
  - **Impact**: Determines the amount to be transferred.
- `mode`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the behavior of ETH/WETH transfers.

Branches and code coverage (including function calls)

Intended branches

Handles ERC20 withdrawals.

- [ ] Test coverage

Handles ETH mode 2 withdrawals (wraps and transfers WETH).

- [ ] Test coverage

Handles ETH mode 0/1 withdrawals (transfers ETH).

☐ Test coverage

Handles WETH mode 1 (transfers ETH).

☐ Test coverage

Handles WETH mode 0/2 (transfers WETH).

☐ Test coverage

Negative behavior

Reverts if user balance is insufficient.

☐ Negative test

Function call analysis

External calls have the same profile of `transfer`.

Function: `withdrawETH(address to, uint256 amount)`

Allows to withdraw ETH from the vault.

Inputs

- `to`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the recipient of the transfer.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: `amount ≤ balances[NATIVE_ETH][msg.sender]`.
  - **Impact**: Determines the amount to be transferred.

Branches and code coverage (including function calls)

Intended branches

Transfers the requested amount of ETH.

☐ Test coverage

Negative behavior

Reverts if `msg.sender` balance is insufficient.

☐ Negative test

Function call analysis

- `rootFunction` → `TransferHelper.safeTransferETH(to, amount)`
  - **What is controllable?** `to`, `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** The return value of the internal call must be `bool(true)` or the transaction is reverted.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented via the `nonReentrant` modifier.

Function: `withdraw(address token, address to, uint256 amount)`

Allows to withdraw assets from the vault.

Inputs

- `token`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the asset to withdraw (ETH corresponds to the special address `0x0`).
- `to`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Determines the recipient of the transfer.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: `amount ≤ balances[token][msg.sender]`.
  - **Impact**: Determines the amount to be withdrawn.

Branches and code coverage (including function calls)

Intended branches

Allows to withdraw ETH.

☑ Test coverage

Allows to withdraw an ERC20 asset.

- ☑ Test coverage

Negative behavior Reverts if `msg.sender` balance is insufficient.

- ☑ Negative test

Function call analysis

The function has three flows.

Transfer ETH

- `rootFunction` → `TransferHelper.safeTransferETH(to, amount)`
  - **What is controllable?** `to`, `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented with the `nonReentrant` modifier.

Transfer WETH

- `rootFunction` → `_wrapAndTransferWETH(to, amount)`
  - **What is controllable?** `to`, `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented with the `nonReentrant` modifier.
- `_wrapAndTransferWETH` → `IWETH(wETH).deposit{value: amount}()`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is not a concern (WETH is trusted and immutable).
- `_wrapAndTransferWETH` → `IWETH(wETH).transfer(to, amount)`
  - **What is controllable?** `to`, `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is not a concern (WETH is trusted and immutable).

Transfer other ERC20

- `_wrapAndTransferWETH` → `TransferHelper.safeTransfer(token, to, amount)`
  - **What is controllable?** `token`, `to`, `amount`.

    - **If return value controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented with the `nonReentrant` modifier.
- `TransferHelper.safeTransfer → address(token).transfer(to, value)`
    - **What is controllable?** `token`, `to`, `value`.
    - **If return value controllable, how is it used and how can it go wrong?** The return value must encode `bool(true)`, otherwise execution is reverted.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented with the `nonReentrant` modifier.

5.2 Module: VaultFlashLoans.sol

Function: `flashLoanMultiple(IFlashLoanRecipient recipient, address[] tokens, uint256[] amounts, byte[] userData)`

Allows to take a flash loan from the vault, taking multiple assets at the same time.

Inputs

- `recipient`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Determines the recipient that receives the borrowed assets.
- `tokens`
    - **Control**: Arbitrary.
    - **Constraints**: Elements must be sorted and unique.
    - **Impact**: Determines the assets to be borrowed.
- `amounts`
    - **Control**: Arbitrary.
    - **Constraints**: `amounts.length == tokens.length`; every amount is less than the corresponding vault balance.
    - **Impact**: Determines the amounts to be borrowed.
- `userData`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Arbitrary data passed to the recipient.

Branches and code coverage (including function calls)

Intended branches

Sends the borrowed funds to the recipient, invokes the recipient, and checks that the funds and interests are returned.

- ☐ Test coverage

Negative behavior

Reverts if the vault balance is not enough to cover the requested loan.

- ☐ Negative test

Reverts if the borrower does not return `ERC3156_CALLBACK_SUCCESS`.

- ☐ Negative test

Reverts if the flash loan is not repaid.

- ☐ Negative test

Reverts if `amounts.length ≠ tokens.length`.

- ☐ Negative test

Reverts if the vault does not have sufficient balance to cover the requested loan.

- ☐ Negative test

Function call analysis

- `rootFunction → preLoanBalances[i] = IERC20(token).balanceOf(address(this))`
  - **What is controllable?** `token`.
  - **If return value controllable, how is it used and how can it go wrong?** Used as the preloan balance of the contract, not controllable for legitimate tokens.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented via the `nonReentrant` modifier.
- `rootFunction → feeAmounts[i] = _calculateFlashLoanFeeAmount(amount)`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** The function returns the interests of the flash loan, proportional to `amount`.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

Reverts (e.g., due to overflow) are bubbled up; reentrancy is not a concern.

- `rootFunction → TransferHelper.safeTransfer(token, address(receiver), amount)`
  - **What is controllable?** `token`, `receiver`, `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** The return value of the internal call must be `bool(true)`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented via the `nonReentrant` modifier.
- `rootFunction → recipient.receiveFlashLoan(tokens, amounts, feeAmounts, userData)`
  - **What is controllable?** `tokens`, `amounts`, `userData`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented via the `nonReentrant` modifier.
- `rootFunction → postLoanBalance = IERC20(token).balanceOf(address(this))`
  - **What is controllable?** `token`.
  - **If return value controllable, how is it used and how can it go wrong?** Used as the postloan balance of the contract, not controllable for legitimate tokens.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented via the `nonReentrant` modifier.
- `rootFunction → _payFeeAmount(token, receivedFeeAmount)`
  - **What is controllable?** `token`, `receivedFeeAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented via the `nonReentrant` modifier.

Function: `flashLoan(IERC3156FlashBorrower receiver, address token, uint256 amount, byte[] userData)`

Allows to take a flash loan from the vault.

Inputs

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

    - **Impact**: Determines the contract that receives the flash loan.
- `token`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Determines the asset to be borrowed.
- `amount`
    - **Control**: Arbitrary.
    - **Constraints**: Must be lower of equal to the vault balance.
    - **Impact**: Determines the amount to be borrowed.
- `userData`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Data passed to the borrower contract.

Branches and code coverage (including function calls)

Intended branches

Transfers the borrowed amount to the contract, invokes its callback, and checks that funds and interests are returned.

☐ Test coverage

Negative behavior

Reverts if the vault balance is not enough to cover the requested loan.

☐ Negative test

Reverts if the borrower does not return `ERC3156_CALLBACK_SUCCESS`.

☐ Negative test

Reverts if the flash loan is not repaid.

☐ Negative test

Function call analysis

- `rootFunction → IERC20(token).balanceOf(address(this))`
    - **What is controllable?** `token`.
    - **If return value controllable, how is it used and how can it go wrong?** Used as the preloan balance of the contract, not controllable for legitimate tokens.
    - **What happens if it reverts, reenters, or does other unusual control flow?**

Reverts are bubbled up; reentrancy is prevented via the `nonReentrant` modifier.

- `rootFunction → _calculateFlashLoanFeeAmount(amount)`
  - **What is controllable?** `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** The function returns the interests of the flash loan, proportional to `amount`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts (e.g., due to overflow) are bubbled up; reentrancy is not a concern.
- `rootFunction → TransferHelper.safeTransfer(token, address(receiver), amount)`
  - **What is controllable?** `token`, `receiver`, `amount`.
  - **If return value controllable, how is it used and how can it go wrong?** The return value of the internal call must be `bool(true)`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented via the `nonReentrant` modifier.
- `rootFunction → receiver.onFlashLoan(msg.sender, token, amount, feeAmount, userData)`
  - **What is controllable?** `token`, `amount`, `userData`.
  - **If return value controllable, how is it used and how can it go wrong?** Return value must be `ERC3156_CALLBACK_SUCCESS` to signal successful execution.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented via `nonReentrant` modifier.
- `rootFunction → IERC20(token).balanceOf(address(this))`
  - **What is controllable?** `token`.
  - **If return value controllable, how is it used and how can it go wrong?** Used as the postloan balance of the contract, not controllable for legitimate tokens.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented via the `nonReentrant` modifier.
- `rootFunction → _payFeeAmount(token, receivedFeeAmount)`
  - **What is controllable?** `token`, `receivedFeeAmount`.
  - **If return value controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are bubbled up; reentrancy is prevented via the `nonReentrant` modifier.

6 Audit Results

At the time of our audit, the code was deployed to mainnet zkSync Era.

During our audit, one low impact finding was reported with the goal of improving the security posture and maintainability of the codebase.

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 audit version of 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.

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.