Zellic

# GammaSwap

## Smart Contract Security Assessment

**June 5, 2023**

*Prepared for:*

**Daniel Alcarraz**

GammaSwap Protocol

*Prepared by:*

**Ayaz Mammadov and Vlad Toie**

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 GammaSwap Protocol from May 15th to May 24th, 2023. During this engagement, Zellic reviewed GammaSwap'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:

- Could a malicious attacker manipulate the GammaSwap pools to drain capital from either short or long positions?
- Could GammaSwap pools be manipulated to liquidate positions?
- Are there any mechanisms that result in perverse incentives that could result in the nongrowth or stagnation of the pool?
- Are there any unexpected cases due to the interaction of the GammaSwap pools and the underlying pools (UniSwap, Balancer...)?

## 1.2 Non-goals and Limitations

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

- Failures of the underlying pool that would result in loss of capital for positions in the GammaSwap pools
- 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.

During the review process, due to time constraints, we faced limitations in our ability to formally verify the mathematical correctness of the GammaSwap contracts. Consequently, we were unable to thoroughly assess the mathematical aspects of borrowing, rebalancing, and liquidation, as well as the overall mathematical accuracy of the GammaSwap contracts. To address this, we highly recommend allocating additional time for comprehensive reviews that focus on these mathematical aspects to ensure the contracts' correctness and reliability.

## 1.3 Results

During our assessment on the scoped GammaSwap contracts, we discovered one finding. No critical issues were found. It was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for GammaSwap Protocol'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 GammaSwap

GammaSwap offers vaults to hedge impermanent loss (IL) in various AMMs and LP pools. The vaults can also be used to build novel structured products and other types of automated strategies.

## 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:

### GammaSwap Contracts

**Repositories** https://github.com/gammaswap/v1-implementations

https://github.com/gammaswap/v1-core

**Versions** v1-core: `4b4feb4a4615b0402117b879891ebd7930023ee1`

v1-implementations: `b84d9240d7ed435d4685453fc70248343e8da8d7`

**Programs**

- BaseStrategy.sol
- ShortStrategySync.sol
- LongStrategy.sol
- ShortStrategyERC4626.sol
- BaseLongStrategy.sol
- ShortStrategy.sol
- LiquidationStrategy.sol
- ExternalBaseStrategy.sol
- ExternalLongStrategy.sol

- ExternalLiquidationStrategy.sol
- TwoStepOwnable.sol
- LogDerivativeRateModel.sol
- AbstractRateParamsStore.sol
- CPMMBaseStrategy.sol
- CPMMShortStrategy.sol
- CPMMLongStrategy.sol
- CPMMBaseLongStrategy.sol
- CPMMLiquidationStrategy.sol
- CPMMExternalLongStrategy.sol
- CPMMExternalLiquidationStrategy.sol
- CPMMGammaPool.sol

**Type** Solidity

**Platform** EVM-compatible

## 2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two 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:

**Ayaz Mammadov**, Engineer
ayaz@zellic.io

**Vlad Toie**, Engineer
vlad@zellic.io

## 2.5 Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **May 15, 2023** | Start of primary review period |
| **May 16, 2023** | Kick-off call |
| **May 24, 2023** | End of primary review period |

# 3 Detailed Findings

## 3.1 Some array lengths are not checked

- **Target**: Codebase
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Low

### Description

There are instances in the codebase where array lengths are not properly checked, which can result in unexpected behavior like out-of-bounds errors.

Some examples include the following:

- In the `_depositReserves` function, the lengths of the `amountsDesired` and `amountsMin` arrays should be checked.

```
function _depositReserves(address to, uint256[] calldata amountsDesired,
    uint256[] calldata amountsMin, bytes calldata data)
    external virtual override lock returns(uint256[] memory reserves,
    uint256 shares) {
    address payee; // address that will receive reserve tokens from
    depositor

    // Calculate amounts of reserve tokens to send and address to send
    them to
    (reserves, payee) = calcDepositAmounts(amountsDesired, amountsMin);
    // ...
}
```

- The length of the `ratio` array in the `_borrowLiquidity` function should ideally be checked against the expected number of reserves, which is typically two.

```
function _borrowLiquidity(uint256 tokenId, uint256 lpTokens, uint256[]
    calldata ratio)
    external virtual override lock returns(uint256 liquidityBorrowed,
    uint256[] memory amounts) {
    // Revert if borrowing all CFMM LP tokens in pool
```

```
    if(lpTokens ≥ s.LP_TOKEN_BALANCE) {
        revert ExcessiveBorrowing();
    }

    // Get loan for tokenId, revert if not loan creator
    LibStorage.Loan storage _loan = _getLoan(tokenId);

    // Update liquidity debt to include accrued interest since last update
    uint256 loanLiquidity = updateLoan(_loan);

    // Withdraw reserve tokens from CFMM that lpTokens represent
    amounts = withdrawFromCFMM(s.cfmm, address(this), lpTokens);

    // Add withdrawn tokens as part of loan collateral
    (uint128[] memory tokensHeld,) = updateCollateral(_loan);

    // Add liquidity debt to total pool debt and start tracking loan
    (liquidityBorrowed, loanLiquidity) = openLoan(_loan, lpTokens);

    if(ratio.length > 0) {
        //
    }
    // ...
}
```

- In the `_rebalanceCollateral` function, the `deltas` and `ratios` arrays lengths should be checked.

```
function _rebalanceCollateral(uint256 tokenId, int256[] memory deltas,
    uint256[]
    calldata ratio)external virtual override lock returns(uint128[]
    memory tokensHeld) {

        // Get loan for tokenId, revert if not loan creator
        LibStorage.Loan storage _loan = _getLoan(tokenId);

        // Update liquidity debt to include accrued interest since last
    update
        uint256 loanLiquidity = updateLoan(_loan);

        if(ratio.length > 0) {
```

```
            deltas = _calcDeltasForRatio(_loan.tokensHeld,
    s.CFMM_RESERVES, ratio);
        }
    // ...
}
```

## Impact

The impact of this issue is low, as it is not possible to exploit it to steal funds or cause a denial of service.

## Recommendations

We recommend explicitly checking the length of the arrays in the functions where they are passed as arguments.

```
function _depositReserves(address to, uint256[] calldata amountsDesired,
    uint256[] calldata amountsMin, bytes calldata data)
    external virtual override lock returns(uint256[] memory reserves,
    uint256 shares) {

    require(amountsDesired.length == amountsMin.length, "LENGTH_MISMATCH");

    address payee; // address that will receive reserve tokens from
    depositor

    // Calculate amounts of reserve tokens to send and address to send
    them to
    (reserves, payee) = calcDepositAmounts(amountsDesired, amountsMin);
    // ...
}
```

## Remediation

This issue has been acknowledged by GammaSwap Protocol, and a fix was implemented in commit 64f6f5f2.

# 4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

## 4.1 Codebase documentation

Zellic commends GammaSwap for their exceptional documentation efforts. The supplied documentation, consisting of over 15 pages, provided comprehensive insights into the protocol's mechanics, design choices, and intricate details, greatly facilitating the auditing process.

In addition, it is worth mentioning that the in-code comments were extensively utilized throughout the codebase to enhance code comprehension. These comments effectively explained mathematical equations and outlined any compromises that were made during the development process, further enhancing the overall clarity and understanding of the codebase.

# 5 Threat Model

This provides a full 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: AbstractRateParamsStore.sol

### Function: `setRateParams(address _pool, byte[] data, bool active)`

Function used to set the rate parameters for a given pool (`_pool`). The parameters are passed as a byte array (`data`), and the rate model is activated or deactivated (`active`).

#### Inputs

- `_pool`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked in `validateParameters`.
  - **Impact**: The pool for which the rate parameters are set.
- `data`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked in `validateParameters`.
  - **Impact**: The rate parameters.
- `active`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Whether the rate model is active or not.

#### Branches and code coverage (including function calls)

**Intended branches**

- Assumed this function can be called both for setting and for updating the rate parameters.
  ☑ Test coverage
- Update the state of the `rateParams` variable to the `data` and `active` values.

☑ Test coverage

**Negative behavior**

- Should only be callable by the owner of the rate parameters store contract.
  ☑ Negative test
- Should only be callable if the parameters are valid.
  ☑ Negative test

### Function call analysis

- `IRateModel(_pool).validateParameters(data)`
  - **What is controllable?** `_pool` and `data` — basically an arbitrary call to an arbitrary address, but the caller should be the owner of the rate parameters store contract.
  - **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?** If it reverts, try catch will return false and the the call will revert.

## 5.2 Module: BalancerBaseLongStrategy.sol

### Function: `calcInAndOutAmounts(uint128 reserves0, uint128 reserves1, int256 deltas0, int256 deltas1)`

Calculates the expected bought and sold amounts corresponding to a change in collateral given by delta.

### Inputs

- `reserve0`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The amount of reserve0 tokens in the Balancer pool.
- `reserve1`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The amount of reserve1 tokens in the Balancer pool.
- `deltas0`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The desired amount of collateral tokens from the loan to swap (>

0 buy, < 0 sell, 0 ignore).

- `deltas1`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The desired amount of collateral tokens from the loan to swap (> 0 buy, < 0 sell, 0 ignore).

### Branches and code coverage (including function calls)

**Intended branches**

- Calculates amounts according to the Constant product equations.
  ☑ Test coverage

### Function call analysis

- `calcInAndOutAmounts` → `getAmountOut` → `getSwapFeePercentage(s.cfmm)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Swap fee.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

### Function: `function swapTokens(LibStorage.Loan storage, uint256[] memory outAmts, uint256[] memory inAmts)`

Swap using the balancer pool.

### Inputs

- `OutAmts`
  - **Control**: Full.
  - **Constraints**: Cannot be == 0 if `InAmts`[0] == 0.
  - **Impact**: Out amounts.
- `inAmts`
  - **Control**: Full
  - **Constraints**: Cannot be == 0 if `OutAmts`[0] == 0
  - **Impact**: In amounts.

### Branches and code coverage (including function calls)

**Intended branches**

- Swap works as expected.
  ☑ Test coverage

### Function call analysis

- `swapTokens → getVault()`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** The vault.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `swapTokens → Vault.swap( ... )`
  - **What is controllable?** Assets in/out, amounts in/out.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

## 5.3 Module: BalancerBaseStrategy.sol

### Function: `depositToCFMM(address _cfmm, address to, uint256[] memory amounts)`

Deposit into the balancer pool.

### Inputs

- `_cfmm`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The GammaPool to measure the difference of balances.
- `to`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The recipient of the balancer tokens.
- `amounts`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Slippage control.

## Branches and code coverage (including function calls)

### Intended branches

- Deposits the tokens into the balancer pool and receives LP tokens.
  ☑ Test Coverage

### Negative behavior

- Removes approval after usage.
  ☐ Negative test

## Function call analysis

- `depositToCFMM` → `addVaultApproval`
  - **What is controllable?** `_cfmm`.
  - **If return value controllable, how is it used and how can it go wrong?** Used to determine actual amounts sent to account for tokens with fees, and so forth.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `depositToCFMM` → `GammaSwapLibrary.balanceOf(_cfmm, address(this))`
  - **What is controllable?** `_cfmm`.
  - **If return value controllable, how is it used and how can it go wrong?** Used to determine actual amounts sent to account for tokens with fees, and so forth.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `depositToCFMM` → `getVault()`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Get the vault.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `depositToCFMM` → `joinPool( ... )`
  - **What is controllable?** `amounts`.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

### Function: `withdrawFromCFMM(address, address to, uint256 amount)`

Withdraw from the balancer pool.

#### Inputs

- `to`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Recipient of the underlying pool tokens.
- `amount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount to exit the balancer pool.

#### Branches and code coverage (including function calls)

**Intended branches**

- Exited pool position and received LP Tokens.
  ☑ Test Coverage

**Negative behavior**

- Cannot withdraw without the relevant tokens.
  ☐ Negative test
- Cannot withdraw 0 amount.
  ☐ Negative test

#### Function call analysis

- `withdrawFromCFMM → getStrategyReerves → GammaSwapLibrary.balancerOf(s.tokens[0/1], this)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Log differences.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `withdrawFromCFMM → getVault()`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** The vault.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

N/A.

- `withdrawFromCFMM` → `exitPool( ... , payable(to), ... , userData: abi.encode( 1, amount), ... )`
  - **What is controllable?** `to` address.
  - **If return value controllable, how is it used and how can it go wrong?** The vault.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

## 5.4 Module: CPMMBaseLongStrategy.sol

### Function: `calcInAndOutAmounts(LibStorage.Loan storage _loan, uint256 reserve0, uint256 reserve1, int256 delta0, int256 delta1)`

Is called before token swaps; it executes tokens swaps and measures the actual amount transferred this way, accounting for fees and other nuances

#### Inputs

- `_loan`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: The loan.
- `reserve0`
  - **Control**: Some. (If the call does not call any locked functions in the underlying pool, then flash loans can be used to call the function in a reentrant context / position manager should stop pool manipulation.)
  - **Constraints**: None.
  - **Impact**: Reserve of token0 of the underlying CPMM.
- `reserve1`
  - **Control**: Some. (If the call does not call any locked functions in the underlying pool, then flash loans can be used to call the function in a reentrant context / position manager should stop pool manipulation.)
  - **Constraints**: None.
  - **Impact**: Reserve of token1 of the underlying CPMM.
- `delta0`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The delta for this swap for token 0 (buy = positive/sell = negative).
- `delta1`

- **Control**: Full.
- **Constraints**: None.
- **Impact**: The delta for this swap for token 1 (buy = positive/sell = negative).

### Branches and code coverage (including function calls)

**Negative behavior**

- Does accounting work with tokens that encourage transfers using encouragement schemes (discount / get more tokens back)?
  ☐ Negative test

### Function call analysis

- `calcInAndOutAmounts → calcActualOutAmt → GammaSwapLibrary.balanceOf(token, s.cfmm)`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** The balances of the swap used before and after transfers to determine the real amount of tokens transferred, accounting for side effects like fees.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If reentrancy was involved, one could withdraw extra money sent for swaps before the swap is called; however, this should be protected by the position manager.
- `calcInAndOutAmounts → calcActualOutAmt → sendToken → sendToken(token, to, amount, balance, collateral)`
  - **What is controllable?** Nothing by amounts as a side effect of deltas.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If reentrancy was involved, one could withdraw extra money sent for swaps before the swap is called; however, this should be protected by the position manager.

## 5.5 Module: CPMMLongStrategy.sol

### Function: `_calcDeltasForRatio(uint128[] memory tokensHeld, uint128[] memory reserves, uint256[] calldata ratio)`

Calculates the deltas required to move the ratio of a CPMM dex. It does this by just moving around the variables in the constant product equation and then using these

to solve a quadratic equation.

#### Inputs

- `tokensHeld`
  - **Control**: Can add more to loan (cannot be modified arbitrarily).
  - **Constraints**: None.
  - **Impact**: The current `tokensHeld` by the loan.
- `reserves`
  - **Control**: Some. (If the call does not call any locked functions in an underlying pool, then flash loans can be used to call the function in a reentrant context / position manager should stop pool manipulation.)
  - **Constraints**: None.
  - **Impact**: Reserves of the underlying CPMM.
- `ratio`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Desired ratio.

#### Branches and code coverage (including function calls)

**Negative behavior**

- Checks for impossible values like 0 or a reserve token amount of 0.
  - ☐ Negative test

### 5.6 Module: CPMMShortStrategy.sol

#### Function: `calcDepositAmounts(uint256[] calldata amountsDesired, uint256[] calldata amountsMin)`

This just makes sure that deposits are in ratio as to not waste any extra tokens and enforces a certain minimum amount.

#### Inputs

- `amountsDesired`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amounts desired.
- `amountsMin`

- **Control**: Full.
- **Constraints**: None.
- **Impact**: Minimum acceptable deposit amounts.

## 5.7 Module: ExternalLongStrategy.sol

### Function: `_rebalanceExternally(uint256 tokenId, uint128[] amounts, uint256 lpTokens, address to, byte[] data)`

Flash loan pool's collateral and/or LP tokens to external address. Rebalanced loan collateral is acceptable in repayment of flash loan.

#### Inputs

- `tokenId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked to be a valid loan and that caller is loan creator.
  - **Impact**: The loan to be rebalanced.
- `amounts`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Collateral amounts to be rebalanced.
- `lpTokens`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Loan's CFMM LP tokens to be rebalanced.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Destination address must be a valid address.
  - **Impact**: Address that will receive flash loan swaps and potentially rebalance loan's collateral.
- `data`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Optional bytes parameter for custom user-defined data.

#### Branches and code coverage (including function calls)

**Intended branches**

- Should update the `s.lastCFMMIndex` to the newly calculated value.
  ☑ Test coverage
- Should update the `s.LP_TOKEN_BORROWED_PLUS_INTEREST` to the newly calculated value.
  ☑ Test coverage
- Should update the `s.LP_INVARIANT` to the newly calculated value.
  ☑ Test coverage
- Should update the `s.BORROWED_INVARIANT` to the newly calculated value.
  ☑ Test coverage
- Should update the `loan.rateIndex` to the newly calculated value.
  ☑ Test coverage
- Should update the `loan.liquidity` to the newly calculated value.
  ☑ Test coverage
- Send collateral of LP to `to` address if required.
  ☑ Test coverage
- Send CFMM LP to `to` address if required.
  ☑ Test coverage
- Perform callback on `to` and thus assume that during this callback, the funds are returned to the pool.
  ☑ Test coverage

**Negative behavior**

- Should not be callable by anyone than the loan creator.
  ☑ Negative test
- Should not allow funds to be below values previous to the flash loan at the end of the function.
  ☑ Negative test
- Assure loan is overcollateralized after external swap.
  ☑ Negative test

### Function call analysis

- `IExternalCallee(to).externalCall(msg.sender, amounts, lpTokens, data);`
  - **What is controllable?** `to` and `lpTokens`.
  - **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?** Should perform a callback on `to`, and it should result in the funds being returned to the pool.

## 5.8 Module: LiquidationStrategy.sol

### Function: `function _liquidate(uint256 tokenId, int256[] calldata deltas , uint256[] calldata fees)`

Liquidate an underwater position.

#### Inputs

- `tokenId`
  - **Control**: Full.
  - **Constraints**: Must be an existent loan.
  - **Impact**: The loan to liquidate.
- `deltas`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The swaps to make for the tokens.
- `fees`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The fees.

#### Branches and code coverage (including function calls)

**Intended branches**

- Liquidation works as intended, and liquidators are paid.
  ☑ Test Coverage
- Liquidators are refunded if they overpay.
  ☑ Test Coverage

**Negative behavior**

- Zero liquidations should fail.
  ☐ Negative test
- Liquidations without enough tokens.
  ☐ Negative test

#### Function call analysis

- `_liquidate` → `payLoanAndRefundLiquidator` → `GammaSwapLibrary.balanceOf`
  - **What is controllable?** Nothing.

  - **If return value controllable, how is it used and how can it go wrong?** Pre- / post-transfer balance.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `_liquidate` → `payLoanAndRefundLiquidator` → `refundOverPayment` → `GammaSwapLibrary.safeTransfer`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

## 5.9 Module: LongStrategy.sol

### Function: `_borrowLiquidity(uint256 tokenId, uint256 lpTokens, uint256[] ratio)`

Allow borrowing liquidity from loan identified by `tokenId`; should revert if loan is undercollateralized after withdrawal.

#### Inputs

- `tokenId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that the loan is a valid one and that the `msg.sender` is the owner of the loan.
  - **Impact**: The loan on which the liquidity is borrowed.
- `lpTokens`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it is greater than current `s.LP_TOKEN_BALANCE`.
  - **Impact**: The amount of liquidity to be borrowed.
- `ratio`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The weight of collateral to be added to the loan.

#### Branches and code coverage (including function calls)

**Intended branches**

- Should check that `ratio` length == `s.CFMM_RESERVES` length. Currently not explic-

itly checked.
  ☐ Test coverage
- Assumes the loan is overcollateralized before calling this function.
  ☑ Test coverage
- Update the `loan.liquidity` before the withdrawal.
  ☑ Test coverage
- Update the `loan.rateIndex` before the withdrawal.
  ☑ Test coverage
- Increase the `s.TOKEN_BALANCE` by the amount of tokens borrowed.
  ☑ Test coverage
- Increase the `loan.tokensHeld` by the amount of tokens borrowed.
  ☑ Test coverage
- Update the `s.BORROWED_INVARIANT` after the withdrawal.
  ☑ Test coverage
- Update the `s.LP_INVARIANT` after the withdrawal.
  ☑ Test coverage
- Update the `s.LP_TOKEN_BORROWED` after the withdrawal.
  ☑ Test coverage
- Update the `s.LP_TOKEN_BORROWED_PLUS_INTEREST` after the withdrawal.
  ☑ Test coverage
- Update the `loan.px` after the withdrawal.
  ☑ Test coverage
- Update the `loan.initLiquidity` after the withdrawal.
  ☑ Test coverage
- Update the `loan.lpTokens` after the withdrawal.
  ☑ Test coverage
- Update the `loan.liquidity` after the withdrawal.
  ☑ Test coverage
- If ratio is not empty, rebalance and update the `loan.tokensHeld` and `s.TOKEN_BALANCE` after the swap according to the new balances.
  ☑ Test coverage
- Assure that the loan is not undercollateralized after the liquidity is borrowed. This is ensured by the `checkMargin` function.
  ☑ Test coverage

**Negative behavior**

- Assure that the loan cannot borrow multiple times. Currently not explicitly checked.
  ☐ Negative test
- Check that no one other than the owner of the loan can borrow liquidity. This is ensured by the `getLoan` function.

☑ Negative test

### Function call analysis

- `withdrawFromCFMM(s.cfmm, address(this), lpTokens);`
  - **What is controllable?** N/A.
  - **If return value controllable, how is it used and how can it go wrong?** Amount of CFMM tokens that have been withdrawn.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts if the the amount of LPs to be withdrawn is greater than what the pool has.
- `GammaSwapLibrary.balanceOf(tokens[i], address(this))`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the balance of the specific token; this should be influenced by the caller and assumed greater than the previous `s.TOKEN_BALANCE[i]`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts if the token is not a valid ERC20 token.

## Function: `_decreaseCollateral(uint256 tokenId, uint128[] amounts, address to)`

Allow withdrawing collateral from loan identified by `tokenId`; should revert if loan is undercollateralized after withdrawal.

### Inputs

- `tokenId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumes `tokenId` is a valid loan and that `msg.sender` is the owner of the loan.
  - **Impact**: The loan from which the collateral is removed.
- `amounts`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumes that the amounts are valid and that the amounts are less than the current balance of the contract as well as the current balance of the loan.
  - **Impact**: The amount of tokens to be removed from the loan.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumes that the address is valid.

- **Impact**: The address to which the tokens are transferred.

### Branches and code coverage (including function calls)

#### Intended branches

- ☐ Delete the entire loan after the withdrawal if the loan is fully withdrawn. This would mean that all debt has been repaid and all collateral has been withdrawn.
- ☑ Decrease the `loan.tokensHeld` by the amount of tokens withdrawn.
- ☑ Update the `loan.liquidity` before the withdrawal.
- ☑ Update the `loan.rateIndex` before the withdrawal.
- ☑ Decrease the `s.TOKEN_BALANCE` by the amount of tokens withdrawn.
- ☑ Check that the new invariant and loan liquidity amounts do not lead to under-collateralization. Enforced through `checkMargin` function.

#### Negative behavior

- ☐ Should not allow withdrawing collateral from a terminated loan. Currently not explicitly checked.
- ☐ In `sendTokens`, should not allow multiple entries of same `token`, so that the `loan.tokensHeld` is not double counted.
- ☑ Should not allow withdrawing collateral from a loan that is undercollateralized. This is ensured by the `checkMargin` function.
- ☑ Should not allow anyone other than the owner of the loan to withdraw collateral. This is ensured by the `getLoan` function.
- ☑ Should not allow withdrawing more collateral than the loan currently has. This is ensured by the `withdrawCollateral` function.

### Function call analysis

- `GammaSwapLibrary.safeTransfer(token, to, amount);`
  - **What is controllable?** `to` and `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 if the token is not a valid ERC20 token or if user wanted to send more than the current balance of the contract.

### Function: `_increaseCollateral(uint256 tokenId)`

Account for deposited tokens in the loan. Assumes transfer happened before calling this function atomically.

### Inputs

- `tokenId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumes `tokenId` is a valid loan and that `msg.sender` is the owner of the loan.
  - **Impact**: The loan to which the collateral is added.

### Branches and code coverage (including function calls)

#### Intended branches

- Assumes that the tokens were already transferred to the contract.
  ☐ Test coverage
- Assumes that the new balance is greater than the previous balance. If not, it should revert, since no tokens were deposited.
  ☐ Test coverage
- Should increase the `loan.tokensHeld` by the amount of tokens deposited.
  ☑ Test coverage
- Assumes that no double accounting occurs, and only the difference between the previously accounted tokens and the new amount is added to the loan.
  ☑ Test coverage
- Assumes that the tokens were already transferred to the contract.
  ☑ Test coverage
- Should incerase the `s.LP_TOKEN_BALANCE` by the amount of tokens deposited.
  ☑ Test coverage

#### Negative behavior

- Should not allow increasing collateral to a terminated loan. Currently not explicitly checked.
  ☐ Negative test
- Should not be callable by anyone other than the owner of the loan. This is ensured by the `getLoan` function.
  ☑ Negative test

### Function call analysis

- `GammaSwapLibrary.balanceOf(tokens[i], address(this))`
  - **What is controllable?** Nothing.
  - **If return value controllable, how is it used and how can it go wrong?** Returns the balance of the specific token; this should be influenced by the caller and assumed greater than the previous `s.TOKEN_BALANCE[i]`.

- **What happens if it reverts, reenters, or does other unusual control flow?** Reverts if the token is not a valid ERC20 token.

### Function: `_rebalanceCollateral(uint256 tokenId, int256[] deltas, uint256[] ratio)`

Allow rebalancing the collateral (i.e., swapping one token for another, from the CFMM pair of reserve tokens) of a loan.

#### Inputs

- `tokenId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Check that loan exists and that `msg.sender` is the loan creator.
  - **Impact**: The loan's collateral will be rebalanced.
- `deltas`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should check at least one of the deltas is non-zero; currently not implemented.
  - **Impact**: The loan's collateral will be rebalanced by the deltas.
- `ratio`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Should check that the ratio is between 0 and 1; currently not implemented.
  - **Impact**: The loan's collateral will be rebalanced by the ratio.

#### Branches and code coverage (including function calls)

**Intended branches**

- Assure that arrays' lengths match (`deltas` and `ratio`). Currently not implemented.
  - ☐ Test coverage
- Should check that the ratio is between 0 and 1. Currently not implemented.
  - ☐ Test coverage
- Should update the `s.lastCFMMIndex` to the newly calculated value.
  - ☑ Test coverage
- Should update the `s.LP_TOKEN_BORROWED_PLUS_INTEREST` to the newly calculated value.
  - ☑ Test coverage
- Should update the `s.LP_INVARIANT` to the newly calculated value.
  - ☑ Test coverage

- Should update the `s.BORROWED_INVARIANT` to the newly calculated value.
  ☑ Test coverage
- Should update the `loan.rateIndex` to the newly calculated value.
  ☑ Test coverage
- Should update the `loan.liquidity` to the newly calculated value.
  ☑ Test coverage
- Check that the new invariant and loan liquidity amounts do not lead to under-collateralization. Enforced through `checkMargin` function.
  ☑ Test coverage

**Negative behaviour**

- Should not be callable by anyone other than the loan creator. Enforced through `getLoan` function.
  ☑ Negative test

### Function call analysis

- `swapTokens(_loan, outAmts, inAmts);`
  - **What is controllable?** `_loan`.
  - **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?** This is supposed to be the swap within `rebalanceCollateral`, so it should not revert.

### Function: `_repayLiquidity(uint256 tokenId, uint256 payLiquidity, uint256[] fees, uint256 collateralId, address to)`

Allows repaying liquidity debt of loan identified by `tokenId`; debt is repaid using available collateral in loan.

### Inputs

- `tokenId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumes `tokenId` is a valid loan and that `msg.sender` is the owner of the loan.
  - **Impact**: The loan from which the liquidity is repaid.
- `payLiquidity`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumes that the amount is valid and that the amount is less or equal to the current liquidity debt of the loan.

  - **Impact**: The amount of liquidity to be repaid.
- `fees`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A; assumes that the amount is valid.
  - **Impact**: The amount of fees to be paid.
- `collateralId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A; assumes that the ID is valid.
  - **Impact**: The index of the `tokensHeld` array to rebalance to.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumes that the address is valid.
  - **Impact**: The address to which the tokens are transferred.

### Branches and code coverage (including function calls)

**Intended branches**

- Delete the entire loan if paid in full and no more collateral is left. Currently not implemented.
  ☐ Test coverage
- Assumes someone can just not specify 0 for the fees in case of an atomically constructed transaction.
  ☐ Test coverage
- Update the `loan.rateIndex` before and after repayment.
  ☑ Test coverage
- If `collateralId` is greater than 0, rebalance the collateral to the desired ratio.
  ☑ Test coverage
- If collateral is rebalanced, update the `loan.tokensHeld` before and after repayment.
  ☑ Test coverage
- If collateral is rebalanced, update the `s.TOKEN_BALANCE` before and after repayment.
  ☑ Test coverage
- Update the `loan.liquidity` before and after repayment.
  ☑ Test coverage
- Update the `s.LP_TOKEN_BORROWED_PLUS_INTEREST` before and after the repayment, to reflect the paid debt.
  ☑ Test coverage
- Update the `s.LP_TOKEN_BORROWED` after the repayment to reflect the paid debt.

☑ Test coverage

- Update the `s.LP_INVARIANT` before and after the repayment.
  ☑ Test coverage
- Update the `loan.liquidity` after repayment.
  ☑ Test coverage
- Update the `loan.lpTokens` after repayment.
  ☑ Test coverage
- Update the `loan.initLiquidity` after repayment.
  ☑ Test coverage
- Update the `loan.rateIndex` after repayment, if paid in full.
  ☑ Test coverage

**Negative behavior**

- Should not allow repaying liquidity to a terminated loan. Currently not explicitly checked.
  ☐ Negative test
- In `sendTokens`, should not allow multiple entries of same `token`, so that the `loan.tokensHeld` is not double counted.
  ☐ Negative test
- Should not allow anyone other than the owner of the loan to repay liquidity. This is ensured by the `getLoan` function.
  ☑ Negative test

### Function call analysis

- `swapTokens(_loan, outAmts, inAmts);`
  - **What is controllable?** `_loan`.
  - **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?** This is supposed to be the swap within `rebalanceCollateral`, so it should not revert.
- `depositToCFMM(s.cfmm, address(this), amounts);`
  - **What is controllable?** N/A.
  - **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?** This call happens in `repayTokens` and is responsible for depositing the collateral to the CFMM, for CFMM LPs.
- `GammaSwapLibrary.safeTransfer(token, to, amount);`
  - **What is controllable?** `to` and partly `amount`, since it depends on the `loan.tokensHeld` and `loan.lpTokens`.

  - **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?** This call happens in `withdrawCollateral` and is responsible for sending tokens to `to`.

### Function: `_updatePool(uint256 tokenId)`

Allow caller to update the pool/loan debt state.

### Inputs

- `tokenId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it is a valid loan, but not that it is owned by the caller.
  - **Impact**: The ID of the loan.

### Branches and code coverage (including function calls)

**Intended branches**

- Should update the `s.lastCFMMIndex` to the newly calculated value.
  ☑ Test coverage
- Should update the `s.LP_TOKEN_BORROWED_PLUS_INTEREST` to the newly calculated value.
  ☑ Test coverage
- Should update the `s.LP_INVARIANT` to the newly calculated value.
  ☑ Test coverage
- Should update the `s.BORROWED_INVARIANT` to the newly calculated value.
  ☑ Test coverage
- Should update the `loan.rateIndex` to the newly calculated value.
  ☑ Test coverage
- Should update the `loan.liquidity` to the newly calculated value.
  ☑ Test coverage

**Negative behavior**

- Should not be callable by anyone other than the owner of the loan. Currently not enforced, since `_getExistingLoan` does not perform the check.
  ☐ Negative test

## 5.10 Module: ShortStrategyERC4626.sol

### Function: `_deposit(uint256 assets, address to)`

Allow user to deposit CFMM LP tokens into the strategy and receive GS LP tokens in return.

### Inputs

- `assets`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must be greater than 0.
  - **Impact**: The amount of CFMM LP tokens to deposit.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to receive the GS LP tokens.

### Branches and code coverage (including function calls)

**Intended branches**

- Transfer CFMM LP tokens from caller to GammaPool.
  ☑ Test coverage
- Mint GS LP tokens to receiver.
  ☑ Test coverage
- Update `s.lastCFMMIndex` with the current CFMM index before deposit.
  ☑ Test coverage
- Update `s.LP_INVARIANT` with the current CFMM invariant before deposit.
  ☑ Test coverage
- Update `s.LP_TOKEN_BORROWED_PLUS_INTEREST` with the current CFMM LP tokens borrowed plus interest before deposit.
  ☑ Test coverage

**Negative behavior**

- Should not allow depositing 0 CFMM LP tokens.
  ☑ Negative test
- Should not allow sending funds that are not CFMM LP tokens.
  ☑ Negative test
- Should not allow sending less/more funds than indicated by `assets`.
  ☑ Negative test

### Function call analysis

- `GammaSwapLibrary.safeTransferFrom(s.cfmm, caller, address(this), assets);`
  - **What is controllable?** `caller` and `assets`.
  - **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?** Means caller does not have enough funds to cover deposit.

### Function: `_mint(uint256 shares, address to)`

Allow user to mint GS LP tokens by depositing CFMM LP tokens into the strategy.

### Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: `convertToAssets(shares) > 0`.
  - **Impact**: The amount of GS LP tokens to mint.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address to receive the GS LP tokens.

### Branches and code coverage (including function calls)

**Intended branches**

- Convert GS LP tokens to CFMM LP tokens.
  ☑ Test coverage
- Transfer CFMM LP tokens from caller to GammaPool.
  ☑ Test coverage
- Mint GS LP tokens to receiver.
  ☑ Test coverage
- Update `s.lastCFMMIndex` with the current CFMM index before deposit.
  ☑ Test coverage
- Update `s.LP_INVARIANT` with the current CFMM invariant before deposit.
  ☑ Test coverage
- Update `s.LP_TOKEN_BORROWED_PLUS_INTEREST` with the current CFMM LP tokens borrowed plus interest before deposit.
  ☑ Test coverage

**Negative behavior**

- Should not allow minting 0 GS LP tokens.
  ☑ Negative test
- Should not allow sending funds that are not CFMM LP tokens.
  ☑ Negative test
- Should not allow sending less/more funds than indicated by `assets`.
  ☑ Negative test

### Function call analysis

- `GammaSwapLibrary.safeTransferFrom(s.cfmm, caller, address(this), assets);`
  - **What is controllable?** `caller` and `assets`.
  - **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?** Means caller does not have enough funds to cover mint.

## Function: `_redeem(uint256 shares, address to, address from)`

Redeem GS LP tokens from the strategy by burning GS LP tokens.

### Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount of GS LP tokens to redeem.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The address to receive the CFMM LP tokens.
- `from`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumed it is either `msg.sender` OR that it has given allowance to the strategy to `msg.sender` to burn its GS LP tokens,
  - **Impact**: The owner of the GS LP tokens to burn

### Branches and code coverage (including function calls)

#### Intended branches

- Assumes the advantage of using withdraw over redeem is minimal, in terms of rounding issues when converting between assets and shares.
  ☐ Test coverage

- Withdraw CFMM LP tokens from GammaPool.
  - ☑ Test coverage
- Burn necessary amount of GS LP tokens from `from`.
  - ☑ Test coverage
- Assure that `from` has enough GS LP tokens to burn.
  - ☑ Test coverage
- Update `s.lastCFMMIndex` with the current CFMM index before withdraw.
  - ☑ Test coverage
- Update `s.LP_INVARIANT` with the current CFMM invariant before and after withdraw.
  - ☑ Test coverage
- Update `s.LP_TOKEN_BORROWED_PLUS_INTEREST` with the current CFMM LP tokens borrowed plus interest before withdraw.
  - ☑ Test coverage
- Update the `s.LP_TOKEN_BALANCE` with the current CFMM LP tokens balance after withdraw.
  - ☑ Test coverage

**Negative behavior**

- Assure that `shares` is greater than 0. Curerntly not explicitly checked.
  - ☐ Negative test
- Should not allow burning 0 GS LP tokens.
  - ☐ Negative test
- Assure that `msg.sender` is either `from` or that it has given allowance to the strategy to `from` to burn its GS LP tokens. This is taken care of in `withdrawAssets`.
  - ☑ Negative test

## Function call analysis

- `GammaSwapLibrary.safeTransfer(cfmm, to, assets);`
  - **What is controllable?** `to` and `assets` partly.
  - **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?** Means GammaPool does not have enough funds to cover withdraw.

### Function: `_withdraw(uint256 assets, address to, address from)`

Withdraw CFMM LP tokens from the strategy by burning GS LP tokens.

### Inputs

- `assets`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Checked that it is greater than `s.LP_TOKEN_BALANCE`.
  - **Impact**: The amount of assets to withdraw.
- `to`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: The address to receive the CFMM LP tokens.
- `from`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Assumed it is either `msg.sender` OR that it has given allowance to the strategy to `msg.sender` to burn its GS LP tokens.
  - **Impact**: The owner of the GS LP tokens to burn.

### Branches and code coverage (including function calls)

#### Intended branches

- Assumes the advantage of using withdraw over redeem is minimal, in terms of rounding issues when converting between assets and shares.
  ☐ Test coverage
- Withdraw CFMM LP tokens from GammaPool.
  ☑ Test coverage
- Burn necessary amount of GS LP tokens from `from`.
  ☑ Test coverage
- Assure that `from` has enough GS LP tokens to burn.
  ☑ Test coverage
- Update `s.lastCFMMIndex` with the current CFMM index before withdraw.
  ☑ Test coverage
- Update `s.LP_INVARIANT` with the current CFMM invariant before and after withdraw.
  ☑ Test coverage
- Update `s.LP_TOKEN_BORROWED_PLUS_INTEREST` with the current CFMM LP tokens borrowed plus interest before withdraw.
  ☑ Test coverage
- Update the `s.LP_TOKEN_BALANCE` with the current CFMM LP tokens balance after withdraw.
  ☑ Test coverage

**Negative behavior**

- Assure that `msg.sender` is either `from` or that it has given allowance to the strategy to `from` to burn its GS LP tokens. This is taken care of in `withdrawAssets`.
  ☑ Negative test
- Should not allow burning 0 GS LP tokens.
  ☑ Negative test

### Function call analysis

- `GammaSwapLibrary.safeTransfer(cfmm, to, assets);`
  - **What is controllable?** `to` and `assets` partly.
  - **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?** Means GammaPool does not have enough funds to cover withdraw.

## 5.11 Module: ShortStrategySync.sol

### Function: `_sync()`

Synchronize `LP_TOKEN_BALANCE` with actual CFMM LP tokens deposited in GammaPool.

### Branches and code coverage (including function calls)

**Intended branches**

- Should update the `s.lastCFMMIndex` to the newly calculated value.
  ☑ Test coverage
- Should update the `s.LP_TOKEN_BORROWED_PLUS_INTEREST` to the newly calculated value.
  ☑ Test coverage
- Should update the `s.LP_INVARIANT` to the newly calculated value.
  ☑ Test coverage
- Should update the `s.BORROWED_INVARIANT` to the newly calculated value.
  ☑ Test coverage
- Should update the `s.LP_TOKEN_BALANCE` to the newly calculated value.
  ☑ Test coverage

**Negative behavior**

### Function call analysis

- `GammaSwapLibrary.balanceOf(s.cfmm, address(this))`
  - **What is controllable?** N/A.
  - **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?** Means contract is not configured properly.

## 5.12 Module: ShortStrategy.sol

### Function: `_depositNoPull(address to)`

Deposit CFMM LP tokens and get GS LP tokens without doing a `transferFrom` transaction from the user.

### Inputs

- `to`
  - **Control**: Full control over the address that receives the GS LP tokens.
  - **Constraints**: None.
  - **Impact**: The address that receives the GS LP tokens can be controlled by the attacker.

### Branches and code coverage (including function calls)

**Intended branches**

- Allow the deposit of CFMM LP tokens for GS LP tokens.
  ☑ Test coverage
- Update the `s.lastCFMMFeeIndex` with the new fee index.
  ☑ Test coverage
- Update `s.LP_TOKEN_BORROWED_PLUS_INTEREST` with the new amount of CFMM LP tokens.
  ☑ Test coverage
- Update `s.LP_TOKEN_BALANCE` with the deposited CFMM LP tokens.
  ☑ Test coverage
- Update `s.LP_INVARIANT` with the new invariant.
  ☑ Test coverage
- Mint GS LP tokens to the receiver; thus, the total supply of GS LP tokens increases.
  ☑ Test coverage

- Assumes it cannot be front-run, since theoretically users should perform this transaction atomically with the deposit in CFMM.
  ☑ Test coverage

**Negative behavior**

- Should not allow to deposit 0 CFMM LP tokens.
  ☑ Negative test
- Assumes the deposit has happened before this function call and that the accounting is correct.
  ☑ Negative test

### Function call analysis

- `GammaSwapLibrary.balanceOf(s.cfmm, address(this))`
  - **What is controllable?**: The amount of CFMM LP tokens that GammaPool has, since someone could send CFMM LP tokens to GammaPool.
  - **If return value controllable, how is it used and how can it go wrong?**: The amount of CFMM LP tokens is used to calculate the amount of GS LP tokens to mint.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The function reverts.

### Function: `_depositReserves(address to, uint256[] amountsDesired, uint256[] amountsMin, byte[] data)`

Deposit reserve token quantities to CFMM to get CFMM LP tokens, store them in GammaPool, and receive GS LP tokens.

### Inputs

- `to`
  - **Control**: Full control over the address that receives the GS LP tokens.
  - **Constraints**: None.
  - **Impact**: The address that receives the GS LP tokens can be controlled by the attacker
- `amountsDesired`
  - **Control**: Full control over the amounts of reserve tokens to deposit.
  - **Constraints**: None; assumes it matches the amounts of reserve tokens that were sent to GammaPool; should be explicitly checked.
  - **Impact**: Presumably, the amounts of reserve tokens that are deposited in the CFMM.

- `amountsMin`
  - **Control**: Full control over the minimum amounts of reserve tokens to deposit.
  - **Constraints**: None; should be explicitly checked.
  - **Impact**: The minimum amounts of reserve tokens that are deposited in the CFMM.
- `data`
  - **Control**: Full control over the data.
  - **Constraints**: None.
  - **Impact**: Included in the callback of the `sendTokensCallback` function.

### Branches and code coverage (including function calls)

#### Intended branches

- Should check that the `amountsDesired` and `amountsMin` have the same length.
  ☐ Test coverage
- Should check that the `amountsDesired` is the actual deposited amount. Currently not checked.
  ☐ Test coverage
- Mint GS LP tokens to the receiver; thus, the total supply of GS LP tokens increases.
  ☑ Test coverage
- Increase `s.LP_TOKEN_BALANCE` with the deposited CFMM LP tokens.
  ☑ Test coverage
- Increase `s.LP_INVARIANT` with the new invariant.
  ☑ Test coverage
- Updates `s.lastCFMMFeeIndex` with the new fee index.
  ☑ Test coverage
- Updates `s.LP_TOKEN_BORROWED_PLUS_INTEREST` with the new amount of CFMM LP tokens.
  ☑ Test coverage

#### Negative behavior

- Should not allow more reserve tokens to be deposited than the ones that have been sent to GammaPool.
  ☐ Negative test
- Assumes `msg.sender` has implemented the `ISendTokensCallback` interface; otherwise, should revert.
  ☑ Negative test

### Function call analysis

- `ISendTokensCallback(msg.sender).sendTokensCallback(tokens, amounts, to, data);`
  - **What is controllable?** `data` and `msg.sender`, also `amounts` partly.
  - **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?** Potential issue with this is that callback is given to `msg.sender`; assumed that during this callback the `msg.sender` is supposed to send the `amounts` to `to`(which is the address of a CFMM). Moreover, there are no explicit checks that the amount that the user initially indicated accounts for the previous balance of the CFMM and the new balance of the CFMM.
- `GammaSwapLibrary.balanceOf(tokens[i], to);`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** Returns balance of the specific token in the CFMM.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.

### Function: `_withdrawNoPull(address to)`

Withdraw CFMM LP tokens, by burning GS LP tokens, without doing a `transferFrom` transaction. Assumes the user has already sent GS LP tokens to GammaPool.

### Inputs

- `to`
  - **Control**: Full control over the address that receives the CFMM LP tokens.
  - **Constraints**: None.
  - **Impact**: The address that receives the CFMM LP tokens can be controlled by the attacker.

### Branches and code coverage (including function calls)

#### Intended branches

- Allows the withdrawal of CFMM LP tokens for GS LP tokens.
  ☑ Test coverage
- Assumes it cannot be front-run, since theoretically users should perform this transaction atomically with the deposit in GS LP tokens.
  ☑ Test coverage
- Updates `s.lastCFMMFeeIndex` with the new fee index.

☑ Test coverage

- Updates `s.LP_TOKEN_BORROWED_PLUS_INTEREST` with the new amount of CFMM LP tokens.
  ☑ Test coverage
- Updates `s.LP_INVARIANT` with the new invariant.
  ☑ Test coverage
- Updates `s.LP_TOKEN_BALANCE` with the deposited CFMM LP tokens.
  ☑ Test coverage
- Decreases the total supply of GS LP tokens and burns the GS LP tokens from the `address(this)` since the user has already sent them to GammaPool.
  ☑ Test coverage

**Negative behavior**

- Assumes there are no issues with fetching the balance of the CFMM LP tokens.
  ☑ Negative test
- Should not allow withdrawal of more CFMM LP tokens than the ones that GammaPool has.
  ☑ Negative test

### Function call analysis

- `s.balanceOf[address(this)]`
  - **What is controllable?** The amount of GS LP tokens that GammaPool has, since someone could send GS LP tokens to GammaPool.
  - **If return value controllable, how is it used and how can it go wrong?** The amount of GS LP tokens is used to calculate the amount of CFMM LP tokens to burn.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The function reverts.

### Function: `_withdrawReserves(address to)`

Withdraw reserve tokens from CFMM and send them to receiver address (`to`).

### Inputs

- `to`
  - **Control**: Full control over the address that receives the reserve tokens.
  - **Constraints**: None.
  - **Impact**: The address that receives the reserve tokens can be controlled by the attacker.

### Branches and code coverage (including function calls)

#### Intended branches

- Allow the withdrawal of reserve tokens for CFMM LP tokens.
  ☑ Test coverage
- Assumes it cannot be front-run, since theoretically users should perform this transaction atomically with the deposit in GS LP tokens.
  ☑ Test coverage
- Updates `s.lastCFMMFeeIndex` with the new fee index.
  ☑ Test coverage
- Updates `s.LP_TOKEN_BORROWED_PLUS_INTEREST` with the new amount of CFMM LP tokens.
  ☑ Test coverage
- Updates `s.LP_INVARIANT` with the new invariant.
  ☑ Test coverage
- Updates `s.LP_TOKEN_BALANCE` with the deposited CFMM LP tokens.
  ☑ Test coverage
- Updates `s.lastCFMMTotalSupply` with the new total supply of CFMM LP tokens.
  ☑ Test coverage
- Updates `s.lastCFMMInvariant` with the new invariant.
  ☑ Test coverage
- Decreases the total supply of GS LP tokens and burns the GS LP tokens from the `address(this)` since the user has already sent them to GammaPool.
  ☑ Test coverage
- Withdraw reserve tokens from CFMM and send them to receiver address (`to`).
  ☑ Test coverage
- Balance of reserve tokens in CFMM should be greater than the amount that is withdrawn.
  ☐ Test coverage
- Balance of reserve tokens in CFMM should decrease by the amount that is withdrawn.
  ☑ Test coverage
- Assure that the balance of reserves before conversion is less than the balance of reserves after conversion.
  ☐ Test coverage

#### Negative behavior

- Assumes there are no issues with fetching the balance of the CFMM LP tokens.
  ☑ Negative test
- Should revert if anything goes wrong with withdrawing from the CFMM.

☑ Negative test

- Should not allow withdrawal of more CFMM LP tokens than the ones that GammaPool has (which is the amount that was deposited).
  ☑ Negative test

### Function call analysis

- `s.balanceOf[address(this)];`
  - **What is controllable?** The amount of GS LP tokens that GammaPool has, since someone could send GS LP tokens to GammaPool.
  - **If return value controllable, how is it used and how can it go wrong?** The amount of GS LP tokens is used to calculate the amount of CFMM LP tokens to burn.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A.
- `GammaSwapLibrary.balanceOf(cfmm, address(this));`
  - **What is controllable?** The amount of CFMM LP tokens that GammaPool has, since someone could send CFMM LP tokens to GammaPool.
  - **If return value controllable, how is it used and how can it go wrong?** The amount of CFMM LP tokens is used to calculate the amount of GS LP tokens to burn.
  - **What happens if it reverts, reenters, or does other unusual control flow?** The function reverts.
- `withdrawFromCFMM(cfmm, to, assets);`
  - **What is controllable?** `assets` and `to`.
  - **If return value controllable, how is it used and how can it go wrong?** Amount of CFMM tokens that have been withdrawn.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts if the the amount of LPs to be withdrawn is greater than what the pool has.

## 5.13 Module: TwoStepOwnable.sol

### Function: `acceptOwnership()`

Function used to accept the ownership transfer of the contract. The pending owner must call this function to accept the ownership transfer.

### Branches and code coverage (including function calls)

**Intended branches**

- Update the state of the `owner` variable to the `pendingOwner` address.
  ☑ Test coverage
- Delete the `pendingOwner` variable.
  ☑ Test coverage

**Negative behavior**

- Should not be callable by anyone other than the pending owner.
  ☑ Negative test

### Function: `transferOwnership(address newOwner)`

Function used to transfer ownership of the contract to a new account (`newOwner`). Via a two-step method, the new owner must accept the ownership transfer.

### Inputs

- `newOwner`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: Must not be the zero address.
  - **Impact**: The pending owner is set to this address.

### Branches and code coverage (including function calls)

**Intended branches**

- Update the state of the `pendingOwner` variable to the `newOwner` address.
  ☐ Test coverage

**Negative behavior**

- Should not allow anyone other than the current owner to call this function.
  ☑ Negative test

# 6 Audit Results

At the time of our audit, the audited code was not deployed to mainnet EVM.

During our assessment on the scoped GammaSwap contracts, we discovered one finding which was of low impact.

## 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.