Zellic

# Definitive

## Smart Contract Security Assessment

**June 9, 2023**

*Prepared for:*

**Blake Arnold**

Definitive

*Prepared by:*

**Junyi Wang and Ayaz Mammadov**

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 Definitive from April 24th, 2023 to May 1st, 2023. During this engagement, Zellic reviewed Definitive'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:

- If a malicious actor is able to compromise Definitive's keys, can they steal client principal?
- Are clients always able to withdraw their principal?
- Can anyone else other than Definitive and the client interact with the contracts?

## 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

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

## 1.3 Results

During our assessment on the scoped Definitive contracts, we discovered five findings. No critical issues were found. Of the five findings, three were of high impact, one was of medium impact, and the remaining finding was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for Definitive's benefit in the Discussion section (4) at the end of the document.

## Breakdown of Finding Impacts

| Issue | Secure Definitive Role | Compromised Definitive Role | Compromised Definitive Admin Role | Status |
|---|---|---|---|---|
| 3.1 User funds may be stolen through swap fees | Informational | High | High | Remediated |
| 3.2 User funds may be stolen through fake pool | Informational | High | High | Remediated |
| 3.3 The Definitive role can steal user funds by forcing bad trades | Informational | High | High | Acknowledged |
| 3.4 The Definitive role can steal user funds by forcing many trades | Informational | Informational | Medium | Remediated |
| 3.5 Reward fees are not validated | Informational | Low | Low | Acknowledged |

*see detailed findings for Definitive's acknowledgement responses

Definitive requested a contract review with the assumption that the administrative *Definitive* role could be compromised due to reasons such as an attacker stealing Definitive's private keys or a malicious insider.

To address varying levels of risk among clients, we have included a table in this report that details the potential consequences of a compromised *Definitive* role.

Please note that our finding's Likelihood ratings are not applicable (N/A) in this audit report, as it is presumed that Definitive is not a malicious actor and does not have intentions to steal client funds.

# 2 Introduction

## 2.1 About Definitive

Definitive offers yield farming, which allows clients to automatically claim and auto-compound on protocols such as Convex, Stargate, Synapse, and Hop. The platform's proprietary back end utilizes external events to automate on-chain positions for clients. Each client has their own vault smart contract deployed and can interact with it via our web front end. Clients can deposit and withdraw funds, while Definitive (our centralized keepers) can claim, auto-compound, add liquidity, and remove liquidity from the protocols.

Definitive is similar to Yearn, but instead of one vault for all customers, Definitive deploys a bespoke vault per customer. Definitive has a specific asset control layer that limits what clients can do and what Definitive (centralized keepers) can do.

## 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:

### Definitive Contracts

| | |
|---|---|
| **Repository** | https://github.com/DefinitiveCo/contracts |
| **Version** | contracts: `e11de755736cfad2241ba0da9c13fce1e7f560da` |
| **Program** | • contracts/* |
| **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:

**Junyi Wang**, Engineer
junyi@zellic.io

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

## 2.5 Project Timeline

The key dates of the engagement are detailed below.

| | |
|---|---|
| **April 24, 2023** | Kick-off call |
| **April 24, 2023** | Start of primary review period |
| **May 1, 2023** | End of primary review period |

# 3 Detailed Findings

## 3.1 User funds may be stolen through swap fees

- **Target**: BaseSwap.sol, CoreSwap.sol
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: High
- **Impact**: High

### Description

The `swap` function allows a parameter of `feePct` to be passed and only checks if it is at most 100%. In the event of a compromise of Definitive's private keys by an attacker, the attacker could set the swap fees to 100%, which would allow them to take the entire swap output.

When swapping, there is a check to ensure that the input coin of the swap is a reward coin paid by an underlying protocol when the swap fees are nonzero.

```
function _getValidatedPayloadAmount(/* ... */)
    private view returns (uint256 amount) {
        //[ ... ]
        if (enforceSwapToken && !_swapTokens[payload.swapToken]) {
            revert InvalidSwapToken();
        }
        // [ ... ]
}
```

The purpose of this check is to allow Definitive to swap only reward coins paid out by underlying protocols while collecting their fees. The whitelist should only include coins that are not part of the user's principal. Only the user may add to this whitelist.

However, there is a `delegatecall` swap handler mechanism that enables bypassing this check.

```
function _swap(
    SwapPayload[] memory payloads,
    address expectedOutputToken,
    bool enforceSwapToken
) internal returns (uint256) {
        // [ ... ]
        if (payload.isDelegate) {
            // slither-disable-next-line controlled-delegatecall
            (_success, _returnBytes)
    = payload.handler.delegatecall(payload.handlerCalldata);
        } else { ... }
        // [ ... ]
}
```

The `payload.handlerCalldata` variable includes a `swapToken` parameter to the called handler, which is the real input token of the swap.

The `payload.isDelegate`, `payload.handlerCalldata`, and `payload.swapToken` variables are set separately, which means the `swapToken` in `payload.handlerCalldata` and `payload.swapToken` are not necessarily the same.

In the event of a compromise of Definitive's private keys by an attacker, the attacker can pass a `payload.swapToken` that passes the whitelist check, while swapping another coin in reality.

Since this is a `delegatecall`, the handler will have access to all of the contract's coins. This allows an attacker to swap any coin, including coins that may be the user's principal.

### Impact

If Definitive's private keys are compromised, an attacker may steal user funds through the actions described above.

### Recommendations

It is recommended to remove the `delegatecall` mechanism, if possible. Additionally, it is advisable to add checks in each of the handlers to ensure that the data being passed into `_swap` (and therefore used for checks) is the same as the data being passed into the handler for the actual swap action. This will help to ensure consistency and prevent potential vulnerabilities.

### Remediation

This issue has been remediated by the Definitive team in commit 35ffa99c.

A new standardized swap handler system has been implemented, each handler now has a validation method that ensures that `swapHandler` parameters and `callData` parameters refer to the same tokens, also performing other checks when necessary.

## 3.2 User funds may be stolen through fake pool

- **Target**: CoreSwap.sol
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: High
- **Impact**: High

### Description

The CoreSwap contract is responsible for swapping the user's funds through pools. The contract performs a `delegatecall` or a regular function `call` to a whitelisted contract as shown below.

```
function swap(/* ... */){
    // [ ... ]
    if (payload.isDelegate) {
        // slither-disable-next-line controlled-delegatecall
        (_success, _returnBytes)
    = payload.handler.delegatecall(payload.handlerCalldata);
    } else {
        _prepareAssetsForNonDelegateHandlerCall(payload, _amount);
        (_success, _returnBytes)
    = payload.handler.call(payload.handlerCalldata);
    }
    // [ ... ]
}
```

The `payload` parameter is fully controlled by the caller, which is presumed to be Definitive. The `handlerCalldata` is then passed to one of the handler contracts, which are currently UniswapV2SwapHandler and UniswapV3SwapHandler. Both of these contracts allow the specification of a swap path, which consists of a sequence of pools to swap the tokens through. However, the swap path is not currently being validated.

### Impact

If an attacker compromises Definitive's private keys, they can manipulate the swap path by specifying a controlled pool and setting the expected output amount to a small value, effectively disabling the slippage protection. This could enable the attacker to steal the user's funds through the controlled pool.

## Recommendations

We recommend that the libraries used by the swap routers of the relevant pool types be imported and that the path in the swap function be parsed as part of a check. The pools can be whitelisted, only allowing trusted public pools to be used while swapping user funds. By validating the swap path, the potential for an attacker to pass a controlled pool can be mitigated.

## Remediation

This issue has been remediated by the Definitive team in commit 35ffa99c.

A new standardized swap handler system has been implemented, each handler now has a validation method that ensures that `swapHandler` parameters and `callData` parameters refer to the same tokens, also performing other checks when necessary.

## 3.3 The `Definitive` role can steal user funds by forcing bad trades

- **Target**: CoreSwap.sol
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: High
- **Impact**: High

### Description

If the `Definitive` role is compromised, it could be used to execute trades that manipulate markets in specific exchanges resulting in below market rates. This could be exploited by the attacker to gain profits. This is particularly relevant in decentralized automated market makers like Uniswap where the calculations and amounts can be predetermined and planned, unlike other closed source exchanges.

### Impact

In the case of a security compromise of the `Definitive` role, all client user funds could be stolen.

### Recommendations

Zellic spoke with the team and two different solutions were proposed:

1. Use the oracle prices to set a minimum bound for trades, using chainlink pair prices if possible to get the most accurate prices; if a specific pair does not exist, extrapolate the pair prices from the relative USD prices of the two assets from chainlink feeds. While this approach offers highly accurate and regularly maintained prices via chainlink, it may not cover assets that are not included in the chainlink price feeds.

   In situations where an oracle is not available or returns stale data, it's important to have a fallback plan. Using pre-registered swap paths can be a viable option, but it's worth noting that the path may result in less favorable pricing, especially if market conditions have changed significantly since the path was registered.

2. Allow the client to define their bounds. This technique allows a client to specify their personal bounds; however, this technique also requires a large amount of upkeep as the user-maintained paths have to be kept close to market prices, otherwise price deviations will allow for compromised `Definitive` roles to repeatedly drain funds.

## Remediation

This issue has been acknowledged by the Definitive team, they state the trusted swap pool paths reduce the likelihood of this attack in practice due to the expensive cost. The team also mentioned that the recommended changes above will be implemented in the future.

Definitive states,

> By (1) Only allowing trusted swap pools and (2) limiting the number of swaps per transaction we believe this attack is more expensive to perform. We will only whitelist high TVL swap pools and regularly verify the allowed pools list to reduce the risk.
>
> Additionally, the swap handlers can be updated at a later date to mitigate any risk if it becomes more viable.

## 3.4 The `Definitive` role can steal user funds by forcing many trades

- **Target**: CoreSwap.sol, Protocol
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Medium
- **Impact**: Medium

### Description

If the `Definitive` role is compromised, an attacker could carry out multiple trades using this role.

### Impact

In the case of a security compromise of the `Definitive` role, an attacker could take a flash loan and become a majority liquidity provider (LP) in a Uniswap pool, and then using the compromised `Definitive` role, they could force many trades such that the client loses a significant amount of capital in fees, which the attacker profits from by being the major LP of the pool.

### Recommendations

Limit the number of swaps allowed per transaction, this would force users to expose their own funds and not flashloan funds, and would make the attack extremely costly.

### Remediation

This issue was remediated by the Definitive team in commit add78562 by adding a limit to the number of swaps that can be executed in a single transaction.

## 3.5 Reward fees are not validated

- **Target**: BaseRewards.sol
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Low

### Description

In the `claimAllRewards` function, the `feePct` parameter is not sufficiently validated beyond ensuring that it is at most 100%. This function can be called by either Definitive or the user.

### Impact

Definitive, or an attacker in possession of their private key, can call this function with 100% fees and take all rewards. The user can call this function with 0% fees before Definitive calls it and avoid paying fees to Definitive. However, either action by Definitive or the user is visible on the blockchain to the other party.

### Recommendations

Add additional validation checks to the `claimAllRewards` function. The function should check that the `feePct` parameter is within a reasonable range and not excessively high.

### Remediation

This issue has been acknowledged by Definitive.

Definitive states,

> Definitive has fee agreements with all clients. Fees can be collected based on the the fee agreements and on-chain data. Due to the variability in these fees, it allows for Definitive to offer more optimized collection of fees, should a black-swan or incentive opportunities (for either party) arise. All fees will be communicated to the clients and can be retroactively modified, if needed.

# 4 Discussion

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

## 4.1 Centralization risk

As Definitive will be closely collaborating with select institutional clients, certain trust assumptions have been made regarding their ability to trade on the client's behalf and collect fees. Therefore, the protocol's design relies heavily on the operational security and trustworthiness of its administrators/owners. However, as our findings highlight, if a malicious actor gains access to the `Definitive` role through a key compromise, it can have severe consequences.

Despite the smart contract protections in place, there is still a significant level of trust that customers must place with Definitive. The platform enables `Definitive` to execute trades on behalf of users, which can potentially result in `Definitive` profiting at the user's expense. For example, `Definitive` may use the user's funds to improve the price of an asset before purchasing it themselves or even front-run the user's market actions. With a broad range of market actions at their disposal, there are numerous ways for `Definitive` to profit off of user funds.

Furthermore, the fees charged by `Definitive` are solely determined by them and do not require user consent at the smart contract level — `Definitive` may take all of the user's rewards from staking or liquidity providing, but not the principal. If the token's design includes rewards as a counterbalance to inflation, users may lose principal value to `Definitive`. However, any such actions to take high fees from users will be recorded on the blockchain, providing a degree of transparency and mutual trust.

To mitigate centralization risks, we recommend that Definitive safeguard trusted roles with a multi-sig wallet. Additionally, private keys should be stored in hardware wallets or other secure offline storage solutions. Regular auditing of these controls and storage solutions should also be conducted to ensure their effectiveness and identify any potential vulnerabilities.

# 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 Only Definitive or client may interact with the contracts

We verify that only Definitive or the client may interact with the contracts by exhaustively checking every `public` and `external` function. Functions with duplicate signatures are not repeated.

### Functions with `public`

```
function enableStopGuardian() public override onlyAdmins
```

Restriction: `onlyAdmins`

```
function disableStopGuardian() public override onlyClientAdmin
```

Restriction: `onlyClientAdmin`

```
function updateFeeAccount(address payable _feeAccount)
    public override onlyDefinitiveAdmin
```

Restriction: `onlyDefinitiveAdmin`

```
function enableSwapTokens(address[] memory swapTokens)
    public override onlyClientAdmin stopGuarded
```

Restriction: `onlyClientAdmin`

```
function disableSwapTokens(address[] memory swapTokens)
    public override onlyAdmins
```

Restriction: `onlyAdmins`

```
function enableSwapOutputTokens(address[] memory swapOutputTokens)
    public override onlyClientAdmin stopGuarded
```

Restriction: `onlyClientAdmin`

```
function disableSwapOutputTokens(address[] memory swapOutputTokens)
    public override onlyAdmins
```

Restriction: `onlyAdmins`

```
function enableSwapHandlers(address[] memory swapHandlers)
    public override onlyClientAdmin stopGuarded
```

Restriction: `onlyClientAdmin`

```
function disableSwapHandlers(address[] memory swapHandlers)
    public override onlyAdmins
```

Restriction: `onlyAdmins`

```
function wrap(uint256 amount) public onlyWhitelisted nonReentrant
```

Restriction: `onlyWhitelisted`

```
function unwrap(uint256 amount) public onlyWhitelisted nonReentrant
```

Restriction: `onlyWhitelisted`

```
function deposit(
    uint256[] calldata amounts,
    address[] calldata erc20Tokens
) public override onlyClients nonReentrant stopGuarded
```

Restriction: `onlyClients`

```
function withdraw(
    uint256 amount,
    address erc20Token
) public virtual override onlyClients nonReentrant stopGuarded returns
    (bool)
```

Restriction: `onlyClients`

```
function withdrawTo(
    uint256 amount,
    address erc20Token,
    address to
) public virtual override onlyWhitelisted nonReentrant stopGuarded
    returns (bool)
```

Restriction: `onlyWhitelisted`

```
function supportsNativeAssets()
    public pure virtual override returns (bool)
```

This function is `pure`.

```
function getBalance(address assetAddress) public view returns (uint256)
```

This function is `view`.

```
function getAmountStaked() public view returns (uint256)
```

This function is `view`.

```
function unclaimedRewards() public view override returns (IERC20[]
    memory, uint256[] memory)
```

This function is `view`.

```
function unclaimedRewards(
    address convexRewarder
) public view returns (IERC20[] memory rewardTokens, uint256[]
    memory earnedAmounts)
```

This function is `view`.

```
function getRewardTokens(address convexRewarder)
    public view returns (IERC20[] memory rewardTokens)
```

This function is `view`.

## Functions with `external`

```
function claimAllRewards(
    uint256 feePct
)
    external
    override
    onlyWhitelisted
    nonReentrant
    stopGuarded
    returns (IERC20[] memory rewardTokens, uint256[] memory earnedAmounts)
```

Restriction: `onlyWhitelisted`

```
function swap(
    SwapPayload[] memory payloads,
    address outputToken,
    uint256 amountOutMin,
    uint256 feePct
) external override onlyDefinitive nonReentrant stopGuarded returns
    (uint256 outputAmount)
```

Restriction: `onlyDefinitive`

```
function depositNative(uint256 amount)
    external payable override onlyClients nonReentrant stopGuarded
```

Restriction: `onlyClients`

```
function addLiquidity(
    uint256[] calldata amounts,
    uint256 minAmount
) external onlyDefinitive nonReentrant returns (uint256 lpTokenAmount)
```

Restriction: `onlyDefinitive`

```
function removeLiquidity(
    uint256 lpTokenAmount,
    uint256[] calldata minAmounts
) external onlyDefinitive nonReentrant returns (uint256[] memory amounts)
```

Restriction: `onlyDefinitive`

```
function removeLiquidityOneCoin(
    uint256 lpTokenAmount,
    uint256 minAmount,
    uint8 index
) external onlyDefinitive nonReentrant returns (uint256[] memory amounts)
```

Restriction: `onlyDefinitive`

```
function stake(uint256 amount) external onlyDefinitive nonReentrant
```

Restriction: `onlyDefinitive`

```
function unstake(uint256 amount) external onlyDefinitive nonReentrant
```

Restriction: `onlyDefinitive`

```
function enter(
    uint256[] calldata amounts,
    uint256 minAmount
) external onlyWhitelisted stopGuarded nonReentrant returns (uint256
    stakedAmount)
```

Restriction: `onlyWhitelisted`

```
function exitOne(
    uint256 lpTokenAmount,
    uint256 minAmount,
    uint8 index
) external onlyWhitelisted stopGuarded nonReentrant returns (uint256
    amount)
```

Restriction: `onlyWhitelisted`

```
function exit(
    uint256 lpTokenAmount,
    uint256[] calldata minAmounts
) external onlyWhitelisted stopGuarded nonReentrant returns (uint256[]
    memory amounts)
```

Restriction: `onlyWhitelisted`

```
function redeemLocal(
    StargateRedeemPayload memory payload
) external payable nonReentrant onlyWhitelisted returns (bool)
```

Restriction: `onlyWhitelisted`

```
function redeemRemote(
    StargateRedeemPayload memory payload
) external payable nonReentrant onlyWhitelisted returns (bool)
```

Restriction: `onlyWhitelisted`

### Exception: `UniswapV2SwapHandler` and `UniswapV3SwapHandler`

These functions only handle funds on behalf of `msg.sender` and do not keep any funds themselves. Therefore it is safe for these contracts to be callable externally by anyone.

### Exception: OwnableMulticall

The contract is unused, so it is safe for it to be externally callable by anyone.

### Exception: `multicall` function

```
function multicall(bytes[] calldata data) external returns (bytes[]
    memory results) {
    results = new bytes[](data.length);
    for (uint256 i = 0; i < data.length; i++) {
        results[i] = Address.functionDelegateCall(address(this),
    data[i]);
    }
}
```

Even though this function is callable by anyone, the function merely `delegatecalls` into the current contract. Any of the external functions callable by such an action will have an enforced restriction. Therefore it is safe for this function to be callable by anyone.

### Exception: `ConvexHelper`

`ConvexHelper` is a library. That means all code contained in this code file is called in the context of another contract. It is safe for the functions of this file to be callable by anyone, since changes will be made only to the calling contract.

## 5.2 Only the client may withdraw the principal

For the purposes of this section, we only consider direct methods of stealing the client principal. It may be possible to profit at the expense of the client using only the restricted control over client trades. This is explicitly excluded from consideration. We also assume that the underlying protocols will not be exploited.

To ensure that the client funds are safe, we consider every funds transfer and ensure that it is either to a protocol, for the purposes of a swap, to the client themselves, or some other legitimate purpose. Definitive is allowed to take a fee as long as it is from rewards and not principal. There is no limit on the magnitude of this fee.

Functions included in multiple categories are only included in one of them and not repeated.

### All transfers utilizing `safeTransferFrom`

From CoreDeposit.sol,

```
function _deposit(uint256[] calldata amounts, address[]
    calldata erc20Tokens) internal virtual {
    // [ ... ]
    for (uint256 i = 0; i < amounts.length; i++) {
        IERC20(erc20Tokens[i]).safeTransferFrom(_msgSender(),
    address(this), amounts[i]);
```

This transfer can only be used to deposit funds into the contract, since the destination is hardcoded.

From UniswapV2SwapHandler.sol,

```
function swapExactInputCall(
    // [ ... ]
) public returns (uint256 amountOut, address) {
    IERC20(swapToken).safeTransferFrom(_msgSender(), address(this),
    amountIn);
```

These pose an insecurity; see sections 3.2, 3.3 and 3.4 for remediations and responses.

From UniswapV3SwapHandler.sol,

```
function swapExactInputSingleHopCall(
    // [ ... ]
) public returns (uint256, address) {
    IERC20(swapToken).safeTransferFrom(_msgSender(), address(this),
    amountIn);
```

```
function swapExactInputMultihopCall(
    // [ ... ]
) public returns (uint256, address) {
    IERC20(swapToken).safeTransferFrom(_msgSender(), address(this),
    amountIn);
```

These pose an insecurity; see sections 3.2, 3.3 and 3.4 for remediations and responses.

From OwnableMulticall.sol,

```
function withdraw(uint256 amount, address assetAddress)
    external onlyOwner returns (bool) {
    // [ ... ]
        IERC20(assetAddress).safeTransferFrom(address(this),
    _msgSender(), amount);
```

This is unused.

### All transfers utilizing `safeTransfer`

From BaseFees.sol,

```
function _handleFeesOnAmount(address token, uint256 amount,
    uint256 feePct) internal returns (uint256 feeAmount) {
    // [ ... ]
    if (feeAmount > 0) {
        if (token == DefinitiveConstants.NATIVE_ASSET_ADDRESS) {
            DefinitiveAssets.safeTransferETH(FEE_ACCOUNT, feeAmount);
        } else {
            IERC20(token).safeTransfer(FEE_ACCOUNT, feeAmount);
        }
    }
}
```

These pose an insecurity; see sections 3.1 for remediation.

From CoreWithdraw.sol,

```
function _withdrawTo(uint256 amount, address erc20Token, address to)
    internal returns (bool success) {
    if (erc20Token == DefinitiveConstants.NATIVE_ASSET_ADDRESS) {
        DefinitiveAssets.safeTransferETH(payable(to), amount);
    } else {
        IERC20(erc20Token).safeTransfer(to, amount);
    }
```

`_withdrawTo` is only used in two places.

```
function withdrawTo(
    uint256 amount,
    address erc20Token,
    address to
) public virtual override onlyWhitelisted nonReentrant stopGuarded
    returns (bool)
    {
    // `to` account must be a client
    _checkRole(ROLE_CLIENT, to);
    return _withdrawTo(amount, erc20Token, to);
}
```

In this use, the `to` address is guaranteed to be a client address,

```
function _withdraw(uint256 amount, address erc20Token)
    internal returns (bool) {
    return _withdrawTo(amount, erc20Token, _msgSender());
```

and `_withdraw` itself is only used here:

```
function withdraw(
    uint256 amount,
    address erc20Token
) public virtual override onlyClients nonReentrant stopGuarded returns
    (bool)
    {
    return _withdraw(amount, erc20Token);
}
```

The function is guarded by `onlyClients`, and the destination is `msg.sender`.

From ExampleHandler.sol,

```
function handleStuffCall(uint256 amount, address token)
    public returns (uint256 amountOut, address) {
    if (token == DefinitiveConstants.NATIVE_ASSET_ADDRESS) {
        DefinitiveAssets.safeTransferETH(payable(_msgSender()), amount);
    } else {
        IERC20(token).safeTransfer(_msgSender(), amount);
    }
```

`ExampleHandler` is presumably not meant to be used in production.

### Transfers involving `resetAndSafeIncreaseAllowance`

From CoreSwap.sol,

```
function _prepareAssetsForNonDelegateHandlerCall(SwapPayload memory
    payload, uint256 amount)
    private {
    if (payload.swapToken == DefinitiveConstants.NATIVE_ASSET_ADDRESS) {
        DefinitiveAssets.safeTransferETH(payable(payload.handler),
    amount);
    } else {

    IERC20(payload.swapToken).resetAndSafeIncreaseAllowance(address(this),
    payload.handler, amount);
```

These pose an insecurity; see sections 3.2, 3.3 and 3.4 for remediations and responses.

From UniswapV2SwapHandler.sol,

```
function swapExactInput(
    // [...]
) internal returns (uint256 amountOut, address) {
    IERC20(swapToken).resetAndSafeIncreaseAllowance(address(this),
    address(swapRouter), amountIn);
```

These pose an insecurity; see sections 3.1, 3.3 and 3.4 for remediations and responses.

From UniswapV3SwapHandler.sol,

```
function swapExactInputSingleHop(
    // [...]
) public returns (uint256, address) {
    IERC20(swapToken).resetAndSafeIncreaseAllowance(address(this),
    address(swapRouter), amountIn);
```

```
function swapExactInputMultihop(
    // [...]
) internal returns (uint256, address) {
    IERC20(swapToken).resetAndSafeIncreaseAllowance(address(this),
    address(swapRouter), amountIn);
```

These pose an insecurity; see sections 3.1, 3.3 and 3.4 for remediations and responses.

### The external interface contracts

For the external interface contracts, we confirm that they only ever transfer funds to into hardcoded addresses that are presumed to be trusted and that the deposit is set up such that the current contract is the recipient of the deposit. The below code segments are abridged to show only the relevant parts.

From HopStrategy.sol,

```
IERC20(LP_UNDERLYING_TOKENS[i]).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, amounts[i]);
ISwapHop(LP_DEPOSIT_POOL).addLiquidity(amounts, minAmount,
    block.timestamp + DEFAULT_DEADLINE);
```

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, lpTokenAmount);
ISwapHop(LP_DEPOSIT_POOL).removeLiquidity(lpTokenAmount, minAmounts,
    block.timestamp + DEFAULT_DEADLINE);
```

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, lpTokenAmount);
ISwapHop(LP_DEPOSIT_POOL).removeLiquidityOneToken(
    lpTokenAmount,
    index,
    minAmount,
    block.timestamp + DEFAULT_DEADLINE
);
```

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this), LP_STAKING,
    amount);
IStakingRewards(LP_STAKING).stake(amount);
```

From BalancerBase.sol,

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this), LP_STAKING,
    amount);
IRewardsGauge(LP_STAKING).deposit(amount);
```

From BalancerStrategy.sol,

```
IERC20(LP_UNDERLYING_TOKENS[i]).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, amounts[i]);
// [ ... ]
IVault(LP_DEPOSIT_POOL).joinPool(cfg.poolId, address(this),
    address(this), joinPoolRequest);
```

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, lpTokenAmount);
// [ ... ]
IVault(LP_DEPOSIT_POOL).exitPool(cfg.poolId, address(this),
    payable(address(this)), exitPoolRequest);
```

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, lpTokenAmount);
// [ ... ]
IVault(LP_DEPOSIT_POOL).exitPool(cfg.poolId, address(this),
    payable(address(this)), exitPoolRequest);
```

From BalancerStrategyNative.sol,

```
IERC20(underlyingTokens[i]).resetAndSafeIncreaseAllowance(
    address(this),
    LP_DEPOSIT_POOL,
    amounts[i]
);
// [ ... ]
IVault(LP_DEPOSIT_POOL).joinPool{ value: nativeAssetAmount }(
    cfg.poolId,
    address(this),
    address(this),
    joinPoolRequest
);
```

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, lpTokenAmount);
// [ ... ]
IVault(LP_DEPOSIT_POOL).exitPool(cfg.poolId, address(this),
    payable(address(this)), exitPoolRequest);
```

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, lpTokenAmount);
// [ ... ]
IVault(LP_DEPOSIT_POOL).exitPool(cfg.poolId, address(this),
    payable(address(this)), exitPoolRequest);
```

From ConvexStrategy.sol,

```
IERC20(LP_UNDERLYING_TOKENS[i]).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, amounts[i]);
return
    ConvexHelper.addCurveLiquidity(
        LP_TOKEN,
        LP_DEPOSIT_POOL,
        amounts,
        minAmount,
        cfg.isMetapool,
        LP_UNDERLYING_TOKENS_COUNT
    );
```

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, lpTokenAmount);
if (cfg.isMetapool) {
    ICurveBase(LP_DEPOSIT_POOL).remove_liquidity_one_coin(
        LP_TOKEN,
        lpTokenAmount,
        int128(uint128(index)),
        minAmount
    );
} else {
    ICurveBase(LP_DEPOSIT_POOL).remove_liquidity_one_coin(lpTokenAmount,
    index, minAmount);
}
```

From ConvexStrategyNative.sol,

```
IERC20(LP_UNDERLYING_TOKENS[i]).resetAndSafeIncreaseAllowance(
            address(this),
            LP_DEPOSIT_POOL,
            amounts[i]
        );
    }
}
return ConvexHelper.addCurveLiquidityNative(LP_DEPOSIT_POOL, amounts,
    nativeAssetAmount, minAmount);
```

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, lpTokenAmount);
ICurveNative(LP_DEPOSIT_POOL).remove_liquidity_one_coin(lpTokenAmount,
    int128(uint128(index)), minAmount);
```

From StargateBase.sol,

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this), LP_STAKING,
    amount);
ILPStaking(LP_STAKING).deposit(LP_STAKING_POOL_ID, amount);
```

From StargateStrategy.sol,

```
IERC20(LP_UNDERLYING_TOKENS[0]).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, amounts[0]);
IStargateRouter(LP_DEPOSIT_POOL).addLiquidity(cfg.depositTokenPoolId,
    amounts[0], address(this));
```

From SynapseBase.sol,

```
IERC20(LP_TOKEN).resetAndSafeIncreaseAllowance(address(this),
    LP_DEPOSIT_POOL, lpTokenAmount);
ISwapSynapse(LP_DEPOSIT_POOL).removeLiquidity(lpTokenAmount, minAmounts,
    block.timestamp + DEFAULT_DEADLINE);
```

From SynapseStrategy.sol,

```
IERC20(LP_UNDERLYING_TOKENS[i]).resetAndSafeIncreaseAllowance(
    address(this),
    LP_DEPOSIT_POOL,
    amounts[i]
);
return ISwapSynapse(LP_DEPOSIT_POOL).addLiquidity(amounts, minAmount,
    deadline) > 0;
```

From SynapseStrategyNative.sol,

```
IERC20(LP_UNDERLYING_TOKENS[i]).resetAndSafeIncreaseAllowance(
    address(this),
    LP_DEPOSIT_POOL,
    amounts[i]
);
return
    ISwapSynapse(payable(LP_DEPOSIT_POOL)).addLiquidity{
    value: nativeAssetAmount }(
        amounts,
        minAmount,
        deadline
    ) > 0;
```

### Transfers of native assets with function call

Note that code snippets already checked are omitted.

From WETH9NativeWrapper.sol,

```
function _wrap(uint256 amount) internal override returns (bool) {
    IWETH9(WRAPPED_NATIVE_ASSET_CONTRACT).deposit{ value: amount }();
    return true;
}
```

transfer is made to a hardcoded trusted address.

### Transfers of native assets using `safeTransferETH`

From CoreTransfersNative.sol,

```
function _withdrawNative(uint256 amount, address to)
    internal returns (bool) {
    DefinitiveAssets.safeTransferETH(to, amount);
```

### Execption: `DefinitiveAssets`

The above does not analyze `DefinitiveAssets` even though `DefinitiveAssets` contains many functions that transfer value, since this is an underlying library, and the calls to this library with the functions of the same name are analyzed above.

## 5.3 Module: BalancerBase.sol

### Function: `_exit(uint256 lpTokenAmount, uint256[] minAmounts)`

Exit the balancer position.

### Inputs

- `lpTokenAmount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount of lpToken to unwind.
- `minAmounts`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Slippage control.

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

**Intended branches**

- Can Exit
  - ☑ Test Coverage
- Can Exit with Multical
  - ☑ Test coverage

**Negative behaviour**

- Cannot exit with more than balancer
  - ☑ Negative test
- Cannot exit if not client

☐ Negative test

## Function call analysis

- `_exit → _unstake(lpTokenAmount)`
  - **What is controllable?** Everything.
  - **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?** Nothing.
- `_exit → _unstake(lpTokenAmount) → LP_STAKING.withdraw(amount)`
  - **What is controllable?** `amount`.
  - **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?** User funds are locked.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts)`
  - **What is controllable?** `lpAmount`, `minAmount`.
  - **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?** User funds are locked.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → validateBalance(LP_TOKEN lpTokenAmount)`
  - **What is controllable?** `lpAmount`.
  - **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?** Not enough `LpToken` to remove.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → resetAndSafeIncreaseAllowance(this, LP_DEPOSIT_TOOL, lpTokenAmount)`
  - **What is controllable?** `lpTokenAmount`.
  - **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?** Was not able to reset and increase allowance.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → IVault.ExitPoolRequest({ asset, minAmountsOut, userData, toInternalBalance })`
  - **What is controllable?** `minAmountsOut`.
  - **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?** User funds are locked.

## 5.4 Module: ConvexBase.sol

### Function: `_exit(uint256 lpTokenAmount, uint256[] minAmounts)`

Exit the convex base.

### Inputs

- `lpTokenAmount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount of lpToken to unwind.
- `minAmounts`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Slippage control.

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

**Intended branches**

- Can Exit
  - ☑ Test Coverage
- Can Exit with Multical
  - ☑ Test coverage

**Negative behaviour**

- Cannot exit with more than balancer
  - ☑ Negative test
- Cannot exit if not client
  - ☐ Negative test

### Function call analysis

- `_exit → _unstake(lpTokenAmount)`
  - **What is controllable?** Everything.
  - **If return value controllable, how is it used and how can it go wrong?** Dis-

carded.
    - **What happens if it reverts, reenters, or does other unusual control flow?** Nothing.
- `_exit → _unstake(lpTokenAmount) → cfg.convexRewarder.withdrawAndUnwrap()`
    - **What is controllable?** `amount`.
    - **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?** User funds are locked.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts)`
    - **What is controllable?** `lpAmount`, `minAmount`.
    - **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?** User funds are locked.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → validateBalance(LP_TOKEN lpTokenAmount)`
    - **What is controllable?** `lpAmount`.
    - **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?** Not enough `LpToken` to remove.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → resetAndSafeIncreaseAllowance(this, LP_DEPOSIT_TOOL, lpTokenAmount)`
    - **What is controllable?** `lpTokenAmount`.
    - **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?** Was not able to reset and increase allowance.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → resetAndSafeIncreaseAllowance(this, LP_DEPOSIT_TOOL, lpTokenAmount)`
    - **What is controllable?** `lpTokenAmount`.
    - **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?** Was not able to reset and increase allowance.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → ConvexHelper.removeCurveLiquidity(LP_TOKEN, LP_DEPOSIT_POOL, lpTokenAmount, minAmounts, cfg.isMetaPool, LP_UNDERLYING_TOKENS_COUNT)`

- **What is controllable?** `lpTokenAmount`, `minAmounts`.
- **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?** User funds are locked.

## 5.5 Module: HopStrategy.sol

### Function: `_exit(uint256 lpTokenAmount, uint256[] minAmounts)`

Exit the hop position.

### Inputs

- `lpTokenAmount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount of `lpToken` to unwind.
- `minAmounts`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Slippage control.

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

**Intended branches**

- Can Exit
  - ☑ Test Coverage
- Can Exit with Multical
  - ☑ Test coverage

**Negative behaviour**

- Cannot exit with more than balancer
  - ☑ Negative test
- Cannot exit if not client
  - ☐ Negative test

### Function call analysis

- `_exit` → `_unstake(lpTokenAmount)`

- **What is controllable?** Everything.
- **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?** Nothing.

- `_exit → _unstake(lpTokenAmount) → LP_STAKING.withdrawAndHarvest(LP_STAKING_POOL_ID, amount, address(this))`
  - **What is controllable?** `amount`.
  - **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?** User funds are locked.
- `_exit → _claimAllRewards()`
  - **What is controllable?** None.
  - **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?** User funds are locked.
- `_exit → _claimAllRewards() → LP_STAKING.getReward()`
  - **What is controllable?** None.
  - **If return value controllable, how is it used and how can it go wrong?** Funds are locked.
  - **What happens if it reverts, reenters, or does other unusual control flow?** User funds are locked.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts)`
  - **What is controllable?** `lpAmount`, `minAmount`.
  - **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?** User funds are locked.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → validateBalance(LP_TOKEN lpTokenAmount)`
  - **What is controllable?** `lpAmount`.
  - **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?** Not enough `LpToken` to remove.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → resetAndSafeIncreaseAllowance(this, LP_DEPOSIT_POOL, lpTokenAmount)`
  - **What is controllable?** `lpTokenAmount`.

    - **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?** Was not able to reset and increase allowance.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → resetAndSafeIncreaseAllowance(this, LP_DEPOSIT_POOL, lpTokenAmount)`
    - **What is controllable?** `lpTokenAmount`.
    - **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?** Was not able to reset and increase allowance.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → LP_DEPOSIT_POOL.removeLiquidity(lpTokenAmount, minAmount, block.timestamp + DEFAULT_DEADLINE)`
    - **What is controllable?** `lpTokenAmount`.
    - **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?** Synapse remove liquidity reversion -> user funds are locked.

## 5.6 Module: StargateBase.sol

### Function: `_exit(uint256 lpTokenAmount, uint256[] minAmounts)`

exit a stargate position.

### Inputs

- `lpTokenAmount`
    - **Control**: Full
    - **Constraints**: None
    - **Impact**: amount of lpToken to unwind
- `minAmounts`
    - **Control**: Full
    - **Constraints**: None
    - **Impact**: Slippage Control

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

**Intended branches**

- Can Exit
  - ☑ Test Coverage
- Can Exit with Multical
  - ☑ Test coverage

**Negative behaviour**

- Cannot exit with more than balancer
  - ☑ Negative test
- Cannot exit if not client
  - ☐ Negative test

## Function call analysis

- `_exit → _unstake(lpTokenAmount)`
  - **What is controllable?**: Everything
  - **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?**: Nothing
- `_exit → _unstake(lpTokenAmount) → LP_STAKING.withdraw(LP_STAKING_POOL_ID, amount)`
  - **What is controllable?**: amount
  - **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?**: user funds are locked
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts)`
  - **What is controllable?**: lpAmount, minAmount
  - **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?**: user funds are locked
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → validateBalance(LP_TOKEN lpTokenAmount)`
  - **What is controllable?**: lpAmount
  - **If return value controllable, how is it used and how can it go wrong?**: Discared
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Not enough LpToken to remove.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → LP_DEPOSIT_POOL.i`

`nstantRedeemLocal(cfg.depositTokenPoolId, lpTokenAmount, this)`
  - **What is controllable?**: lpTokenAmount
  - **If return value controllable, how is it used and how can it go wrong?**: discard
  - **What happens if it reverts, reenters, or does other unusual control flow?**: user funds locked

## 5.7 Module: SynapseBase.sol

### Function: `_exit(uint256 lpTokenAmount, uint256[] minAmounts)`

Exit the synapse position.

### Inputs

- `lpTokenAmount`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount of `lpToken` to unwind.
- `minAmounts`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Slippage control.

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

**Intended branches**

- Can Exit
  - [x] Test Coverage
- Can Exit with Multical
  - [x] Test coverage

**Negative behaviour**

- Cannot exit with more than balancer
  - [x] Negative test
- Cannot exit if not client
  - [ ] Negative test

## Function call analysis

- `_exit → _unstake(lpTokenAmount)`
  - **What is controllable?** Everything.
  - **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?** Nothing.
- `_exit → _unstake(lpTokenAmount) → LP_STAKING.withdrawAndHarvest(LP_STAKING_POOL_ID, amount, address(this))`
  - **What is controllable?** `amount`.
  - **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?** User funds are locked.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts)`
  - **What is controllable?** `lpAmount`, `minAmount`.
  - **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?** User funds are locked.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → validateBalance(LP_TOKEN lpTokenAmount)`
  - **What is controllable?** `lpAmount`.
  - **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?** Not enough `LpToken` to remove.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → resetAndSafeIncreaseAllowance(this, LP_DEPOSIT_TOOL, lpTokenAmount)`
  - **What is controllable?** `lpTokenAmount`.
  - **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?** Was not able to reset and increase allowance.
- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → resetAndSafeIncreaseAllowance(this, LP_DEPOSIT_TOOL, lpTokenAmount)`
  - **What is controllable?** `lpTokenAmount`.
  - **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?** Was not able to reset and increase allowance.

- `_exit → _removeLiquidity(lpTokenAmount, minAmounts) → LP_DEPOSIT_POOL.removeLiquidity(lpTokenAmount, minAmount, block.timestamp + DEFAULT_DEADLINE)`
  - **What is controllable?** `lpTokenAmount`.
  - **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?** Synapse remove liquidity reversion -> user funds are locked.

# 6 Audit Results

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

During our audit, we discovered five findings. Of these, three were high impact, one was medium impact, and one was low impact. Definitive acknowledged all findings and implemented fixes.

## 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.