Zellic

GammaSwap

Smart Contract Security Assessment

August 24, 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 August 7th to August 17th, 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:

- Can a flash loan that leads to collateral rebalancing violate the collateral balances of other users?
- Can GammaPool be attacked or produce undesired results when using a flash loan, especially in interaction with CFMMs?
- Are there risks for miscalculations or rounding errors that lead to loss of funds, especially in the liquidation strategy?
- Is any of the overridden functionality-changing behavior in concrete, existing strategies?
- What are the additional implications of the functionality added after our last assessment?
- How are the implementations of the strategies designed? Do they add aditional complexity to the system? What are the possible attack vectors?

1.2 Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- Key custody
- Code outside the scope of the engagement
- Failure of the underlying CFMMs

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

During this assessment, time constraints prevented us from creating a robust invariant

test suite for the many different invariants in the protocol, such as impossible values when calculating deltas, the growth of the indexes not being affected by the order in which they are affected, and rounding errors for more extreme tokens with decimal differences.

By ascertaining which functions have which effects on indexes (positively/negatively correlated), a fuzzing suite can be created that tests edge cases and random operations and measures the deltas to ensure that certain invariants are not broken.

1.3 Results

During our assessment on the scoped GammaSwap contracts, we discovered one finding, which was informational in nature.

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	0
Informational	1

2 Introduction

2.1 About GammaSwap

GammaSwap is the modular scaling layer for DeFi liquidity, offering leveraged exposure to any token by turning impermanent loss into impermanent gains. It integrates existing AMM architectures and allows users to borrow LP tokens to take leveraged exposure to the volatility of the underlying assets.

2.2 Methodology

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

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

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

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

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

Code maturity. We look for potential improvements in the 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.

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

2.3 Scope

The engagement involved a review of the following targets:

GammaSwap Contracts

Repositories	https://github.com/gammaswap/v1-core https://github.com/gammaswap/v1-implementations
Versions	v1-core: `48f7220395af3d62ec8f75789291679745e244e8` v1-implementations: `ac26ad20bf3a63a9cfc5f18e73b3b4d6901274ad`

Programs	CPMMBaseLongStrategy.sol CPMMBaseRebalanceStrategy.sol CPMMBaseStrategy.sol CPMMBorrowStrategy.sol CPMMRepayStrategy.sol CPMMBatchLiquidationStrategy.sol CPMMExternalLiquidationStrategy.sol CPMMLiquidationStrategy.sol CPMMExternalRebalanceStrategy.sol CPMMMath.sol BaseBorrowStrategy.sol BaseLongStrategy.sol BaseStrategy.sol BaseExternalStrategy.sol BaseRebalanceStrategy.sol BaseLiquidationStrategy.sol BaseRepayStrategy.sol AbstractCollateralManager.sol AbstractLoanObserver.sol AbstractLoanObserverStore.sol BorrowStrategy.sol RepayStrategy.sol BatchLiquidationStrategy.sol SingleLiquidationStrategy.sol ExternalLiquidationStrategy.sol ExternalRebalanceStrategy.sol RebalanceStrategy.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 and a half person-weeks. The assessment was conducted over the course of one and a half calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

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.

August 7, 2023	Kick-off call
August 7, 2023	Start of primary review period
August 17, 2023	End of primary review period

3 Detailed Findings

3.1 The `abi.encodeCall` function should be used instead of `abi.encodeWithSelector`

- **Target**: Project Wide
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Project wide, `abi.encodeWithSelector` is used to encode the function calldata to an external contract. This function does not, however, perform type checking on the arguments.

Impact

This can lead to unexpected behavior if used in a dangerous context, such as arbitrary calls. Currently, however, the risk is mitigated by the fact that the function is only used to call functions on the ERC-20 interface, which is a well-known and trusted interface, as well as the fact that the inheriting contract properly sanitizes the parameters before calling the function.

Recommendations

We recommend using the `abi.encodeCall` to protect against typos and type mismatches. For example, the `safeTransfer` function can be rewritten as follows:

```
function safeTransfer(address _token, address _to, uint256 _amount)
    internal {
    (bool success, bytes memory data) = _token.call(abi.encodeWithSelecto
        r(IERC20(_token).transfer.selector, _to, _amount));

    (bool success, bytes memory data) = _token.call(abi.encodeCall(
        IERC20(_token).transfer, (_to, _amount)));

    if(!(success && (data.length == 0 || abi.decode(data, (bool)))))
    revert ST_Fail();
}
```

Remediation

This issue has been acknowledged by GammaSwap, and fixes were implemented in the following commits: 4f15249e, e4cf0325.

4 Discussion

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

4.1 Diffs over previous audit

This assessment has primarily focused on the changes implemented in the code following the previous security review report dated June 5, 2023. The following section outlines these specific changes.

CPMMMath.sol

This is a new addition to the codebase. This contract is a library that contains the math functions used in the strategies. These are mainly used to calculate deltas for specific actions, such as closing a position, withdrawing, and so forth.

CPMMBaseLongStrategy.sol

The code has been refactored, and minimal changes have been introduced, such as removing the `calcDeltasToClose`, since the math functions are now in the CPMMMath library.

CPMMBaseRebalanceStrategy.sol

This is a new addition to the codebase. It comprises of several functions used to calculate the value of the position, the deltas to rebalance, and more. It also makes use of the CPMMMath library, acting more as a wrapper for the math functions from the library.

CPMMBaseStrategy.sol

The code has been minimally refactored.

CPMMBorrowStrategy.sol

This is a new addition to the codebase. It implements over the CPMMBaseRebalanceStrategy, BorrowStrategy, and RebalanceStrategy, adding a `getCurrentCFMMPrice` function that returns the current price of the CFMM.

```
function getCurrentCFMMPrice()
    internal virtual override view returns(uint256) {
    return s.CFMM_RESERVES[1] * (10 ** s.decimals[0])
    / s.CFMM_RESERVES[0];
}
```

CPMMRepayStrategy.sol

This is a new addition to the codebase. It implements over the CPMMBaseRebalanceStrategy and RepayStrategy.

CPMMBatchLiquidationStrategy.sol

This is a new addition to the codebase. It implements over the CPMMBaseRebalanceStrategy and BatchLiquidationStrategy, overwriting the `_calcMaxCollateralNotMktImpact` function.

```
function _calcMaxCollateralNotMktImpact(uint128[] memory tokensHeld,
    uint128[] memory reserves) internal override virtual returns(uint256)
    {
        uint256 lastCFMMInvariant = Math.sqrt(uint256(reserves[0])
    * reserves[1]);
        uint256 leftVal = uint256(tokensHeld[0]) * lastCFMMInvariant
    / reserves[0];
        uint256 rightVal = uint256(tokensHeld[1]) * reserves[0]
    / lastCFMMInvariant;
        return (leftVal + rightVal) / 2;
}
```

CPMMExternalLiquidationStrategy.sol

The code has been minimally refactored. It now inherits the CPMMBaseRebalanceStrategy over the CPMMLiquidationStrategy.

CPMMLiquidationStrategy.sol

The code has been minimally refactored. It now inherits the CPMMBaseRebalanceStrategy over the CPMMBaseLongStrategy.

CPMMExternalRebalanceStrategy.sol

This is a new addition to the codebase. It implements over the CPMMBaseLongStrategy and ExternalRebalanceStrategy.

BaseBorrowStrategy.sol

The code has been minimally refactored — previously a part of `LongStrategy`, separated out into a new strategy.

BaseLongStrategy.sol

The code has been minimally refactored. It has had many functions removed into other strategies and has been moved in the new project structure.

BaseStrategy.sol

The code has not been changed.

BaseExternalStrategy.sol

The code has been minimally changed. A length check on `sendAndCalcCollateralLPTokens` has been added for the tokens and amounts variables to check that they are the same. Also, `externalSwap` has been refactored to get variables from storage and update them before fetching them with `updateIndex`.

BaseRebalanceStrategy.sol

The code has been minimally refactored — previously a part of long strategy, this code has been introduced as a new strategy due to its frequent use in all strategies. It is responsible for calculating deltas to change the ratio of CFMMs.

BaseLiquidationStrategy.sol

This is a new strategy. It contains some functions from LiquidationStrategy that were previously there, and now contains functions for getting information on loans. It has added new functions `getLiquiditableLoan` and `payLiquiditableLoan`, which are responsible for tracking the amount of liquidatable LP units in a loan even if the loan grows. This is because of an issue that allowed a user to liquidate loans more than their initial collateral because the growth of the LP tokens' worth as a result of the invariant growth was not accounted for.

BaseRepayStrategy.sol

Previously, this was a part of `BaseLongStrategy` and `LongStrategy`, responsible for paying back the loan and, it has now been separated into a separate strategy.

AbstractLoanObserver.sol

This is a new, significant addition to the codebase. This contract implements functions that keep track of the GammaPool loans, such as the `onLoanUpdate`. This function **has** to be called whenever the loan is updated in any of the strategies, so that the observer is always up-to-date.

```
function onLoanUpdate(address cfmm, uint16 protocolId, uint256 tokenId,
    bytes memory data)
    external override virtual returns(uint256 collateral) {
    address gammaPool = getGammaPoolAddress(cfmm, protocolId);
    require(msg.sender == gammaPool, "FORBIDDEN");

    // Never return collateral value
    collateral = _getCollateral(gammaPool, tokenId);

    LoanObserved memory loan = abi.decode(data, (LoanObserved));
    _onLoanUpdate(gammaPool, tokenId, loan);
}
```

AbstractLoanObserverStore.sol

This is a new addition to the codebase. This contract is inherited by the `GammaPoolFactory`. It stores collateral manager addresses that are used by the `GammaPools` for further reference. For example, in the `isPoolObserved` mapping, it stores whether a pool is observed or not.

```
function setPoolObserved(uint256 refId, address pool)
    external override virtual {
    require(msg.sender == _loanObserverStoreOwner(), "FORBIDDEN");
    require(pool != address(0), "ZERO_ADDRESS");
    require(refId > 0, "INVALID_REF_ID");

    LoanObserver storage ref = observers[refId];

    require(ref.refType > 0, "NOT_EXISTS");
```

```
    require(ref.refType < 2 || ILoanObserver(ref.refAddr).validate(pool),
    "INVALID_POOL") ;

    isPoolObserved[refId][pool] = true;
}
```

Similarly, it stores all observers in the `observers` mapping, being referenced by the `refId` of the `GammaPool`.

```
function getPoolObserverByUser(uint16 refId, address pool, address user)
    external override virtual view returns(address, uint16, uint8) {

    require(refId > 0, "REF_ID");
    require(pool ≠ address(0), "ZERO_ADDRESS_POOL");
    require(user ≠ address(0), "ZERO_ADDRESS_USER");
    require(isPoolObserved[refId][pool], "NOT_SET");

    LoanObserver memory exRef = observers[refId];
    if(!exRef.active) {
        return(address(0), 0, 0);
    }

    require(!exRef.restricted || isAllowedToBeObserved[refId][user],
    "FORBIDDEN");

    return(exRef.refAddr, exRef.refFee, exRef.refType);
}
```

AbstractCollateralManager.sol

This is a new addition to the codebase. It implements over AbstractLoanObserver. It can also liquidate positions, if called from `gammaPool`. Note that the contract inheriting this should overwrite the `_getCollateral` function so that it does not return 0.

BorrowStrategy.sol

Previously a part of LongStrategy, BorrowStrategy is responsible for exposing the functions that are used to manage loans such as `_increaseCollateral`, `_decreaseCollateral`, and `_borrowLiquidity`. Also, it has added the function `getUnfundedAmounts`, which rebalances if there is not enough collateral of one of the tokens.

RepayStrategy.sol

Previously a part of LongStrategy, RepayStrategy is now responsible for all repayment-related functionality, whether it be repaying with LP tokens or repaying and rebalancing (setting ratios). The new function `CheckCollateral` has been added to check collateral after interacting with observers.

BatchLiquidationStrategy.sol

A new liquidation strategy has been added for gas-efficiency purposes that allows for batch liquidations of positions, which aggregates the liquidity of the loans to be liquidated — if not eligible for liquidation, the loans are skipped.

SingleLiquidationStrategy.sol

Previously a part of LiquidationStrategy, this strategy exposes previously implemented functions such as `_liquidate` and `_liquidateWithLP`.

ExternalLiquidationStrategy.sol

This has been slightly changed to call the new functions that are used for liquidating loans; however, the function is mostly the same.

ExternalRebalanceStrategy.sol

Previously a part of ExternalLongStrategy, this holds all the functions that were previously used to rebalance externally (as a third party, executing the swap yourself).

RebalanceStrategy.sol

The code has been minimally refactored. Previously a part of long strategy, this code has been introduced as a new strategy due to its frequent use in all strategies. It is responsible for calculating deltas to change the ratio of CFMMs.

4.2 Consider locking pragma versions

Additionally, consider locking the pragma to ensure that experimental compilers or compilers that lack recently added optimizations or security updates cannot be used to compile the contracts.

5 Assessment Results

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

During our assessment on the scoped GammaSwap contracts, we discovered one finding, which was informational in nature. GammaSwap Protocol acknowledged the finding and implemented a fix.

5.1 Disclaimer

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

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

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