Zellic

Brahma Protected MoonShots

Smart Contract Security Assessment

May 16, 2022

Prepared for:

Bapi Reddy

Brahma

Prepared by:

Konstantin Nikolayev 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 email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Introduction

1.1 About Brahma Protected MoonShots

Brahma is a protocol enabling seamless access to risk-managed, cross-chain strategies for sustainable yield.

1.2 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 open-source 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. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these "shallow" bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, etc. 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 manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions, and summarize the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines, or code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, up-

gradeability weaknesses, centralization risks, etc.

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; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document, rather than 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 numerous soft factors, such as our clients' threat models, their business needs, project timelines, etc. We aim to provide useful and actionable advice to our partners that consider their long-term goals, rather than simply a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

Brahma Contracts

Repository	https://github.com/Brahma-fi/protected_moonshots
Versions	`858e1462980397f7022b770965eca17d66b40d3f`
Contracts	• Batcher • Vault • ConvexTradeExecutor • Harvester • PerpTradeExecutor • PerpPositionHandlerL2
Type	Solidity
Platform	Ethereum, Optimism

1.4 Project Overview

Zellic was approached to perform an assessment with two consultants, for a total of 3 person-weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-Founder
jazzy@zellic.io

Stephen Tong, Co-Founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Konstantin Nikolayev, Engineer
nyanko@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

1.5 Project Timeline

The key dates of the engagement are detailed below.

May 2, 2022	Kick-off call
May 2, 2022	Start of primary review period
May 13, 2022	End of primary review period
May 25, 2022	Closing call

1.6 Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered financial or investment advice.

2 Executive Summary

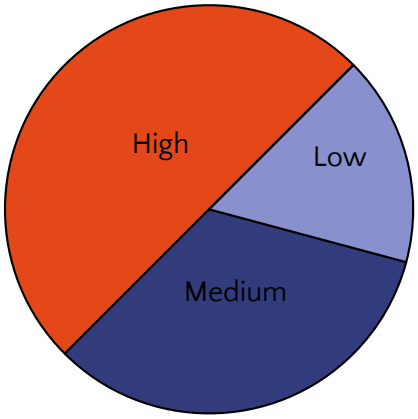
Zellic conducted an audit for Brahma from May 2nd to May 13th, 2022 on the scoped contracts and discovered 6 findings. The audit uncovered 3 findings of high impact, 2 of medium impact, and 1 of low impact.

Zellic thoroughly reviewed the Brahma Protected MoonShots codebase to find protocol-breaking bugs as defined by the documentation, or any technical issues outlined in the Methodology section of this document. Specifically, taking into account Brahma's threat model, we focused heavily on issues that would break core invariants like the issuance and redeeming of shares within the Vault, as well as problems that may arise on various interactions with the protocol.

Our general overview of the code is that the codebase is of satisfactory quality. The code coverage is high and tests are included for the majority of the functions. The documentation is adequate.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	3
Medium	2
Low	1
Informational	0



3 Detailed Findings

3.1 Deposits can be potentially frontrun and stolen

- **Target**: `Vault`
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

The shares minted in `deposit()` are calculated as a ratio of `totalVaultFunds()` and `totalSupply()`. The `totalVaultFunds()` can be potentially inflated, reducing the amounts of shares minted (even 0).

```
function deposit(uint256 amountIn, address receiver)
    ...
        shares = totalSupply() > 0
            ? (totalSupply() * amountIn) / totalVaultFunds()
            : amountIn;
        IERC20(wantToken).safeTransferFrom(receiver, address(this),
    amountIn);
        _mint(receiver, shares);
    }
...

    function totalVaultFunds() public view returns (uint256) {
        return
            IERC20(wantToken).balanceOf(address(this)) +
    totalExecutorFunds();
    }
```

By transferring `wantToken` tokens directly, `totalVaultFunds()` would be inflated (because of `balanceOf()`) and as the division result is floored, there could be a case when it would essentially mint 0 shares, causing a loss for the depositing user. If an attacker controls all of the share supply before the deposit, they would be be able to withdraw all the user deposited tokens.

Impact

Consider the following attack scenario:

1. The `Vault` contract is deployed.
2. The governance sets `batcherOnlyDeposit` to `false`.
3. The attacker deposits[1] `X` stakeable tokens and receives `X` LP tokens.
4. The victim tries to deposit `Y` stakeable tokens.
5. The attacker frontruns the victim's transaction and transfers[2] `X * (Y - 1) + 1` stakeable tokens to the `Vault` contract.
6. The victim's transaction is executed, and the victim receives 0 LP tokens.[3]
7. The attacker redeems her LP tokens, effectively stealing `Y` stakeable tokens from the victim.

The foregoing is just an example. Variations of the foregoing attack scenario are possible.

The impact of this finding is mitigated by the fact that the default value of `batcherOnlyDeposit` is `true`, which allows the keeper of the `Batcher` contract to: 1) prevent the attacker from acquiring 100% of the total supply of LP tokens; 2) prevent the attacker from redeeming her LP tokens for stakeable tokens.

Recommendations

Consider:

- adding an `amountOutMin` parameter to the `deposit(uint256 amountIn, address receiver)` function of the `Vault` contract;
- adding a `require` statement that ensures that the `deposit()` function never mints 0 or less than `amountOutMin` LP tokens.

Remediation

The issue has been acknowledged by Brahma and mitigated in commit 413b9cc.

[1] By calling the `deposit()` function of the `Vault` contract.
[2] By calling the `transfer()` function of the stakeable token contract. This doesn't increase the total supply of LP tokens. The attacker is always able to call `transfer()` to directly transfer stakeable tokens to the `Vault` contract, even when `batcherOnlyDeposit` is set to `true`.
[3] The formula for calculating the number of LP tokens received: `LPTokensReceived = Y * totalSupplyOfLPTokens / totalStakeableTokensInVault`. Substitute `totalSupplyOfLPTokens = X` and `totalStakeableTokensInVault = X + X * (Y - 1) + 1`. The result: `LPTokensReceived = Y * X / (X + X * (Y - 1) + 1) = Y * X / (X * Y + 1) = 0`.

3.2 Centralization risks

- **Target**: `Batcher`, `Vault`, `ConvexTradeExecutor`, `PerpTradeExecutor`, `Harvester`, `PerpPositionHandlerL2`
- **Category**: Code Maturity
- **Severity**: High
- **Likelihood**: n/a
- **Impact**: High

Description

The protocol is heavily centralized. This may be by design due to the the nature of yield aggregators.

The governance can call the `sweep()` function of the `Batcher`, `Vault`, `ConvexTradeExecutor`, `PerpTradeExecutor` and `Harvester` contracts, effectively draining the token balances of the aforementioned contracts.

The strategist can call the `sweep()` function of the `PerpPositionHandlerL2` contract, effectively draining the token balances of the aforementioned contract.

The documentation states that 1-10% of the user-deposited funds stay within the vault as a buffer and only the yield harvested from Curve and Convex is used for trading on Perpetual Protocol. These invariants are not enforced in any way in the `Vault` contract itself. The keeper can freely move the user-deposited funds between the vault and its trade executors. It is therefore the responsibility of keepers to enforce the aforementioned invariants

Impact

Centralization carries heavy risks, most of which have been outlined in the Description section above. A compromised governance, strategist or a keeper could potentially steal all user funds.

Recommendations

- Consider setting up multisig wallets for the governance, the strategist and the keeper.
- Consider enforcing, on the protocol level, the invariants outlined in the documentation.
- Consider following best security practices when handling the private keys of the externally-owned accounts.

Remediation

The issue has been acknowledged by the Brahma team. Further steps to securing private keys and usage of a multisig address are being addressed.

3.3 Unwanted deposits and withdrawals can be triggered on behalf of another user

- **Target**: `Vault`
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

The `deposit()` and `withdraw()` functions of the `Vault` contract accept 2 arguments:

```
function deposit(uint256 amountIn, address receiver)
        public
        override
        nonReentrant
        ensureFeesAreCollected
        returns (uint256 shares)
    {
        /// checks for only batcher deposit
        onlyBatcher();

        ...
    }

function withdraw(uint256 sharesIn, address receiver)
        public
        override
        nonReentrant
        ensureFeesAreCollected
        returns (uint256 amountOut)
    {
        /// checks for only batcher withdrawal
        onlyBatcher();
        ...
    }
```

Both of the functions call `onlyBatcher()` to check and enforce the validity of `msg.sender`:

```
function onlyBatcher() internal view {
    if (batcherOnlyDeposit) {
```

```
        require(msg.sender == batcher, "ONLY_BATCHER");
    }
}
```

Both of the functions perform no other checks of the validity of `msg.sender`.

By default (`batcherOnlyDeposit = true`), only the `Batcher` contract can deposit and withdraw funds on behalf of the receiver.

The governance can change `batcherOnlyDeposit` to `false`. When `batcherOnlyDeposit = false`, the `deposit()` and `withdraw()` functions perform no `msg.sender` validity checks whatsoever, allowing any third-party user to trigger deposits[4] and withdrawals[5] on behalf of any receiver.

Impact

A third party can trigger unwanted deposits and withdrawals on behalf of another user. This can lead to the users' confusion, lost profits and even potentially to a loss of funds.

Recommendations

Consider adding `if (!batcherOnlyDeposit) { require(msg.sender == receiver); }` checks to the `deposit()` and `withdraw()` functions.

Remediation

The issue has been fixed in commit 32d30c8.

[4] Deposits only work if the receiver has `approve()`d enough of stakeable tokens.
[5] Withdrawals only work if the receiver owns enough of the vault LP tokens.

3.4 Some emergency-only functions can be called outside of an emergency state

- **Target**: `Batcher`, `Vault`, `ConvexTradeExecutor`, `PerpTradeExecutor`, `Harvester`, `PerpPositionHandlerL2`
- **Category**: Business Logic
- **Severity**: Medium
- **Likelihood**: High
- **Impact**: Medium

Description

The project contains 6 contracts that implement a `sweep()` function:

- `Batcher`
- `Vault`
- `ConvexTradeExecutor` (derived from `BaseTradeExecutor`)
- `PerpTradeExecutor` (derived from `BaseTradeExecutor`)
- `Harvester`
- `PerpPositionHandlerL2`

The `sweep()` functions in `Batcher`, `Vault`, `ConvexTradeExecutor` and `PerpTradeExecutor` are documented as callable only in an emergency state.

Only the `sweep()` function in `Vault` implements emergency state checks. The `sweep()` functions in all other contracts do not.

Impact

The emergency-only `sweep()` functions in `Batcher`, `ConvexTradeExecutor` and `PerpTradeExecutor` can be called outside of an emergency state.

The `sweep()` functions in `Harvester` and `PerpPositionHandlerL2` can also be called outside of an emergency state, but they are not documented as callable only in an emergency state.

Recommendations

Consider adding emergency state checks to the `sweep()` functions of the `Batcher`, `ConvexTradeExecutor` and `PerpTradeExecutor` contracts.

Consider adding emergency state checks to the `sweep()` function of the `Harvester` contract and documenting it accordingly.

Consider: 1) adding an emergency state variable to the `PerpPositionHandlerL2` con-

tract;[6] 2) adding emergency state checks to the `sweep()` function of the `PerpPositionHandlerL2` contract; 3) documenting this accordingly.

Remediation

The issue has been acknowledged by the Brahma team.

[6] This step is required because the `PerpPositionHandlerL2` is deployed on top of Optimism, an L2 network, and therefore (and unlike all the other contracts) cannot access the emergency state variable of the `Vault` contract in a timely manner.

3.5 Invalid business logic in `Batcher.sol`

- **Target**: `Batcher.sol`
- **Category**: Coding Mistakes
- **Severity**: Medium
- **Likelihood**: n/a
- **Impact**: Medium

Description

The `depositFunds()` function of the `Batcher` contract contains this incorrect `require` statement at L94:

```
require(
    IERC20(vaultInfo.vaultAddress).totalSupply() -
        pendingDeposit +
        pendingWithdrawal +
        amountIn ≤
        vaultInfo.maxAmount,
    “MAX_LIMIT_EXCEEDED”
);
```

The correct `require` statement should contain `- pendingWithdrawal + pendingDeposit` instead of `- pendingDeposit + pendingWithdrawal`.

Impact

The incorrect `require` statement fails to properly enforce the “users can deposit only up to `vaultInfo.maxAmount` of stakeable tokens” invariant.

Recommendations

Consider changing `- pendingDeposit + pendingWithdrawal` to `- pendingWithdrawal + pendingDeposit` in the `require` statement.

Remediation

The issue has been mitigated and fixed accordingly in commit 0c2c815.

3.6 `batchDeposit()` and `batchWithdraw()` leave undistributed, unaccounted dust within the `Batcher` contract

- **Target**: `Batcher`
- **Category**: Code Maturity
- **Likelihood**: High
- **Severity**: Low
- **Impact**: Low

Description

The users of the `Batcher` contract use the `depositFunds()` function to deposit stakeable tokens into the batcher. The keeper of the `Batcher` contract uses the `batchDeposit()` function to merge several user deposits, send the users' stakeable tokens to the vault, receive LP tokens from the vault and distribute them among the users.

The users of the `Batcher` contract use the `initiateWithdrawal()` function to deposit LP tokens into the batcher. The keeper of the `Batcher` contract uses the `batchWithdraw()` function to merge several user withdrawals, send the users' LP tokens to the vault, receive stakeable tokens from the vault and distribute them among the users.

Here's how `batchDeposit()` distributes LP tokens:

```
for (uint256 i = 0; i < users.length; i++) {
    uint256 userAmount = depositValues[i];

    // Checks if userAmount is not 0, only then proceed to allocate LP
    tokens
    if (userAmount > 0) {
        uint256 userShare = (userAmount * (lpTokensReceived)) /
            (amountToDeposit);

        // Allocating LP tokens to user, can be calimed by the user later
    by calling claimTokens
        userLPTokens[users[i]] = userLPTokens[users[i]] + userShare;
        ++totalUsersProcessed;
    }
}
```

The code that distributes stakeable tokens in the `batchWithdraw()` function is analogous.

Consider the following scenario (using the `batchDeposit()` function as an example):

1. 3 users deposit 1 USDC each.
2. The batcher exchanges the 3 USDC for 2 LP tokens.
3. Each user receives 0.666666 LP tokens; 0.000002 LP tokens stay within the batcher, as unaccounted dust.

The `Batcher` contract provides no way to withdraw this dust.

There is an analogous issue in the `batchWithdraw()` function.

Impact

Unaccounted, undistributed and unreclaimable dust accumulates within the `Batcher` contract.

Recommendations

Consider introducing: 1) 2 dust counter variables: one for stakeable tokens and another for LP tokens; 2) a dust withdrawal function.[7]

Consider modifying the `batchWithdraw()` and `batchDeposit()` functions so that they immediately distribute the dust among some of the users.[8][9]

Consider modifying the `sweep()` function so that it allows the governance to withdraw partial amounts and use that to withdraw the dust.[10]

Remediation

The issue has been acknowledged by the Brahma team.

[7] This introduces significant additional gas costs.
[8] This introduces additional gas costs.
[9] This introduces some (hopefully insignificant) unfairness: some of the users receive slightly more tokens, some receive slightly less.
[10] This fails to address centralization risks.

4 Discussion

In this section, we discuss miscellaneous interesting observations discovered during the audit that are noteworthy and merit some consideration.

In `PerpPositionHandler`: There are no events for opening and closing positions. You might want to consider adding them.

```
function _openPosition(bytes calldata data) internal override {
    OpenPositionParams memory openPositionParams = abi.decode(
        data,
        (OpenPositionParams)
    );
    bytes memory L2calldata = abi.encodeWithSelector(
        IPositionHandler.openPosition.selector,
        openPositionParams._isShort,
        openPositionParams._amount,
        openPositionParams._slippage
    );

    sendMessageToL2(
        positionHandlerL2Address,
        L2calldata,
        openPositionParams._gasLimit
    );
}

function _closePosition(bytes calldata data) internal override {
    ClosePositionParams memory closePositionParams = abi.decode(
        data,
        (ClosePositionParams)
    );
    bytes memory L2calldata = abi.encodeWithSelector(
        IPositionHandler.closePosition.selector,
        closePositionParams._slippage
    );
    sendMessageToL2(
        positionHandlerL2Address,
        L2calldata,
        closePositionParams._gasLimit
```

```
    );
}
```

In `library/AddArrayLib.sol`: The `removeAddress()` function is documented as returning boolean. But it returns nothing.

```
/**
 * @notice remove an address from the array
 * @dev finds the element, swaps it with the last element, and then
    deletes it;
 * returns a boolean whether the element was found and deleted
 * @param self Storage array containing address type variables
 * @param element the element to remove from the array
 */
function removeAddress(Addresses storage self, address element) internal
    {
    for (uint256 i = 0; i < self.size(); i++) {
        if (self._items[i] == element) {
            self._items[i] = self._items[self.size() - 1];
            self._items.pop();
        }
    }
}
```

Here is an interesting observation about how fees are charged: Let's say 1000 USDC are deposited into the protocol. The executors manage to lose 500 USDC. The next deposit or withdrawal happens, no fee is charged. Then the executors manage to gain 100 USDC. The next deposit or withdrawal happens, a fee is charged on that 100-USDC "gain". The stakers have effectively lost 400 USDC, but still have to pay a fee.

Lax access control in token withdrawals from `Harvester`: If someone accidentally deposits token into it, anyone can take them away. This mitigated by the fact that the default user of `Harvester`, `ConvexTradeExecutor`, never leaves tokens in `Harvester`.

```
/// @notice Harvest the entire swap tokens list, i.e convert them into
    wantToken
/// @dev Pulls all swap token balances from the msg.sender, swaps them
    into wantToken, and sends back the wantToken balance
function harvest() external override {
    uint256 crvBalance = crv.balanceOf(address(this));
```

```
    uint256 cvxBalance = cvx.balanceOf(address(this));
    uint256 _3crvBalance = _3crv.balanceOf(address(this));
...
    // send token usdc back to vault
    IERC20(vault.wantToken()).safeTransfer(
        msg.sender,
        IERC20(vault.wantToken()).balanceOf(address(this))
    );
}
```

In `ConvexPositionHandler.sol`: To protect against any possible Curve manipulations, the usage of virtual price has to be taken into account when calcuating the `positionInWantToken`, and for that we need to set the parameter to `true`. Instead of relying on on-chain AMM's pricing when calculating the value of staked and lp tokens in this case, using the `virtual_price` offered by Curve does not allow for any possible manipulations that an attacker could ever cause.

```
...
(
    uint256 stakedLpBalance,
    uint256 lpTokenBalance,
    uint256 usdcBalance
) = _getTotalBalancesInWantToken(true);
```

Although the usage of the virtual price was set in the `_withdraw()` function, we also have to take the minimum of the calculated and actual staked balance, such that no overflow can occur:

```
function _withdraw(bytes calldata _data) internal override {
    ...
    uint256 lpTokensToUnstake = _USDCValueInLpToken(
        amountToUnstake
    ) > baseRewardPool.balanceOf(address(this))
        ? baseRewardPool.balanceOf(address(this))
        : _USDCValueInLpToken(amountToUnstake);
    ...
```

The protection also applies in the case of calculating the amount of lp tokens to convert into `USDC`:

```
function _withdraw(bytes calldata _data) internal override {
    ...
    uint256 lpTokensToConvert = _USDCValueInLpToken(
        usdcValueOfLpTokensToConvert
    ) > lpToken.balanceOf(address(this))
        ? lpToken.balanceOf(address(this))
        : _USDCValueInLpToken(usdcValueOfLpTokensToConvert);
    ...
```

This has been found and properly mitigated by the Brahma team.