Zellic

Fractal Protocol

Smart Contract Security Assessment

April 3, 2023

Prepared for:

Alexandre Elkrief

Fractal

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 Fractal from February 1st to February 16th, 2023. During this engagement, Zellic reviewed Fractal Protocol'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:

- What are the dynamics of lenders and borrowers in openTermLoans, and how can they potentially steal capital or DOS the contract?
- How do the strategies function, and is there a risk of them being forced to take bad trades that could result in loss of funds?
- What are the on-chain invariants and validation measures that need to be considered in light of the large amount of off-chain decision making?

1.2 Non-goals and Limitations

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

- Problems relating to the front-end components and infrastructure of the project
- Problems due to improper key custody or off-chain access control that would result in the compromise of the main operators
- Issues stemming from code or infrastructure outside of the assessment scope
- Liquidations and margin calls managed by the off-chain risk management engine

Security assessments are timeboxed by nature and therefore have limitations in terms of the coverage they can provide. During our assessment of the protocol, we noted areas where the GLP price is determined by means of an on-chain AUM measure. However, due to the time and resource constraints of the assessment as well as the project scope, we were unable to thoroughly examine this aspect of the protocol.

Moreover, during the remmediation phase of the engagement, some extra features have been added to the protocol. These features were not included in the scope of the assessment and therefore were not assessed for security. Thus, from the commits that are mentioned in each of the findings' remmediation sections, we have only focused on the edits related to the remmediation of the finding itself, rather than the entire commit.

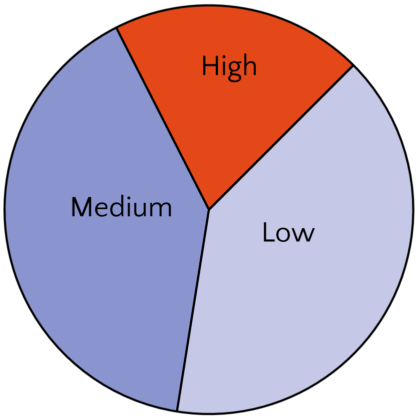
1.3 Results

During our assessment on the scoped Fractal Protocol contracts, we discovered five findings. No critical issues were found. Of the five findings, one was of high impact, two were of medium impact, and the remaining findings were low in nature.

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

Breakdown of Finding Impacts

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



2 Introduction

2.1 About Fractal Protocol

Fractal gives institutions the tools they need to seamlessly interact with the on-chain finance ecosystem across prime brokers, OTC desks, lenders, custodians, and exchanges — for institutional-grade capital efficiency.

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:

Fractal Protocol Contracts

Repository	https://github.com/fractal-protocol/f-strategy-contracts https://github.com/fractal-protocol/f-smart-contracts
Versions	f-smart-contracts:6b895a2e7a82d712d6bc4534ad5ab73bb6707f3f f-strategy-contracts:4aaba8b8db1f87ecb07753ea26711831eae9cdf2
Contracts	• AddressWhitelist • BaseProvider • CounterPartyRegistry • CrosschainProviderConfigManager • CurveConvexRegistry • CurveConvexWallet • CustomControllable • CustomInitializable • CustomInitializable • CustomOwnable • ForeignVault • FractAaveConvexFraxUsdc • FractAaveV3Strategy • FractBaseStrategy • FractCompoundStrategy • FractCrossProtocolStrategy • FractFraxConvex • FractFraxConvexAlusdBp • FractFraxConvexTusdBp • FractGmxAvax • FractMoonwellStrategy • FractStableswap • FractTusdFraxBp • GlpSpotMarginAccount • Initializable • LayerOneProvider • LoansManager • MultichainProvider • OpenTermLoan • PriceOracle • ProxyAdmin

.	• ProxyFactory • ReceiptToken • SpotMarginAccount • SubAccount • SwapContractManager • TokenMigration • TotalReturnSwapContract • TwapOrder • Vault • YieldReserve
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 4.5 person-weeks. The assessment was conducted over the course of three calendar weeks.

Contact Information

The following project managers were 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.

February 1, 2022	Start of primary review period
February 16, 2022	End of primary review period

3 Detailed Findings

3.1 Borrower cannot withdraw funds

- **Target**: OpenTermLoan
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: High

Description

In certain situations where the price drops below the maintenance collateral ratio, the borrower is unable to withdraw the principal and activate the loan.

```
function withdraw () public onlyBorrower {
    require(loanState == FUNDED, "Invalid loan state");

    // Enforce the maintenance collateral ratio, if applicable
    _enforceMaintenanceRatio();

    loanState = ACTIVE;

    // Send principal tokens to the borrower
    _withdrawPrincipalTokens(_effectiveLoanAmount, borrower);

    // Emit the event
    emit OnBorrowerWithdrawal(_effectiveLoanAmount);
}
```

Impact

The borrower is not able to withdraw their funds. The only way for the borrower to regain access to their funds is for the lender to call the loan and return the funds.

Recommendations

Remove `_enforceMaintenanceRatio` in `withdraw`.

Remediation

Fractal has addressed the issue by implementing a fix in commit 4888d6b6 which removes the `_enforceMaintenanceRatio` call in `withdraw`.

3.2 Initialize function can be called multiple times

- **Target**: GlpSpotMarginAccount
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Medium
- **Impact**: Medium

Description

The `initialize` function can be called multiple times.

```
function initializeSubAccount(
    address loanAddress,
    address feeCollectorAddress,
    address paraswapAddress,
    address tokenTransferProxyAddress,
    uint256 feeAmount) public onlyOperator
{
    // @audit shouldn't be callable more than once.
    _loanContract = loanAddress;
    _feeCollector = feeCollectorAddress;
    _paraswap = paraswapAddress;
    _tokenTransferProxy = tokenTransferProxyAddress;
    _feeBips = feeAmount;
}
```

Impact

This can lead to unexpected behavior, since the state variable changes will break the logic of the contract.

The impact of this finding is diminished by the restriction that only the operator has the authority to invoke this function.

Recommendations

We recommend adding a check to ensure that the function is not called more than once, such as using OpenZeppelin's `initializer` modifier.

Remediation

Fractal has addressed the issue by implementing a fix in commit d463725e through the use of the `initializer` modifier. The function has also been renamed to `initialize` to match the naming convention of the other contracts.

3.3 Insufficient slippage protection

- **Target**: Project Wide
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The codebase has several areas where slippage checks are either absent or insufficient to guard against MEV attacks. Some of these areas are listed below.

All instances of `depositCurveConvex` in the various strategies:

```
function depositCurveConvex(
    uint256 amount,
    uint256[3] calldata amounts,
    uint256 slippageBips,
    address proxyVault) public onlyOperator
{
     ...
    uint256 calcAmount
    = ICurveSwap(FRAXZAPPOOL).calc_token_amount(ALUSDFRAXPOOL, amounts,
    true);

    uint256 minAmount = (calcAmount * (BIPS_DIVISOR - slippageBips))
    / BIPS_DIVISOR;

    ICurveSwap(FRAXZAPPOOL).add_liquidity(ALUSDFRAXPOOL, amounts,
    minAmount);
}
```

In this instance, minimum amounts are calculated from on-chain prices that will have already been skewed; therefore, the `add_liquidity` operation will always pass.

All instances of `withdrawCurveConvex` in the various strategies:

```
function withdrawCurveConvex(
    bytes32 kek_id,
    address proxyVault,
    uint256 slippageBips) public onlyOperator
{
    ...
    uint256 calcAmount
    = ICurveSwap(FRAXZAPPOOL).calc_withdraw_one_coin(ALUSDFRAXPOOL,
    lpTokenBalance, 2);

    uint256 minAmount = (calcAmount * (BIPS_DIVISOR - slippageBips))
    / BIPS_DIVISOR;

    ICurveSwap(FRAXZAPPOOL).remove_liquidity_one_coin(ALUSDFRAXPOOL,
    lpTokenBalance, 2, minAmount);
    ...
}
```

In this instance, minimum amounts are calculated from on-chain prices that will have already been skewed; therefore, the `remove_liquidity_one_coin` operation will always pass.

This also applies to `addLiquidityAndDeposit` and `removeLiquidityAndWithdraw` in FractStableSwap.

The same issue is present for the `minUsdcAmount` function in `GlpUnwind`.

Then there are slippage issues in borrow pools where the manipulation of borrow pools can result in less than expected tokens. In the Aave strategies, these exchanges are protected by a min amount number. However, in the Compound strategies, there is no minimum check for borrowed tokens received.

```
function mintAndBorrow(
    address mintAddress,
    address borrowAddress,
    uint256 mintAmount,
    uint256 collateralFactorBips
) public onlyOperator
{
    require(mintAddress ≠ address(0), "0 Address");
    require(borrowAddress ≠ address(0), "0 Address");
    require(mintAmount > 0, "Mint failed");
    require(collateralFactorBips ≤ BIPS_DIVISOR, "Collateral factor");

    _mint(mintAddress, mintAmount);
    _borrow(borrowAddress, collateralFactorBips);
}
```

In FractAaveConvexFraxUsdc, the interface for the CRVFRAXPOOL is incorrect and a timestamp is supplied instead of a min amount.

```
ICurveSwap(CRVFRAXPOOL).add_liquidity(amounts, block.timestamp + 10);
    //@audit timestamp supplied for min amount
```

In FractMoonwellStrategy.sol, in the `harvestByMarket` function, both swaps pass a 0 for the min amount out.

```
function harvestByMarket(address mintAddress, address borrowAddress)
public onlyOperator {
     ...
    _swapTokens(MOONWELL_TOKEN, underlyingAddress, rewardBalance, 0);
    //@audit pass a 0 for min amount out
    _swapNativeToken(underlyingAddress, movrBalance, 0); //@audit pass a 0
    for min amount out
}
```

Impact

Insufficient slippage protection can result in the loss of funds of users.

Recommendations

Our recommendation is to incorporate minimum amount arguments obtained from off-chain sources and verify that the slippage is within acceptable limits.

Remediation

Fractal has addressed the issue by implementing fixes in commit 5da58e4c8 by the addition of a minimum amount parameter in every impacted function.

3.4 State variables are shadowed by function parameters

- **Target**: FractCompoundStrategy, SwapContractManager
- **Category**: Coding Mistakes
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Description

Two state variables are shadowed by function parameters. This applies for `counterPartyRegistry` in SwapContractManager and `comptroller` in FractCompoundStrategy.

```
constructor(
    address feeCollectorAddr,
    address counterPartyRegistryAddr
){
    require(feeCollectorAddr != address(0), '0 address');
    require(counterPartyRegistryAddr != address(0), '0 address');

    feeCollector = feeCollectorAddr;
    counterPartyRegistry = counterPartyRegistryAddr;
}


function deployTotalReturnSwapContract(
    uint8 direction,
    address operator,
    address counterPartyRegistry,
    // ...
```

Impact

This can lead to unexpected behavior, since the state variables will be shadowed by the function parameters.

Recommendations

We recommend opting for a different name for the function parameters, or removing the state variables.

Remediation

Fractal has addressed the issue by implementing a fix in commit 66ef7c87f and dc471fdd by removing and/or renaming the local variables.

3.5 Potential lockup of funds in the event of insufficient AnySwap liquidity

- **Target**: MultichainProvider
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

In the edge case in which a large multichain transfer is done through the anySwap bridge and there is not enough liquidity, anySwap token placeholders will be supplied that can be redeemed for the underlying token once the bridge accumulates enough liquidity.

```
    function executeTransfer (IERC20 underlyingTokenInterface,
    address destinationAddr, uint256 transferAmount,
    bytes32 foreignNetwork) public override onlyIfWhitelistedSender {
     ...
     // Run the crosschain transfer
    IAnyswapV5Router(routerAddress).anySwapOutUnderlying(anyTokenAddress,
    destinationAddr, transferAmount, destinationChainId);
     ...
}
```

However, there is no mechanism to redeem or transfer these funds in the `YieldReserve`.

Impact

Funds can potentially be locked.

Recommendations

Add a mechanism to transfer/redeem these placeholder tokens.

Remediation

This issue has been acknowledged by Fractal.

4 Discussion

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

4.1 Test coverage and negative testing

Test coverage is a critical metric that measures the extent to which tests have been written for a codebase. The current coverage for the codebase is 29.81% line coverage and 16.08% branch coverage. To ensure that the code functions as intended, both positive and negative tests are essential. During our security assessment, we identified several areas of the code that were not adequately covered by tests. As a result, certain assumptions had to be made regarding the code's functionality and usage. These assumptions are detailed in the Threat Model section 5 of the report.

To mitigate any potential issues and ensure that the assumptions are well-founded, it is important that the developers create tests that cover these areas. We strongly recommend that the existing tests be improved to encompass a more comprehensive portion of the code. By implementing these measures, we can have greater confidence in the reliability and security of the code and reduce the risk of potential vulnerabilities.

4.2 Oracles in OpenTermLoans

When an open term loan is deployed, the oracles are agreed upon initially. However, the lender can change the oracle at any point during the loan term. This creates an opportunity for a lender who has been compromised or is malicious to steal funds using various methods:

- If a borrower has not yet withdrawn their funds, a malicious lender could change the oracle to a fake one, causing the borrower to be unable to withdraw their funds due to `_enforceMaintenanceRatio`. Subsequently, the lender could call the loan and liquidate both the unclaimed principal and the collateral.
- If a borrower has already withdrawn funds, the malicious lender could wait until the borrower partially repays (e.g., 90%) before changing the oracle to a fake one. The fake oracle would prevent the borrower from paying back their interest due to failing `_enforceMaintenanceRatio` calls. The lender could then call the loan, eventually claiming a large portion (up to 99.9%) of the principal and collateral.

An ideal solution would be a system where both the lender and borrower agree upon changes to the oracle.

4.3 Paraswap off-chain payload

Almost all the instances of swaps in are carried out using Paraswap. PARASWAP is usually called as such

```
(bool success,) = PARASWAP.call(callData);
```

The byte-encoded payload supplied from off-chain sources is the sole parameter passed to Paraswap. This can pose a risk due to the potential for reentrancy if flashloans are callable, or even just extremely high gas consumption. Therefore, it is essential to understand the formation of the off-chain payload and the potential ways in which a user can manipulate it to carry out malicious actions.

The Fractal team commented on this as such:

> "The off-chain payload is composed by one of our APIs as opposed to being created by a user via Web3/browser in a standalone manner."

4.4 Equality checks in OpenTermLoans

Several areas in the codebase use exact equality checks (`==`), which can have consequences if reentrancy is possible in OpenTermLoan, such as in the case of ERC777 tokens (ERC20 tokens with hooks).

For instance, if an ERC777 token is used as the underlying token of the contract, a borrower who wants to pay back their loan and close it could be denied by a malicious lender. This can happen during the token transfer hooks in `_transferPrincipalAndCollateral`. When code execution is passed to the lender during an ERC777 transfer, the lender can send back one wei of the token to OpenTermLoan, causing the subsequent equality check to fail and resulting in the transaction being reverted.

```
function _transferPrincipalAndCollateral (address
    collateralRecipientAddr, address principalRecipientAddr)
    private {
     ...
    principalTokenInterface.transfer(principalRecipientAddr,
    principalBalanceInTokens);
    require(principalTokenInterface.balanceOf(address(this)) == 0,
    "Principal transfer failed");
     ...
}
```

4.5 Reentrancy

Fractal has implemented their own reentrancy guards instead of relying on the OpenZeppelin reentrancy guards. However, it appears that there are limited reentrancy points when standard ERC20 tokens are used (tokens without hooks/not ERC777 tokens).

Moreover, the placement of `nonReentrant` checks seems to be haphazard, with some functions that alter state lacking reentrancy protection. For instance, the `withdraw` function in OpenTermLoans does not have reentrancy protection, despite being a state-altering function.

We recommend using the standard OpenZeppelin `nonReentrant` modifier and applying it on all state-changing functions.

4.6 Decimals have to be either 6 or 18

In FractAaveV3Strategy, there is a presumption that the tokens used are going to be either 6 or 18 decimals.

```
function _adjustValue(address depositToken, uint256 amount)
    internal view returns (uint256)
{
    uint8 aTokenDecimals = 8;
    uint8 depositTokenDecimals = IERC20(depositToken).decimals();
    if (aTokenDecimals > depositTokenDecimals) {
        uint256 decreaseValue = amount/100;
        return decreaseValue;
    }
    if (aTokenDecimals < depositTokenDecimals) {
        uint256 increaseValue = amount*(10**10);
        return increaseValue;
    } else {
        return amount;
    }
}
```

This function adjusts for the value of a token and assumes that the `depositTokens` are either 6 decimals if smaller than 8 decimals or 18 decimals if larger than 8 decimals. The Fractal team assured us that only 6 or 18 decimal tokens will be used. Nonetheless, it is still important to note this down as this can be the cause of future bugs if unaccounted for.

4.7 Functions that should not be called twice

There are some functions that should theoretically only be called once. One such example is the `setVaultAddress` function in YieldReserve, which sets the vault's address to which the strategy is linked. Invoking this function more than once could result in the strategy being connected to multiple vaults, which is not the intended behavior and may result in unforeseen consequences. Similarly, the `setLoanContract` function in SubAccount is another example of a function that should be called only once.

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 File: AddressWhitelist.sol

Function: `disableAddress(address addr)`

Should allow owner to remove from whitelist.

Inputs

- `addr`:
  - **Control**: Fully controlled by owner.
  - **Constraints**: Should be already whitelisted.
  - **Impact**: Address will be removed from whitelist.

Branches and code coverage (including function calls)

Intended branches

- Change the `whitelistedAddresses` state of the address to `false`.
  ☑ Test coverage

Negative behavior

- Should revert if the address is not whitelisted.
  ☑ Negative test
- Should not be callable by anyone other than the owner.
  ☑ Negative test

Function: `enableAddress(address addr)`

Should allow owner to whitelist an address.

Inputs

- `addr`:
  - **Control**: Fully controlled by owner.
  - **Constraints**: Should not be already whitelisted.
  - **Impact**: Address will be whitelisted.

Branches and code coverage (including function calls)

Intended branches

- Change the `whitelistedAddresses` state of the address to `true`.
  ☑ Test coverage

Negative behaviour

- Should revert if the address is already whitelisted.
  ☑ Negative test
- Should not be callable by anyone other than the owner.
  ☑ Negative test

5.2 File: CustomControllable.sol

Function: `setController(address controllerAddr)`

Sets the controller address.

Inputs

- `controllerAddr`:
  - **Control**: Full control.
  - **Constraints**: Checked against 0, current controller, and owner.
  - **Impact**: The new controllerAddress.

Branches and code coverage (including function calls)

Intended branches

- Update the controllerAddress.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.

☑ Negative test

5.3 File: FractAaveConvexFraxUsdc.sol

Function: `depositCurveConvex(uint256 amount, uint256[Literal(value=2.0, unit=None)] amounts)`

Add liquidity to curve and deposit into Convex.

Inputs

- `amount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: The amount to be deposited into the curve pool.
- `amounts`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: The amounts of curve LP tokens to deposit.

Branches and code coverage (including function calls)

Intended branches

- Assumes the approve succeeds.
  ☑ Test coverage
- Assumes the add liquidity succeeds.
  ☑ Test coverage
- Should decrease the USDC balance.
  ☐ Test coverage
- Should decrease the CRVFRAX balance.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone else than the operator.
  ☑ Negative test

Function call analysis

- `IERC20(USDC).approve(CRVFRAXPOOL, amount)`:
- **What is controllable?**: `amount`.

  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.
- `ICurveSwap(CRVFRAXPOOL).add_liquidity(amounts, block.timestamp + 10)`:
- **What is controllable?**: `amounts`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.
- `IERC20(CRVFRAX).approve(CONVEXBOOSTER, balance)`:
- **What is controllable?**: `balance`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.
- `IBooster(CONVEXBOOSTER).depositAll(100, true)`:
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Depositing into Convex fails.

Function: `harvestRewards(uint256[Literal(value=2.0, unit=None)] minAmounts)`

Harvest all pending rewards on Convex.

Inputs

- `minAmounts`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Should be two elements: 0 = CRV, 1 = CVX.
  - **Impact**: The slippage limits on the swap.

Branches and code coverage (including function calls)

Intended branches

- Increase the Convex rewards balance.
  ☐ Test coverage
- Assumes the paths are legitimate for the swap router.
  ☑ Test coverage
- Assumes the slippage is within the acceptable range.

☑ Test coverage

Negative behavior

- Ensure that it is not front-runnable; maybe call it often with low rewards.
  - ☐ Negative test

Function call analysis

- `IBaseRewardPool(REWARDPOOL).getReward(address(this), true)`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Harvest fails.
- `swapRouter.swapExactTokensForTokens(crvBalance, minAmounts[0], path, address(this), block.timestamp + 10)`:
- **What is controllable?**: minAmounts[0].
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Could fail due to slippage.
- `swapRouter.swapExactTokensForTokens(cvxBalance, minAmounts[1], path, address(this), block.timestamp + 10)`:
  - **What is controllable?**: minAmounts[1].
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Could fail due to slippage.

Function: `rebalance(uint256 withdrawAmount)`

Rebalance position on AAVE if health factor drops too low.

Inputs

- `withdrawAmount`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The amount of LP tokens to withdraw from Convex.

Branches and code coverage (including function calls)

Intended branches

- Rebalance if the health factor is too low.
  - ☐ Test coverage
- Withdraw from Convex.
  - ☑ Test coverage
- Repay to AAVE.
  - ☑ Test coverage

Negative behavior

- Should only be called by the operator.
  - ☑ Negative test

Function call analysis

- `IBaseRewardPool(REWARDPOOL).withdrawAndUnwrap(withdrawAmount, true)`:

- **What is controllable?**: `withdrawAmount`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Entire function reverts.
- `ICurveSwap(CRVFRAXPOOL).calc_withdraw_one_coin(lpTokenBalance, 1)`:

- **What is controllable?**: `lpTokenBalance` to some extent.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Entire function reverts
- `ICurveSwap(CRVFRAXPOOL).remove_liquidity_one_coin(lpTokenBalance, 1, minAmount)`:
  - **What is controllable?**: `lpTokenBalance` to some extent.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Entire function reverts.

Function: `withdrawCurveConvex(uint256 slippageBips)`

Withdraw all from Convex and Curve.

Inputs

- `slippageBips`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: n/a
  - **Impact**: The slippage tolerance in bips.

Branches and code coverage (including function calls)

Intended branches

- Assumes entire balances are withdrawn.
  - ☑ Test coverage
- The USDC balance should increase after the withdrawal.
  - □ Test coverage
- Assumes the slippage is within the acceptable range.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone else than the operator.
  - ☑ Negative test

Function call analysis

- `IBaseRewardPool(REWARDPOOL).withdrawAllAndUnwrap(true)`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Entire withdrawal is reverted.
- `ICurveSwap(CRVFRAXPOOL).calc_withdraw_one_coin(lpTokenBalance, 1)`:
- **What is controllable?**: n/a
  - **If return value controllable, how is it used and how can it go wrong?**: returns calculated amount of USDC to withdraw; not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: n/a
- `ICurveSwap(CRVFRAXPOOL).remove_liquidity_one_coin(lpTokenBalance, 1, minAmount)`:
  - **What is controllable?**: n/a
  - **If return value controllable, how is it used and how can it go wrong?**: n/a
  - **What happens if it reverts, reenters, or does other unusual control flow?**: reverts entire withdrawal if slippage is too high.

5.4 File: FractAaveV3Strategy.sol

Function: `claimRewardsByMarket(address aToken, address aDebtToken)`

Allows operator to claim the rewards from the incentives controller.

Inputs

- `aToken`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Part of the market parameters.
- `aDebtToken`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**:Ppart of the market parameters.

Branches and code coverage (including function calls)

Intended branches

- Should have approved the `depositToken` to the AAVE lending pool beforehand.
  ☑ Test coverage
- Assumes that rewards can be withdrawn from this contract somehow.
  ☐ Test coverage
- Should increase the rewards balance of the strategy.
  ☐ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  ☑ Negative test

Function call analysis

- `incentivesController.claimAllRewards(markets, address(this))`:
  - **What is controllable?**: `markets`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the claim fails.

Function: `deposit(IERC20 token, uint256 amount)`

Allows the owner to deposit tokens in the strategy.

Inputs

- `token`:
  - **Control**: Fully controlled by the owner.

  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will increase.
- `amount`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will increase by the amount deposited.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  ☑ Test coverage
- Assumes that the token will eventually be withdrawable from this contract.
  ☑ Test coverage

Negative behavior

- Should not assume that only the owner can transfer tokens to this contract. But transfers can happen manually too.
  ☑ Negative test

Function call analysis

- `token.transferFrom(msg.sender, address(this), amount)`:
  - **What is controllable?**: `msg.sender` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `repayTokens(address depositToken, uint256 repayAmount)`

Should repay tokens on AAVE.

Inputs

- `depositToken`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes this is the token that the strategy is supposed to be using.
  - **Impact**: Token balance of the strategy will decrease by the repaid amount.
- `repayAmount`:

  - **Control**: Fully controlled by the operator.
  - **Constraints**: assumes that the amount is <= than what can be repaid.
  - **Impact**: Token balance of the strategy will decrease by the repaid amount.

Branches and code coverage (including function calls)

Intended branches

- Should have approved the `depositToken` to the AAVE lending pool beforehand.
  ☐ Test coverage
- Will repay the `depositToken` to the AAVE lending pool.
  ☑ Test coverage
- Assumes there is something to repay.
  ☑ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  ☑ Negative test
- Should not be called if the amount is 0.
  ☐ Negative test

Function call analysis

- `lendingPool.repay(depositToken, repayAmount, 2, address(this))`:
  - **What is controllable?**: `repayAmount` and `depositToken`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the repayment fails.

Function: `resetPosition(address aToken, address depositToken)`

Resets the position.

Inputs

- `aToken`:
  - **Control**: Full.
  - **Constraints**: Should be != 0.
  - **Impact**: The aToken to reset the position for.
- `depositToken`:
  - **Control**: Full.
  - **Constraints**: Should be != 0.

  - **Impact**: The depositToken used in strategy.

Branches and code coverage (including function calls)

Intended branches

- Checked.
  ☑ Test coverage
- Unchecked.
  □ Test coverage

Negative behavior

- Checked.
  ☑ Negative test
- Unchecked.
  □ Negative test

Function call analysis

- `lendingPool.repayWithATokens(depositToken, type(uint256).max, 2)`:
  - **What is controllable?**: `depositToken`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the repay fails.

Function: `setUserEMode(uint8 categoryId)`

Sets the user efficiency mode category for the lending pool.

Inputs

- `categoryId`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: It is checked at `lendingPool` level.
  - **Impact**: Sets the user eMode.

Branches and code coverage (including function calls)

Intended branches

- `categoryID` should be valid.
  □ Test coverage
- Should set the user eMode.

  - □ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  - □ Negative test

Function call analysis

- `lendingPool.setUserEMode(categoryId)`:
  - **What is controllable?**: `categoryId`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the eMode fails.

Function: `supplyAndLeverUp(address depositToken, uint256 amount, uint256 leverageLevel, uint256 minMint, uint256 ltv)`

Should supply tokens to AAVE and lever up the position.

Inputs

- `depositToken`
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes this is the token that the strategy is supposed to be using.
  - **Impact**: Token balance of the strategy will decrease by the supplied amount.
- `amount`
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes that the amount is <= than what can be supplied.
  - **Impact**: Token balance of the strategy will decrease by the supplied amount.
- `leverageLevel`
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: N/A.
- `minMint`
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Checks the adjusted amount is >= minMint.
  - **Impact**: The minimum amount of tokens that will be minted.
- `ltv`
  - **Control**: Fully controlled by the operator.

  - **Constraints**: N/A.
  - **Impact**: Used to calculate the leverage.

Branches and code coverage (including function calls)

Intended branches

- Should have approved the `depositToken` to the AAVE lending pool beforehand.
  ☑ Test coverage
- Will deposit the `depositToken` to the AAVE lending pool.
  ☑ Test coverage
- Ensure, based on off-chain calculations, that the whole operation will not revert, otherwise it will be gas inefficient.
  □ Test coverage
- Will set the leverage to the desired level.
  ☑ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  ☑ Negative test
- Should not be called if the amount is 0.
  □ Negative test

Function call analysis

- `supplyTokens(depositToken, amount)`:
- **What is controllable?**: `amount` and `depositToken`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the supply fails.
- `lendingPool.borrow(deposToken, adjustBorrowable, 2, 0, address(this))`:
- **What is controllable?**: `depositToken`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the borrow fails.
- `lendingPool.supply(depositToken, amount, address(this), 0)`:
  - **What is controllable?**: `amount` and `depositToken`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the supply fails.

Function: `supplyTokens(address depositToken, uint256 amount)`

Should supply tokens to AAVE.

Inputs

- `depositToken`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes this is the token that the strategy is supposed to be using.
  - **Impact**: Token balance of the strategy will decrease by the supplied amount.
- `amount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes that the amount is <= than what can be supplied.
  - **Impact**: Token balance of the strategy will decrease by the supplied amount.

Branches and code coverage (including function calls)

Intended branches

- Will approve the AAVE lending pool max uint256 amount of the `depositToken`.
  ☐ Test coverage
- Will supply the `depositToken` to the AAVE lending pool.
  ☐ Test coverage
- Should receive `aToken` in return.
  ☐ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  ☑ Negative test
- Should not be called if the amount is 0.
  ☐ Negative test
- Should not be called if the `depositToken` is 0.
  ☐ Negative test

Function call analysis

- `IERC20(depositToken).approve(address(lendingPool), type(uint256).max)`:
- **What is controllable?**: `depositToken`.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns whether the approval was successful or not.

  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.
- `lendingPool.supply(depositToken, amount, address(this), 0)`:
  - **What is controllable?**: `amount` and `depositToken`.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the amount of tokens that have been supplied.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the supply fails.

Function: `swap(IERC20 token, uint256 amount, byte[] callData)`

Should swap the rewards tokens for the underlying token.

Inputs

- `token`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes the Paraswap router will accept the token.
  - **Impact**: Token balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Token balance of the strategy will decrease.
- `callData`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Dictates how the routing will be done in Paraswap. Assumes it is validated off chain.

Branches and code coverage (including function calls)

Intended branches

- Assumes the token is the one that the strategy is supposed to be using.
  ☐ Test coverage
- Assumes the amount is less than the balance of the strategy.
  ☐ Test coverage
- Assumes the `callData` is valid.
  ☑ Test coverage
- Decreases the balance of the strategy by the amount.
  ☐ Test coverage

- Increases the balance of the underlying token by the amount. This should be checked to ensure the validity of the `callData` itself.
  - ☐ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  - ☑ Negative test
- Should not be called if the token is not the one that the strategy is supposed to be using.
  - ☑ Negative test

Function call analysis

- `getTokenTransferProxy()`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the token transfer proxy for the Paraswap router.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A; it should revert altogether.
- `token.approve(tokenTransferProxy, amount)`:
- **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the approval was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.
- `PARASWAP.call(callData)`:
- **What is controllable?**: `callData`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the swap was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the swap fails.
- `token.approve(tokenTransferProxy, 0)`:
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the approval was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.

Function: `withdrawToOwner(IERC20 token, uint256 amount)`

Allows the operator to withdraw tokens from the strategy for the owner.

Inputs

- `token`:
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will decrease by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  - ☐ Test coverage
- The balance of the strategy is assumed to be greater than the amount to withdraw.
  - ☐ Test coverage
- Decreases the balance of the strategy by the amount withdrawn.
  - ☐ Test coverage
- Increases the balance of the owner by the amount withdrawn.
  - ☐ Test coverage

Negative behaviour

- Should not be called by anyone other than the operator.
  - ☑ Negative test

Function call analysis

- `token.transfer(owner, amount)`:
  - **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `withdrawTokens(address depositToken, uint256 amount)`

Should withdraw tokens from AAVE.

Inputs

- `depositToken`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes that this token has been deposited into AAVE beforehand.
  - **Impact**: Token balance of the strategy will increase by the withdrawn amount.
- `amount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes that the amount is <= than what can be withdrawn.
  - **Impact**: Token balance of the strategy will increase by the withdrawn amount.

Branches and code coverage (including function calls)

Intended branches

- Assumes no approval is needed to withdraw from AAVE.
  - ☐ Test coverage
- Burn `aToken` balance of the strategy. Assumes there is enough balance to burn.
  - ☐ Test coverage
- Increases the `depositToken` balance of the strategy by the amount.
  - ☐ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  - ☑ Negative test
- Should not be called if the amount is 0.
  - ☐ Negative test
- Should not be called if the depositToken is 0.
  - ☐ Negative test

Function call analysis

- `lendingPool.withdraw(depositToken, amount, address(this))`:
  - **What is controllable?**: `amount` and `depositToken`.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the amount of tokens that have been withdrawn.
  - **What happens if it reverts, reenters, or does other unusual control flow?**:

Reverts if the withdrawal fails.

Function: `withdraw(IERC20 token, uint256 amount)`

Allows the owner to withdraw tokens from the strategy.

Inputs

- `token`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will decrease by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  ☑ Test coverage
- The balance of the strategy is assumed to be greater than the amount to withdraw.
  □ Test coverage
- Decreases the balance of the strategy by the amount withdrawn.
  □ Test coverage
- Increases the balance of the `msg.sender` by the amount withdrawn.
  □ Test coverage

Negative behavior

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

Function call analysis

- `token.transfer(msg.sender, amount)`:
  - **What is controllable?**: `msg.sender` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**:

Reverts if the transfer fails.

5.5 File: FractBaseStrategy.sol

Function: `setOperatorAddress(address operatorAddress)`

Allows the owner to set the operator address.

Inputs

- `operatorAddress`:
  - **Control**: Fully controlled by owner.
  - **Constraints**: None.
  - **Impact**: Sets the `_operator` variable.

Branches and code coverage (including function calls)

Intended branches

- Should set the `_operator` variable to the `operatorAddress`.
  ☑ Test coverage

Negative behavior

- Shoud revert if `msg.sender` is not the owner.
  ☑ Negative test

Function: `withdrawETH(uint256 amount)`

Allows the owner to withdraw ETH from the contract.

Inputs

- `amount`
  - **Control**: Fully controlled by owner.
  - **Constraints**: Assumes the contract has enough ETH to send.
  - **Impact**: Sends ETH to the owner.

Branches and code coverage (including function calls)

Intended branches

- Should send ETH to the owner. Basically, the balance of the contract should be reduced by the amount.

  - □ Test coverage
- Balance of the owner should be increased by the amount.
  - □ Test coverage

Negative behavior

- Shoud revert if `msg.sender` is not the owner.
  - ☑ Negative test

5.6 File: FractCompoundStrategy.sol

Function: `mintAndBorrow(address mintAddress, address borrowAddress, uint256 mintAmount, uint256 collateralFactorBips)`

ALlow operator to mint and borrow from `cTokens`.

Inputs

- `mintAddress`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: The address must be a valid `cToken` address.
  - **Impact**: The operator can mint on any `cToken` market.
- `borrowAddress`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: The address must be a valid `cToken` address.
  - **Impact**: The operator can borrow on any `cToken` market.
- `mintAmount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: The amount must be greater than 0.
  - **Impact**: The operator can mint any amount of underlying token.
- `collateralFactorBips`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: The amount must be less than or equal to 10,000.
  - **Impact**: The operator can borrow any amount of underlying token.

Branches and code coverage (including function calls)

Intended branches

- Transfer underlying token from contract to `cToken`.
  - ☑ Test coverage

- Increase balance of `cToken` in contract.
  ☑ Test coverage
- Assumes the exchange rate from `cToken` is not altered and cannot be altered.
  □ Test coverage
- Increases the borrowed asset balance of the contract.
  ☑ Test coverage
- Assumes the contract currently holds the necessary underlying token required for mint.
  ☑ Test coverage
- Should have a way to exit the market (e.g., calling `comptroller.exitMarket`).
  ☑ Test coverage

Negative behavior

- Should not allow anyone else than the operator to mint and borrow.
  ☑ Negative test
- Should not allow operations on 0 address.
  □ Negative test

Function call analysis

- `cErcToken.mint(mintAmount)`:

- **What is controllable?**: `mintAmount`.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value tells us if the operation is successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Funds are not minted. Should revert.
- `comptroller.enterMarkets(markets)`:

- **What is controllable?**: `cTokens`
  - **If return value controllable, how is it used and how can it go wrong?**: The return value tells us if the operation is successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Means the market is not entered. Should revert.
- `cErcToken.borrow(borrowAmount)`:
  - **What is controllable?**: `borrowAmount`, partly.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value tells us if the operation is successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Funds are not borrowed. Should revert.

Function: `redeemTokens(address mintAddress, address borrowAddress)`

Allow the operator to redeem tokens after all debt has been paid off.

Inputs

- `mintAddress`
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Must be a valid `cToken` address.
  - **Impact**: The address of the `cToken` to redeem.
- `borrowAddress`
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Must be a valid `cToken` address.
  - **Impact**: The address of the `cToken` to check debt against.

Branches and code coverage (including function calls)

Intended branches

- Allow operator to redeem `cTokens`.
  ☑ Test coverage
- Increase the underlying balance attributed to that ctoken of the contract.
  ☑ Test coverage

Negative behavior

- Should not be able to redeem if there is still debt.
  ☑ Negative test

Function call analysis

- `cErcToken.redeem(cTokenBalance)`:
- **What is controllable?**: The amount of `cTokens` to redeem.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: If the function reverts, the contract will be left in an inconsistent state.
- `cErcTokenBorrow.borrowBalanceStored(address(this))`
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?**:

N/A.

Function: `redeemUnderlying(address mintAddress, uint256 underlyingRedeemAmount)`

Allow the operator to redeem underlying tokens by burning minted tokens.

Inputs

- `mintAddress`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Must be a valid `cToken` address.
  - **Impact**: The address of the `cToken` to redeem.
- `underlyingRedeemAmount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Must be greater than 0.
  - **Impact**: The amount of underlying tokens to redeem.

Branches and code coverage (including function calls)

Intended branches

- Allow operator to `cTokens` into a specified quantity of underlying tokens.
  ☑ Test coverage
- Increase the underlying balance attributed to that `cToken` of the contract.
  □ Test coverage
- Decrease the `cToken` balance of the contract.
  □ Test coverage

Negative behavior

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

Function call analysis

- `cErcToken.redeemUnderlying(underlyingRedeemAmount)`:
  - **What is controllable?**: The amount of underlying tokens to redeem.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: If the function reverts, then the redeem failed.

Function: `repayBorrow(address borrowAddress)`

Should reduce the borrow balance of the contract by paying back some amount of the borrowed tokens.

Inputs

- `borrowAddress`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Must be a valid `cToken` address.
  - **Impact**: The address of the `cToken` to repay.

Branches and code coverage (including function calls)

Intended branches

- Should have approved the `cToken` to spend the underlying tokens.
  ☑ Test coverage
- Should have enough underlying tokens to repay the debt.
  ☑ Test coverage
- Should reduce the borrow balance of the contract.
  ☑ Test coverage
- Should reduce the underlying balance of the contract.
  ☑ Test coverage

Negative behaviour

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

Function call analysis

- `IERC20(underlyingAddress).approve(borrowAddress, underlyingBalance)`:
- **What is controllable?**: The amount of underlying tokens to approve.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: If the function reverts, then the approval failed.
- `cErcToken.borrowBalanceStored(address(this))`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value is not controllable.

  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.
- `cErcToken.repayBorrow(borrowBalance)`:
  - **What is controllable?**: The amount of underlying tokens to repay, to some degree.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value is not controllable.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: If the function reverts, then the repayment failed.

5.7 File: FractCrossProtocolStrategy.sol

Function: `deposit(IERC20 token, uint256 amount)`

Allows the owner to deposit tokens in the strategy.

Inputs

- `token`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will increase.
- `amount`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will increase by the amount deposited.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  ☑ Test coverage
- Assumes that the token will eventually be withdrawable from this contract.
  ☑ Test coverage

Negative behavior

- Should not assume that only the owner can transfer tokens to this contract. But transfers can happen manually too.
  ☑ Negative test

Function call analysis

- `token.transferFrom(msg.sender, address(this), amount)`:
  - **What is controllable?**: `msg.sender` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `repayTokensAave(address depositToken, uint256 repayAmount)`

Should repay tokens on AAVE.

Inputs

- `depositToken`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes this is the token that the strategy is supposed to be using.
  - **Impact**: Token balance of the strategy will decrease by the repaid amount.
- `repayAmount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes that the amount is <= than what can be repaid.
  - **Impact**: Token balance of the strategy will decrease by the repaid amount.

Branches and code coverage (including function calls)

Intended branches

- Should have approved the `depositToken` to the AAVE lending pool beforehand.
  ☑ Test coverage
- Will repay the `depositToken` to the AAVE lending pool.
  □ Test coverage
- Assumes there is something to repay.
  □ Test coverage
- Perform the checks like FractAaveV3Strategy does.
  □ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  ☑ Negative test

Function call analysis

- `lendingPool.repay(depositToken, repayAmount, 2, address(this))`:
  - **What is controllable?**: `repayAmount` and `depositToken`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the repayment fails.

Function: `withdraw(IERC20 token, uint256 amount)`

Allows the owner to withdraw tokens from the strategy.

Inputs

- `token`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will decrease by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  - ☐ Test coverage
- The balance of the strategy is assumed to be greater than the amount to withdraw.
  - ☐ Test coverage
- Decreases the balance of the strategy by the amount withdrawn.
  - ☐ Test coverage
- Increases the balance of the `msg.sender` by the amount withdrawn.
  - ☐ Test coverage

Negative behavior

- Should not be called by anyone other than the owner.
  - ☐ Negative test

Function call analysis

- `token.transfer(msg.sender, amount)`:
  - **What is controllable?**: `msg.sender` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

5.8 File: FractFraxConvexAlusdBp.sol

Function: `depositCurveConvex(uint256 amount, uint256[Literal(value=3.0, unit=None)] amounts, uint256 slippageBips, address proxyVault)`

Inputs

- `amount`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The amount of USDC to approve.
- `amounts`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Used in calculating the minimum amount of `lpToken` to receive.
- `slippageBips`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Used in calculating the minimum amount of `lpToken` to receive.
- `proxyVault`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: `proxyVault` is the address of the Convex vault.

Branches and code coverage (including function calls)

Intended branches

- Add liquidity in the ALUSD/FRAX pool.
  ☑ Test coverage
- Stake the received LP tokens in the Convex vault.
  ☑ Test coverage

- Assure that the slippage is within the acceptable range.
  - ☑ Test coverage

Negative behavior

- Should not be callable by anyone else than the operator.
  - ☑ Negative test

Function call analysis

- `ICurveSwap(FRAXZAPPOOL).calc_token_amount(ALUSDFRAXPOOL, amounts, true)`:
- **What is controllable?**: `amounts`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The calculation of the amount of `lpToken` to receive will fail.
- `ICurveSwap(FRAXZAPPOOL).add_liquidity(ALUSDFRAXPOOL, amounts, minAmount)`
- **What is controllable?**: `amounts`, `minAmount`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The addition of liquidity to the ALUSD/FRAX pool will fail.
- `IProxyVault(proxyVault).stakeLockedCurveLp(liquidity, 594000)`
  - **What is controllable?**: `liquidity`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The staking of the received `lpToken` in the Convex vault will fail.

Function: `getRewards(address proxyVault)`

Retrieve all pending rewards from the proxy vault.

Inputs

- `proxyVault`:
  - **Control**: Full control.
  - **Constraints**: Must be a valid address.
  - **Impact**: Retrieves all pending rewards.

Branches and code coverage (including function calls)

Intended branches

- Retrieve all outstanding rewards.

    - ☑ Test coverage
- Increase the rewards balance of the strategy.
    - ☑ Test coverage

Negative behavior

- Should not be callable by anyone else than the operator.
    - ☑ Negative test

Function call analysis

- `IProxyVault(proxyVault).getReward()`:
    - **What is controllable?**: The proxy vault address.
    - **If return value controllable, how is it used and how can it go wrong?**: The proxy vault address is used to retrieve all pending rewards. If the address is not a valid proxy vault, the call will revert.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: The call will revert if the address is not a valid proxy vault.

Function: `withdrawCurveConvex(byte[32] kek_id, address proxyVault, uint 256 slippageBips)`

Add liquidity to curve and deposit into Convex.

Inputs

- `kek_id`:
    - **Control**: Full control.
    - **Constraints**: N/A.
    - **Impact**: The `kek_id` is used to withdraw the `lpToken` from the `ProxyVault`.
- `proxyVault`:
    - **Control**: Full control.
    - **Constraints**: N/A.
    - **Impact**: Address of the `ProxyVault`.
- `slippageBips`:
    - **Control**: Full control.
    - **Constraints**: N/A.
    - **Impact**: Used in calculating the minimum amount of `lpToken` to receive.

Branches and code coverage (including function calls)

Intended branches

- Stake the LP tokens in the Convex vault.
  - □ Test coverage
- Receive `lpToken` from the `ProxyVault`.
  - □ Test coverage
- Stake the received `lpToken` in the Convex vault.
  - □ Test coverage
- Assure that the slippage is within the acceptable range.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone else than the operator.
  - ☑ Negative test

Function call analysis

- `IProxyVault(proxyVault).withdrawLockedAndUnwrap(kek_id)`:
- **What is controllable?**: `kek_id`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The withdrawal of `lpToken` from the `ProxyVault` will fail.
- `ICurveSwap(FRAXZAPPOOL).calc_withdraw_one_coin(ALUSDFRAXPOOL, lpTokenBalance, 2)`:
- **What is controllable?**: `lpTokenBalance`.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the estimated amount that will be received.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.
- `ICurveSwap(FRAXZAPPOOL).remove_liquidity_one_coin(ALUSDFRAXPOOL, lpTokenBalance, 2, minAmount)`:
  - **What is controllable?**: `lpTokenBalance` and `minAmount`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Then the staking of the `lpToken` in the Convex vault will fail.

5.9 File: FractFraxConvexTusdBp.sol

Function: `depositCurveConvex(uint256 amount, uint256[Literal(value=3.0, unit=None)] amounts, uint256 slippageBips, address proxyVault)`

Add liquidity to curve and deposit into Convex.

Inputs

- `amount`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The amount of USDC to approve.
- `amounts`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Used in calculating the minimum amount of `lpToken` to receive.
- `slippageBips`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Used in calculating the minimum amount of `lpToken` to receive.
- `proxyVault`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: `proxyVault` is the address of the Convex vault.

Branches and code coverage (including function calls)

Intended branches

- Add liquidity in the ALUSD/FRAX pool.
  ☑ Test coverage
- Stake the received LP tokens in the Convex vault.
  ☑ Test coverage
- Assure that the slippage is within the acceptable range.
  □ Test coverage

Negative behavior

- Should not be callable by anyone else than the operator.
  ☑ Negative test

Function call analysis

- `ICurveSwap(FRAXZAPPOOL).calc_token_amount(TUSDFRAXPOOL, amounts, true)`:

- **What is controllable?**: `amounts`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The calculation of the amount of `lpToken` to receive will fail.
- `ICurveSwap(FRAXZAPPOOL).add_liquidity(TUSDFRAXPOOL, amounts, minAmount)`:

- **What is controllable?**: `amounts` and `minAmount`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The addition of liquidity to the ALUSD/FRAX pool will fail.
- `IProxyVault(proxyVault).stakeLockedCurveLp(liquidity, 594000)`:
  - **What is controllable?**: `liquidity`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The staking of the received `lpToken` in the Convex vault will fail.

Function: `getRewards(address proxyVault)`

Harvest all pending rewards on Convex.

Inputs

- `proxyVault`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Address of the `ProxyVault`.

Branches and code coverage (including function calls)

Intended branches

- Increase the Convex rewards balance.
  ☐ Test coverage

Negative behavior

- Assure that it is not front-runnable; maybe call it often with low rewards.
  ☐ Negative test

Function call analysis

- `IProxyVault(proxyVault).getReward()`:
  - **What is controllable?**: N/A.

    - **If return value controllable, how is it used and how can it go wrong?**: N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: No rewards will be harvested.

Function: `withdrawCurveConvex(byte[32] kek_id, address proxyVault, uint 256 slippageBips)`

Add liquidity to curve and deposit into Convex.

Inputs

- `kek_id`:
    - **Control**: Full control.
    - **Constraints**: N/A.
    - **Impact**: The `kek_id` is used to withdraw the `lpToken` from the `ProxyVault`.
- `proxyVault`:
    - **Control**: Full control.
    - **Constraints**: N/A.
    - **Impact**: Address of the `ProxyVault`.
- `slippageBips`:
    - **Control**: Full control.
    - **Constraints**: N/A.
    - **Impact**: Used in calculating the minimum amount of `lpToken` to receive.

Branches and code coverage (including function calls)

Intended branches

- Stake the LP tokens in the Convex vault.
    ☐ Test coverage
- Receive `lpToken` from the `ProxyVault`.
    ☐ Test coverage
- Stake the received `lpToken` in the Convex vault.
    ☐ Test coverage
- Assure that the slippage is within the acceptable range.
    ☐ Test coverage

Negative behavior

- Should not be callable by anyone else than the operator.
    ☑ Negative test

Function call analysis

- `IProxyVault(proxyVault).withdrawLockedAndUnwrap(kek_id)`:

- **What is controllable?**: `kek_id`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The withdrawal of `lpToken` from the `ProxyVault` will fail.
- `ICurveSwap(FRAXZAPPOOL).calc_withdraw_one_coin(TUSDFRAXCRV, lpTokenBalance, 2)`:

- **What is controllable?**: `lpTokenBalance`.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the estimated amount that will be received.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.
- `ICurveSwap(FRAXZAPPOOL).remove_liquidity_one_coin(TUSDFRAXCRV, lpTokenBalance, 2, minAmount)`:
  - **What is controllable?**: `lpTokenBalance` and `minAmount`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Then the staking of the `lpToken` in the Convex vault will fail.

5.10 File: FractFraxConvex.sol

Function: `deposit(IERC20 token, uint256 amount)`

Allows the owner to deposit tokens in the strategy.

Inputs

- `token`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will increase.
- `amount`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will increase by the amount deposited.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  - ☐ Test coverage
- Assumes that the token will eventually be withdrawable from this contract.
  - ☐ Test coverage

Negative behavior

- Assumes that the token will eventually be withdrawable from this contract.
  - ☑ Negative test
- Should not assume that only the owner can transfer tokens to this contract. But transfers can happen manually too.
  - ☐ Negative test

Function call analysis

- `token.transferFrom(msg.sender, address(this), amount)`:
  - **What is controllable?**: `msg.sender` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `mintProxyVault(uint256 _pid)`

Should create a proxy vault for the given pool ID.

Inputs

- `_pid`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The pool ID to create a vault for.

Branches and code coverage (including function calls)

Intended branches

- Should store the proxy vault address somewhere.
  - ☐ Test coverage
- Create a vault for the given pool ID.

☑ Test coverage

Negative behavior

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

Function call analysis

- `IFraxBooster(FRAXBOOSTER).createVault(_pid)`:
  - **What is controllable?**: `_pid`.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the proxy vault address.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the call fails.

Function: `swap(IERC20 token, uint256 amount, byte[] callData)`

Should swap the rewards tokens for the underlying token.

Inputs

- `token`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes the Paraswap router will accept the token.
  - **Impact**: Token balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Token balance of the strategy will decrease.
- `callData`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Dictates how the routing will be done in Paraswap. Assumes it is validated off chain.

Branches and code coverage (including function calls)

Intended branches

- Assumes the token is the one that the strategy is supposed to be using.
  □ Test coverage
- Assumes the amount is less than the balance of the strategy.

    - ☐ Test coverage
- Assumes the `callData` is valid.
    - ☐ Test coverage
- Decreases the balance of the strategy by the amount.
    - ☐ Test coverage
- Increases the balance of the underlying token by the amount. This should be checked to ensure the validity of the `callData` itself.
    - ☐ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
    - ☑ Negative test
- Should not be called if the token is not the one that the strategy is supposed to be using.
    - ☑ Negative test

Function call analysis

- `getTokenTransferProxy()`:

- **What is controllable?**: N/A.
    - **If return value controllable, how is it used and how can it go wrong?**: Returns the token transfer proxy for the Paraswap router.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A; it should revert altogether.
- `token.approve(tokenTransferProxy, amount)`:

- **What is controllable?**: `amount`.
    - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the approval was successful or not.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.
- `PARASWAP.call(callData)`:

- **What is controllable?**: `callData`.
    - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the swap was successful or not.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the swap fails.
- `token.approve(tokenTransferProxy, 0)`:
    - **What is controllable?**: N/A.
    - **If return value controllable, how is it used and how can it go wrong?**: Tells

  whether the approval was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.

Function: `withdrawToOwner(IERC20 token, uint256 amount)`

Allows the operator to withdraw tokens from the strategy for the owner.

Inputs

- `token`:
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will decrease by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  □ Test coverage
- The balance of the strategy is assumed to be greater than the amount to withdraw.
  □ Test coverage
- Decreases the balance of the strategy by the amount withdrawn.
  □ Test coverage
- Increases the balance of the owner by the amount withdrawn.
  □ Test coverage

Negative behavior

- Should not be called by anyone other than the controller.
  □ Negative test

Function call analysis

- `token.transfer(owner, amount)`:
  - **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells

whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `withdraw(IERC20 token, uint256 amount)`

Allows the owner to withdraw tokens from the strategy.

Inputs

- `token`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will decrease by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  - □ Test coverage
- The balance of the strategy is assumed to be greater than the amount to withdraw.
  - □ Test coverage
- Decreases the balance of the strategy by the amount withdrawn.
  - □ Test coverage
- Increases the balance of the `msg.sender` by the amount withdrawn.
  - □ Test coverage

Negative behavior

- Should not be called by anyone other than the owner.
  - □ Negative test

Function call analysis

- `token.transfer(msg.sender, amount)`:
  - **What is controllable?**: `msg.sender` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells

whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

5.11 File: FractGmxAvax.sol

Function: `depositEsGmx()`

Deposit esGMX into the GLP Vester.

Branches and code coverage (including function calls)

Intended branches

- Decrease the amount of esGMX in the contract.
  - □ Test coverage
- Stake the esGMX in the GLP Vester.
  - □ Test coverage
- Assure that the GLP Vester has the correct amount of esGMX.
  - □ Test coverage

Negative behavior

- Revert if the approval fails.
  - □ Negative test
- Should not allow anyone else other than the operator to call this function.
  - ☑ Negative test

Function call analysis

- `IERC20(ESGMX).approve(GLP_VESTER, esGmxBalance)`:

- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the approval was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.
- `IVester(GLP_VESTER).deposit(esGmxBalance)`
  - **What is controllable?**: `esGmxBalance` is controllable to some extent.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.

Function: `depositGlp(uint256 amount, uint256 minGlpAmount)`

Allows the operator to deposit USDC into GMX to receive GLP.

Inputs

- `amount`
  - **Control**: Fully controlled by the operator.
  - **Constraints**: > 0.
  - **Impact**: Balance of the strategy will decrease by the amount deposited.
- `minGlpAmount`
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Balance of GLP will increase by the amount deposited.

Branches and code coverage (including function calls)

Intended branches

- Decreases the balance of USDC by the amount.
  - ☐ Test coverage
- Receive at least the minimum amount of GLP.
  - ☐ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  - ☑ Negative test

Function call analysis

- `IRewardRouter(REWARD_ROUTER_V2).mintAndStakeGlp(USDC, amount, 0, minGlpAmount)`:
  - **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the staking was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `deposit(IERC20 token, uint256 amount)`

Allows the owner to deposit tokens in the strategy.

Inputs

- `token`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will increase.
- `amount`
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will increase by the amount deposited.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  ☑ Test coverage
- Assumes that the token will eventually be withdrawable from this contract.
  ☑ Test coverage

Negative behavior

- Should not assume that only the owner can transfer tokens to this contrct. But transfers can happen manually too.
  ☑ Negative test

Function call analysis

- `token.transferFrom(msg.sender, address(this), amount)`:
  - **What is controllable?**: `msg.sender` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `getRewards()`

Retrieve rewards from the reward router.

Branches and code coverage (including function calls)

Intended branches

- Increase the balance of rewards.

  □ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  ☑ Negative test

Function call analysis

- `IRewardRouter(REWARD_ROUTER).handleRewards(true, false, true, false, false, true, false)`:
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Should not receive any rewards.

Function: `swap(IERC20 token, uint256 amount, byte[] callData)`

Should swap the rewards tokens for the underlying token.

Inputs

- `token`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes the Paraswap router will accept the token.
  - **Impact**: Token balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Token balance of the strategy will decrease.
- `callData`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Dictates how the routing will be done in Paraswap. Assumes it is validated off chain.

Branches and code coverage (including function calls)

Intended branches

- Assumes the token is the one that the strategy is supposed to be using.
  □ Test coverage
- Assumes the amount is less than the balance of the strategy.

  - □ Test coverage
- Assumes the `callData` is valid.
  - □ Test coverage
- Decreases the balance of the strategy by the amount.
  - □ Test coverage
- Increases the balance of the underlying token by the amount. This should be checked to ensure the validity of the `callData` itself.
  - □ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  - □ Negative test
- Should not be called if the token is not the one that the strategy is supposed to be using.
  - □ Negative test

Function call analysis

- `getTokenTransferProxy()`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the token transfer proxy for the Paraswap router.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A; it should revert altogether.
- `token.approve(tokenTransferProxy, amount)`:
- **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the approval was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.
- `PARASWAP.call(callData)`:
- **What is controllable?**: `callData`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the swap was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the swap fails.
- `token.approve(tokenTransferProxy, 0)`:
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells

whether the approval was successful or not.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.

Function: `withdrawEsGmx()`

Allows operator to withdraw from GMX vesting contract.

Branches and code coverage (including function calls)

Intended branches

- Increase the esGMX balance of the contract.
  - ☐ Test coverage

Negative behaviour

- Should not allow anyone other than the owner to withdraw.
  - ☑ Negative test

Function call analysis

- `IVester(GLP_VESTER).withdraw()`
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Withdrawal will fail.

Function: `withdrawGlp(uint256 amount, uint256 minUsdcAmount)`

Allow the operator to withdraw USDC from GMX.

Inputs

- `amount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: > 0.
  - **Impact**: The amount of GLP that will be burned.
- `minUsdcAmount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: > 0.
  - **Impact**: The minimum amount of USDC that will be received.

Branches and code coverage (including function calls)

Intended branches

- Increase the USDC balance of the strategy by the amount that was withdrawn.
  - ☐ Test coverage

Negative behaviour

- Nobody other than the operator can call this function.
  - ☐ Negative test

Function call analysis

- `IRewardRouter(REWARD_ROUTER_V2).unstakeAndRedeemGlp(USDC, amount, minUsdcAmount, address(this))`:
  - **What is controllable?**: `amount` and `minUsdcAmount`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the unstake and redeem fails.

Function: `withdrawToOwner(IERC20 token, uint256 amount)`

Allows the operator to withdraw tokens from the strategy for the owner.

Inputs

- `token`:
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will decrease by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  - ☐ Test coverage
- The balance of the strategy is assumed to be greater than the amount to withdraw.

  - □ Test coverage
- Decreases the balance of the strategy by the amount withdrawn.
  - □ Test coverage
- Increases the balance of the owner by the amount withdrawn.
  - ☑ Test coverage

Negative behavior

- Should not be called by anyone other than the controller.
  - ☑ Negative test

Function call analysis

- `token.transfer(owner, amount)`:
  - **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `withdraw(IERC20 token, uint256 amount)`

Allows the owner to withdraw tokens from the strategy.

Inputs

- `token`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will decrease by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  - □ Test coverage
- The balance of the strategy is assumed to be greater than the amount to withdraw.

  - □ Test coverage
- Decreases the balance of the strategy by the amount withdrawn.
  - □ Test coverage
- Increases the balance of the `msg.sender` by the amount withdrawn.
  - □ Test coverage

Negative behavior

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

Function call analysis

- `token.transfer(msg.sender, amount)`:
  - **What is controllable?**: `msg.sender` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

5.12 File: FractMoonwellStrategy.sol

Function: `compoundRewards(address mintAddress, address borrowAddress, address depositToken)`

Should claim and reinvest rewards.

Inputs

- `mintAddress`
  - **Control**: Full control.
  - **Constraints**: Only checked that it is not 0 address.
  - **Impact**: The `cToken` market to claim on.
- `borrowAddress`
  - **Control**: Full control.
  - **Constraints**: Only checked that it is not 0 address.
  - **Impact**: The `cToken` market to claim on and swap into.
- `depositToken`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The token that will be borrowed.

Branches and code coverage (including function calls)

Intended branches

- Ensure that claiming works as expected.
  - □ Test coverage
- Ensure that swapping works as expected.
  - □ Test coverage
- Ensure that borrowing works as expected. This means that the balance of borrowed token should increase accordingly.
  - □ Test coverage
- Ensure that there are no slippage issues in any of the above steps.
  - □ Test coverage

Negative behavior

- Ensure that no one other than the operator can call this function.
  - □ Negative test
- Ensure that all claimed rewards are used and that no dust is left behind nor any tokens are left unclaimed.
  - □ Negative test
- Should disallow swapping into the same token as the one being swapped from.
  - □ Negative test
- Should disallow swapping into a token that is not supported by the borrow market.
  - □ Negative test

Function call analysis

- `IMoonwellComptroller(MOONWELL_COMPTROLLER).claimReward(0, address(this), claimAddresses);`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: n/a.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Rewards will not be claimed.
- `IMoonwellComptroller(MOONWELL_COMPTROLLER).claimReward(1, address(this), claimAddresses);`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The reward will not be claimed.

- `_swapTokens(MOONWELL_TOKEN, underlyingAddress, rewardBalance, 0);`:

- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Swap will not be successful.
- `_swapNativeTokens(underlyingAddress, movrBalance, 0);`:

- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The swap will not be successful.
- `mintAndBorrow(mintAddress, borrowAddress, depositToken);`:
  - **What is controllable?**: `mintAddress` and `borrowAddress`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Then the borrow will not be successful.

5.13 File: FractStableswap.sol

Function: `addLiquidityAndDeposit(IERC20 token, uint256 amount, uint256 amounts, uint256 slippageBips, uint256 pid)`

Add liquidity to the Synapse stableswap pool and stake in the masterchef.

Inputs

- `token`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes it is the token used by stableswap.
  - **Impact**: Token that will be added to the stableswap pool.
- `amount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Amount to deposit.
- `amounts`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Amounts for the pool.
- `slippageBips`:

  - **Control**: Fully controlled by the operator.
  - **Constraints**: Slippage should be within the acceptable range, calculated as per the `minAmount` calculation.
  - **Impact**: The maximum slippage that will be accepted.
- `pid`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes masterchef has the pool.
  - **Impact**: The pool that the LP tokens will be staked in.

Branches and code coverage (including function calls)

Intended branches

- Add liquidity with the particular token to the Synapse stableswap pool.
  ☑ Test coverage
- Stake the received LP tokens in the masterchef.
  ☑ Test coverage
- Assure that the slippage is within the acceptable range, as per the `minAmount` calculation.
  □ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  ☑ Negative test

Function call analysis

- `token.approve(stableSwap, amount)`:

- **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the approval was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.
- `ISwap(stableSwap).addLiquidty(amounts, minAmount, block.timestamp + 10)`:

- **What is controllable?**: `amounts` and `minAmount`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the liquidity addition fails. Most likely due to slippage issues or the amounts being too low.
- `IERC20(lpToken).approve(miniChef, lpTokenBalance)`:

- **What is controllable?**: `lpTokenBalance`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the approval was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.
- `IMiniChefV2(miniChef).deposit(pid, lpTokenBalance, address(this))`:
  - **What is controllable?**: `lpTokenBalance`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Should revert if deposit fails.

Function: `deposit(IERC20 token, uint256 amount)`

Allows the owner to deposit tokens in the strategy.

Inputs

- `token`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will increase.
- `amount`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will increase by the amount deposited.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  - □ Test coverage
- Assumes that the token will eventually be withdrawable from this contract.
  - □ Test coverage

Negative behavior

- Should not assume that only the owner can transfer tokens to this contract. But transfers can happen manually too.
  - □ Negative test

Function call analysis

- `token.transferFrom(msg.sender, address(this), amount)`:
  - **What is controllable?**: `msg.sender` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `getRewards(uint256 pid)`

Retrieves the rewards from the masterchef.

Inputs

- `pid`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A; assumed to be a valid pool ID.
  - **Impact**: The pool ID that the rewards will be harvested from.

Branches and code coverage (including function calls)

Intended branches

- Harvest the rewards from the masterchef of the `pid` that was staked in.
  ☑ Test coverage
- Increase the balance of the rewards token of the strategy.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone else than the operator.
  ☑ Negative test

Function call analysis

- `IMiniChefV2(miniChef).harvest(pid, address(this))`:
  - **What is controllable?**: `pid`.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Should revert if the harvest fails.

Function: `swap(IERC20 token, uint256 amount, byte[] callData)`

Should swap the rewards tokens for the underlying token.

Inputs

- `token`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes the Paraswap router will accept the token.
  - **Impact**: Token balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Token balance of the strategy will decrease.
- `callData`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Dictates how the routing will be done in Paraswap. Assumes it is validated off chain.

Branches and code coverage (including function calls)

Intended branches

- Assumes the token is the one that the strategy is supposed to be using.
  ☐ Test coverage
- Assumes the amount is less than the balance of the strategy.
  ☐ Test coverage
- Assumes the `callData` is valid.
  ☐ Test coverage
- Decreases the balance of the strategy by the amount.
  ☐ Test coverage
- Increases the balance of the underlying token by the amount. This should be checked to ensure the validity of the `callData` itself.
  ☐ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  ☐ Negative test
- Should not be called if the token is not the one that the strategy is supposed to be using.

- □ Negative test

Function call analysis

- `getTokenTransferProxy()`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the token transfer proxy for the Paraswap router.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A; it should revert altogether.
- `token.approve(tokenTransferProxy, amount)`:
- **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the approval was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.
- `PARASWAP.call(callData)`:
- **What is controllable?**: `callData`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the swap was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the swap fails.
- `token.approve(tokenTransferProxy, 0)`:
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the approval was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.

Function: `withdrawToOwner(IERC20 token, uint256 amount)`

Allows the operator to withdraw tokens from the strategy for the owner.

Inputs

- `token`:
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will decrease.

- `amount`:
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will decrease by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  ☐ Test coverage
- The balance of the strategy is assumed to be greater than the amount to withdraw.
  ☐ Test coverage
- Decreases the balance of the strategy by the amount withdrawn.
  ☐ Test coverage
- Increases the balance of the owner by the amount withdrawn.
  ☐ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  ☑ Negative test

Function call analysis

- `token.transfer(owner, amount)`:
  - **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `withdraw(IERC20 token, uint256 amount)`

Allows the owner to withdraw tokens from the strategy.

Inputs

- `token`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will decrease.

- `amount`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will decrease by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  □ Test coverage
- The balance of the strategy is assumed to be greater than the amount to withdraw.
  □ Test coverage
- Decreases the balance of the strategy by the amount withdrawn.
  □ Test coverage
- Increases the balance of the `msg.sender` by the amount withdrawn.
  □ Test coverage

Negative behavior

- Should not be called by anyone other than the owner.
  □ Negative test

Function call analysis

- `token.transfer(msg.sender, amount)`:
  - **What is controllable?**: `msg.sender` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

5.14 File: FractTusdFraxBp.sol

Function: `depositCurveConvex(uint256 amount, uint256[Literal(value=3.0, unit=None)] amounts, uint256 slippageBips)`

Add liquidity to curve and deposit into Convex.

Inputs

- `amount`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The amount of USDC to approve.
- `amounts`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Used in calculating the minimum amount of `lpToken` to receive.
- `slippageBips`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Used in calculating the minimum amount of `lpToken` to receive.
- `proxyVault`:
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: `proxyVault` is the address of the Convex vault.

Branches and code coverage (including function calls)

Intended branches

- Add liquidity in the ALUSD/FRAX pool.
  ☐ Test coverage
- Stake the received LP tokens in the Convex vault.
  ☐ Test coverage
- Assure that the slippage is within the acceptable range.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone else than the operator.
  ☐ Negative test

Function call analysis

- `ICurveSwap(FRAXZAPPOOL).calc_token_amount(TUSDFRAXPOOL, amounts, true)`:

- **What is controllable?**: `amounts`.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the amount of LP tokens to receive.
  - **What happens if it reverts, reenters, or does other unusual control flow?**:

N/A.

- IERC20(USDC).approve(FRAXZAPPOOL, amount):
- **What is controllable?**: amount.
  - **If return value controllable, how is it used and how can it go wrong?**: Approves the amount of USDC to be used by the Curve pool.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.
- ICurveSwap(FRAXZAPPOOL).add_liquidity(TUSDFRAXPOOL, amounts, minAmount):
- **What is controllable?**: amounts and minAmount.
  - **If return value controllable, how is it used and how can it go wrong?**: Adds liquidity to the Curve pool.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the liquidity addition fails.
- IERC20(TUSDFRAXCRV).approve(CONVEXBOOSTER, calcAmount):
- **What is controllable?**: calcAmount.
  - **If return value controllable, how is it used and how can it go wrong?**: Approves the amount of LP tokens to be used by the Convex vault.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.
- IBooster(CONVEXBOOSTER).depositAll(108, true):
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Deposits the LP tokens in the Convex vault.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the deposit fails.

Function: deposit(IERC20 token, uint256 amount)

Allows the owner to deposit tokens in the strategy.

Inputs

- token:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: token balance of the strategy will increase.
- amount:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.

- **Impact**: Balance of the strategy will increase by the amount deposited.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  - □ Test coverage
- Assumes that the token will eventually be withdrawable from this contract.
  - □ Test coverage

Negative behaviour

- Assumes that the token will eventually be withdrawable from this contract.
  - □ Negative test
- Should not assume that only the owner can transfer tokens to this contract. But transfers can happen manually too.
  - □ Negative test

Function call analysis

- `token.transferFrom(msg.sender, address(this), amount)`:
  - **What is controllable?**: `msg.sender` and `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `getRewards()`

Harvests all pending rewards on Convex.

Branches and code coverage (including function calls)

Intended branches

- Increase the amount of rewards of the strategy.
  - □ Test coverage

Negative behaviour

- Should not be callable by anyone other than the operator.
  - □ Negative test

Function call analysis

- `IBaseRewardPool(REWARDPOOL).getReward(address(this), true)`:
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: No rewards are retrieved.

Function: `swap(IERC20 token, uint256 amount, byte[] callData)`

Should swap the rewards tokens for the underlying token.

Inputs

- `token`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: Assumes the Paraswap router will accept the token.
  - **Impact**: Token balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Token balance of the strategy will decrease.
- `callData`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Dictates how the routing will be done in Paraswap. Assumes it is validated off chain.

Branches and code coverage (including function calls)

Intended branches

- Assumes the token is the one that the strategy is supposed to be using.
  - □ Test coverage
- Assumes the amount is less than the balance of the strategy.
  - □ Test coverage
- Assumes the `callData` is valid.
  - □ Test coverage
- Decreases the balance of the strategy by the amount.
  - □ Test coverage
- Increases the balance of the underlying token by the amount. This should be checked to ensure the validity of the `callData` itself.

- □ Test coverage

Negative behavior

- Should not be called by anyone other than the operator.
  - □ Negative test
- Should not be called if the token is not the one that the strategy is supposed to be using.
  - □ Negative test

Function call analysis

- `getTokenTransferProxy()`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the token transfer proxy for the Paraswap router.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A; it should revert altogether.
- `token.approve(tokenTransferProxy, amount)`:
- **What is controllable?**: `amount`
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the approval was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.
- `PARASWAP.call(callData)`:
- **What is controllable?**: `callData`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the swap was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the swap fails.
- `token.approve(tokenTransferProxy, 0)`:
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the approval was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the approval fails.

Function: `withdrawCurveConvex(uint256 slippageBips)`

Withdraw all from Convex and Curve.

Inputs

- `slippageBips`:
  - **Control**: Fully controlled by the operator.
  - **Constraints**: N/A.
  - **Impact**: Used to calculate the minimum amount of curve LP tokens to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Assumes entire balances are withdrawn.
  - ☐ Test coverage
- The USDC balance should increase after the withdrawal.
  - ☐ Test coverage
- Assumes the slippage is within the acceptable range.
  - ☐ Test coverage

Negative behavior

- Should not be callable by anyone else than the operator.
  - ☐ Negative test

Function call analysis

- `IBaseRewardPool(REWARDPOOL).withdrawAllAndUnwrap(true)`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A; it should revert altogether.
- `IERC20(TUSDFRAXCRV).balanceOf(address(this))`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A; it should revert altogether.
- `IERC20(TUSDFRAXCRV).approve(FRAXZAPPOOL, lpTokenBalance)`:
- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A; it should revert altogether.

- `ICurveSwap(FRAXZAPPOOL).calc_withdraw_one_coin(TUSDFRAXPOOL, lpTokenBalance, 2)`:

- **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A; it should revert altogether.
- `ICurveSwap(FRAXZAPPOOL).remove_liquidity_one_coin(TUSDFRAXPOOL, lpTokenBalance, 2, minAmount)`:

- **What is controllable?**: n/a
  - **If return value controllable, how is it used and how can it go wrong?**: n/a
  - **What happens if it reverts, reenters, or does other unusual control flow?**: n/a; it should revert altogether.
- `IERC20(TUSDFRAXCRV).approve(FRAXZAPPOOL, 0)`:
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A; it should revert altogether.

Function: `withdrawToOwner(IERC20 token, uint256 amount)`

Allows the operator to withdraw tokens from the strategy for the owner.

Inputs

- `token`:
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the controller.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will decrease by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  - □ Test coverage
- The balance of the strategy is assumed to be greater than the amount to with-

draw.
  - □ Test coverage
- Decreases the balance of the strategy by the amount withdrawn.
  - □ Test coverage
- Increases the balance of the owner by the amount withdrawn.
  - □ Test coverage

Negative behavior

- Should not be called by anyone other than the controller.
  - □ Negative test

Function call analysis

- `token.transfer(owner, amount)`:
  - **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

Function: `withdraw(IERC20 token, uint256 amount)`

Allows the owner to withdraw tokens from the strategy.

Inputs

- `token`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: `token` balance of the strategy will decrease.
- `amount`:
  - **Control**: Fully controlled by the owner.
  - **Constraints**: N/A.
  - **Impact**: Balance of the strategy will decrease by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Assumes it is the token that the strategy is supposed to be using.
  - □ Test coverage
- The balance of the strategy is assumed to be greater than the amount to with-

draw.
    - □ Test coverage
- Decreases the balance of the strategy by the amount withdrawn.
    - □ Test coverage
- Increases the balance of the `msg.sender` by the amount withdrawn.
    - □ Test coverage

Negative behavior

- Should not be called by anyone other than the owner.
    - □ Negative test

Function call analysis

- `token.transfer(msg.sender, amount)`:
    - **What is controllable?**: `msg.sender` and `amount`.
    - **If return value controllable, how is it used and how can it go wrong?**: Tells whether the transfer was successful or not.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: Reverts if the transfer fails.

5.15 File: GlpSpotMarginAccount.sol

Function: `depositEsGmx()`

Deposits esGMX for vesting.

Branches and code coverage (including function calls)

Intended branches

- Check that esGmx was deposited.
    - □ Test coverage

Negative behavior

Function call analysis

- `depositEsGmx` → `ESGMX.balanceOf(this)`:
    - **What is controllable?**: Nothing.
    - **If return value controllable, how is it used and how can it go wrong?**: Amount of esGMX held by this.
    - **What happens if it reverts, reenters, or does other unusual control flow?**:

Ok.

- `depositEsGmx` → `ESGMX.approve(GLP_VESTER, esGmxBalance)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `depositEsGmx` → `GLP_VESTER.deposit()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `depositEsGmx` → `ESGMX.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Amount of esGMX held by this after having depositted esGMX.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `depositEsGmx` → `ESGMX.approve(GLP_VESTER, 0)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `depositGlp(uint256 amount, uint256 minGlpAmount)`

Deposits USDC to receive GLP and stake the GLP to get fsGLP.

Inputs

- `amount`:
  - **Control**: Full.
  - **Constraints**: > 0.
  - **Impact**: Amount of GLP to deposit.
- `minGlpAmount`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Slippage check for the amount of fsGLP received.

Branches and code coverage (including function calls)

Intended branches

- Should check that minGlpAmount is honored.
  - ☑ Test coverage

Negative behavior

- Should check that more than amount tokens is not taken.
  - ☑ Negative test

Function call analysis

- `depositGlp` → `USDC.approve(REWARD_ROUTER_V2, amount)`:
  - **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `depositGlp` → `USDC.approve(GLP_MANAGER_V2, amount)`:
  - **What is controllable?**: `amount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `depositGlp` → `FS_GLP.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Amount of fsGlpBalanceBefore.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `depositGlp` → `REWARD_ROUTER_V2.mintAndStakeGlp(USDC, amount, 0, minGlpAmount)`:
  - **What is controllable?**: `amount`, `minGlpAmount`.
  - **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?**: Ok.
- `depositGlp` → `FS_GLP.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**:

  Amount of `fsGlpBalanceBefore`.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `depositGlp` → `USDC.approve(REWARD_ROUTER_V2, 0)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `depositGlp` → `USDC.approve(GLP_MANAGER_V2, 0)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `deposit(address token, uint256 amount)`

Deposits tokens into the GlpSpotMarginAccount.

Inputs

- `token`:
  - **Control**: Full.
  - **Constraints**: != 0.
  - **Impact**: The token.
- `amount`:
  - **Control**: Full.
  - **Constraints**: > 0.
  - **Impact**: The amount to deposit.

Branches and code coverage (including function calls)

Intended branches

- Deposits the specified amount of tokens:
  - ☐ Test coverage

Negative behavior

- Deposits more than the specified amount of tokens:
  - ☐ Negative test

Function call analysis

- `deposit` → `_deposit`

- **What is controllable?**: `token`.
  - **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.
- `deposit` → `_deposit` → `token.transferFrom(msg.sender, this, amount)`
  - **What is controllable?**: `amount` and `token`.
  - **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.

Function: `getRewards()`

Gets the rewards.

Branches and code coverage (including function calls)

Intended branches

- Should check rewards were accumulated.
  ☑ Test coverage

Negative behavior

- Should not be callable by anyone other than the operator.
  □ Negative test

Function call analysis

- `getRewards` → `REWARD_ROUTER.handleRewards`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `glpUnwind(uint256 unwindBips)`

Unwind a GLP position.

Inputs

- `unwindBips`:
  - **Control**: Full.
  - **Constraints**: Should be less than 10,000 (no checks?).
  - **Impact**: The amount to unwind.

Branches and code coverage (including function calls)

Intended branches

- Unwinds (closes) the GLP position according to the number of bips.
  - □ Test coverage is commented out
- Unwinds (closes) the GLP position according to the number of bips and pays interest first when interest is due.
  - □ Test coverage is commented out

Negative behavior

- Reverts when the user does not have enough tokens.
  - □ Test coverage is commented out
- User does not pay for more than the loan amount to be unwinded.
  - □ Test coverage is commented out

Function call analysis

- `glpUnwind → Loan.getDebt()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Determines amount for payback/interest/lateFees.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: User cannot close their debts.
- `glpUnwind → getCurrentGlpPrice()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Price of GLP could be problematic if controllable, causing user to take a bad deal; however, `minUsdc` should prevent any sort of manipulaton.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: User cannot close their debts.
- `glpUnwind → withdrawEsGmx()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.

    - **What happens if it reverts, reenters, or does other unusual control flow?**: User cannot close their debts.
- `glpUnwind` → `FS_GLP.balanceOf()`:
    - **What is controllable?**: Nothing.
    - **If return value controllable, how is it used and how can it go wrong?**: The amount of fsGLP withdrawn from `withdrawEsGmx`.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: User cannot close their debts.
- `glpUnwind` → `withdrawGlp()`:
    - **What is controllable?**: Nothing.
    - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: User cannot close their debts.
- `glpUnwind` → `USDC.balanceOf()`:
    - **What is controllable?**: Nothing.
    - **If return value controllable, how is it used and how can it go wrong?**: Amount of USDC that the spot margin account has.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: User cannot close their debts.
- `glpUnwind` → `repayLoanInterest()`:
    - **What is controllable?**: Nothing.
    - **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 cannot close their debts.
- `glpUnwind` → `repayLoanPrincipal()`:
    - **What is controllable?**: Nothing.
    - **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 cannot close their debts.

Function: `withdrawEsGmx()`

Withdraws the accumulated rewards.

Branches and code coverage (including function calls)

Intended branches

- Check it withdraws when rewards are more than 0.
  - □ Test coverage

Negative behavior

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

Function call analysis

- `withdrawEsGmx` → `FS_GLP.cumulativeRewards(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Amount of rewards used to determine whether to call withdraw or not.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `withdrawEsGmx` → `GLP_VESTER.withdraw()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Nothing.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `withdrawGlp(uint256 amount, uint256 minUsdcAmount)`

Withdraws GLP and receives USDC.

Inputs

- `amount`:
  - **Control**: Full.
  - **Constraints**: > 0.
  - **Impact**: Amount to withdraw.
- `minUsdcAmount`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Slippage check for the amount of USDC received.

Branches and code coverage (including function calls)

Intended branches

- check that `fsGlpAmount` was withdrawn.

    - ☐ Test coverage
- Check that the `minUsdcAmount` slippage check was honored.
    - ☐ Negative test

Negative behavior

Function call analysis

- `withdrawGlp → USDC.balanceOf(this)`:
    - **What is controllable?**: Nothing.
    - **If return value controllable, how is it used and how can it go wrong?**: Amount of USDC prewithdrawal.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `withdrawGlp → REWARD_ROUTER_V2.unstakeAndRedeemGlp(USDC, amount, minUsdcAmount, address(this))`:
    - **What is controllable?**: `amount` and `minUsdcAmount`.
    - **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?**: Ok.
- `withdrawGlp → USDC.balanceOf(this)`:
    - **What is controllable?**: Nothing.
    - **If return value controllable, how is it used and how can it go wrong?**: Amount of USDC prewithdrawal.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `withdrawToOwner(address token, uint256 amount)`

Withdraws to the owner.

Inputs

- `token`:
    - **Control**: Full.
    - **Constraints**: != 0.
    - **Impact**: The token to withdraw from the account.
- `amount`:
    - **Control**: Full.
    - **Constraints**: > 0.

    - **Impact**: The amount of token to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Check that capital has returned to its owner.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the operator.
  - □ Negative test

Function call analysis

- `withdrawToOwner → _withdrawToOwner()`:
  - **What is controllable?**: `token` and `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?**: Ok.
- `withdrawToOwner → _withdrawToOwner() → token.transfer(this, owner, amount)`:
  - **What is controllable?**: `token` and `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?**: Ok.

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

Withdraws from account.

Inputs

- `token`:
  - **Control**: Full.
  - **Constraints**: != 0.
  - **Impact**: The token to withdraw from the account.
- `amount`:
  - **Control**: Full.
  - **Constraints**: > 0.

    - **Impact**: The amount of token to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Withdraws the correct amount.
  - ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the operator.
  - ☐ Negative test

Function call analysis

- `withdraw → _withdraw()`:
  - **What is controllable?**: `token` and `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?**: Ok.
- `withdraw → _withdraw() → token.transfer(this, msg.sender, amount)`:
  - **What is controllable?**: `token` and `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?**: Ok.

5.16 File: LoansManager.sol

Function: `deployOpenTermLoan(address lenderAddr, address borrowerAddr, IERC20NonCompliant newPrincipalToken, IBasicPriceOracle newOracle, address newCollateralToken, uint256 fundingPeriodInDays, uint256 newPaymentIntervalInSeconds, uint256 loanAmountInPrincipalTokens, uint256 originationFeePercent2Decimals, uint256 newAprWithTwoDecimals, uint256 initialCollateralRatioWith2Decimals)`

Deploys a loan.

Inputs

- `lenderAddr`:

  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Lender.
- `borrowerAddr`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Borrower.
- `newPrincipalToken`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Prinicipal token.
- `newOracle`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Oracle.
- `newCollateralToken`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Collateral token.
- `fundingPeriodInDays`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Funding period.
- `newPaymentIntervalInSeconds`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Payment interval.
- `loanAmountInPrincipalTokens`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Loan amount.
- `originationFeePercent2Decimals`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Fee percentage.
- `newAprWithTwoDecimals`:
  - **Control**: Full.
  - **Constraints**: None.

  - **Impact**: APR.
- `initialCollateralRatioWith2Decimals`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Collateral ratio.

Branches and code coverage (including function calls)

Intended branches

- Creates the loan and sets in the loans mapping.
  - □ Test coverage

Negative behavior

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

Function call analysis

- `deployOpenTermLoan` → `new OpenTermLoan`:
  - **What is controllable?**: Everything.
  - **If return value controllable, how is it used and how can it go wrong?**: The contract's address.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Nothing.

5.17 File: Mintable.sol

Function: `burn(address addr, uint256 amount)`

Should allow authorized burners to burn tokens from the specified address.

Inputs

- `addr`:
  - **Control**: `onlyBurner` can call this, only if the contract is initialized.
  - **Constraints**: `addr` must not be the zero address or the contract address; also, the account balance must be greater than or equal to the amount.
  - **Impact**: Destination address will lose the specified amount of tokens.
- `amount`:
  - **Control**: `onlyBurner` can call this, only if the contract is initialized.

  - **Constraints**: Amount must be greater than zero, and the `_totalSupply + amount` must not exceed the max token supply.
  - **Impact**: `addr` will lose amount tokens.

Branches and code coverage (including function calls)

Intended branches

- Should revert if the address is the zero address or the contract address.
  ☑ Test coverage
- Should revert if the amount is zero.
  ☑ Test coverage
- Should revert if the total supply + amount exceeds the max token supply.
  ☑ Test coverage
- Should deplete the balance of the destination address, `_balances[addr] -= amount;`.
  ☑ Test coverage
- `_totalSupply` should be updated, `_totalSupply -= amount;`.
  ☑ Test coverage

Negative behavior

- Should not be able to call this function if the contract is not initialized.
  ☑ Negative test
- Should not be able to call this function if the caller is not a burner.
  ☑ Negative test

Function: `grantBurner(address addr)`

Grants burner role to the specified address.

Inputs

- `addr`:
  - **Control**: `onlyOwner` can call this.
  - **Constraints**: `addr` must not be authorized already.
  - **Impact**: `addr` is authorized to burn tokens.

Branches and code coverage (including function calls)

Intended branches

- Should revert if the address is already authorized.

    - □ Test coverage
- Should emit an event if the address is authorized.
    - □ Test coverage
- Should update the authorized burners mapping so that the address is authorized.
    - □ Test coverage

Negative behavior

- Should not be able to call this function if the contract is not initialized.
    - □ Negative test
- Should not be able to call this function if the caller is not the owner.
    - □ Negative test

Function: `grantMinter(address addr)`

Grants minter role to the specified address.

Inputs

- `addr`:
    - **Control**: `onlyOwner` can call this.
    - **Constraints**: `addr` must not be authorized already.
    - **Impact**: `addr` is authorized to mint tokens.

Branches and code coverage (including function calls)

Intended branches

- Should revert if the address is already authorized.
    - □ Test coverage
- Should emit an event if the address is authorized.
    - □ Test coverage
- Should update the authorized minters mapping so that the address is authorized.
    - □ Test coverage

Negative behavior

- Should not be able to call this function if the contract is not initialized.
    - □ Negative test
- Should not be able to call this function if the caller is not the owner.
    - □ Negative test

Function: `mint(address addr, uint256 amount)`

Should allow authorized minters to mint tokens to the specified address.

Inputs

- `addr`:
  - **Control**: `onlyMinter` can call this, only if the contract is initialized.
  - **Constraints**: `addr` must not be the zero address or the contract address.
  - **Impact**: Destination address will receive tokens.
- `amount`:
  - **Control**: `onlyMinter` can call this, only if the contract is initialized.
  - **Constraints**: Amount must be greater than zero, and the `_totalSupply` + amount must not exceed the max token supply.
  - **Impact**: `addr` will receive amount tokens.

Branches and code coverage (including function calls)

Intended branches

- Should revert if the address is the zero address or the contract address.
  - ☑ Test coverage
- Should revert if the amount is zero.
  - ☑ Test coverage
- Should revert if the total supply + amount exceeds the max token supply.
  - ☑ Test coverage
- Should emit an event if the address is authorized.
  - ☑ Test coverage
- Should update the balance of the destination address, `_balances[addr] += amount;`.
  - ☑ Test coverage
- `_totalSupply` should be updated, `_totalSupply += amount;`.
  - ☑ Test coverage
- The balance of the destination address should be updated, `_balances[addr] += amount;`.
  - ☑ Test coverage

Negative behavior

- Should not be able to call this function if the contract is not initialized.
  - ☑ Negative test
- Should not be able to call this function if the caller is not a minter.

  - ☑ Negative test

Function: `revokeBurner(address addr)`

Removes burner role from the specified address.

Inputs

- `addr`:
  - **Control**: `onlyOwner` can call this.
  - **Constraints**: `addr` must be authorized already.
  - **Impact**: `addr` is no longer authorized to burn tokens.

Branches and code coverage (including function calls)

Intended branches

- Should revert if the address is not authorized.
  - ☐ Test coverage
- Should emit an event if the address is revoked.
  - ☐ Test coverage
- Should update the authorized burners mapping so that the address is no longer authorized.
  - ☐ Test coverage

Negative behavior

- Should not be able to call this function if the contract is not initialized/
  - ☐ Negative test
- Should not be able to call this function if the caller is not the owner.
  - ☐ Negative test

Function: `revokeMinter(address addr)`

Removes minter role from the specified address.

Inputs

- `addr`:
  - **Control**: `onlyOwner` can call this.
  - **Constraints**: `addr` must be authorized already.
  - **Impact**: `addr` is no longer authorized to mint tokens.

Branches and code coverage (including function calls)

Intended branches

- Should revert if the address is not authorized.
  - □ Test coverage
- Should emit an event if the address is revoked.
  - □ Test coverage
- Should update the authorized minters mapping so that the address is no longer authorized.
  - □ Test coverage

Negative behavior

- Should not be able to call this function if the contract is not initialized.
  - □ Negative test
- Should not be able to call this function if the caller is not the owner.
  - □ Negative test

5.18 File: OpenTermLoan.sol

Function: `borrowerCommitment()`

Borrower commits their collateral.

Branches and code coverage (including function calls)

Intended branches

- Takes the user's collateral and changes the stage to FUNDING_REQUIRED.
  - □ Test coverage
- Set the funding deadline.
  - □ Test coverage

Negative behavior

- Should not be callable twice.
  - □ Negative test
- Should not be callable by anyone other than the borrower.
  - □ Negative test
- Should not be called if the state is not PREAPPROVED.
  - □ Negative test

Function call analysis

- `borrowerCommitment()` → `isSecured`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether loan needs collateral.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `borrowerCommitment()` → `getInitialCollateralAmount`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Agreed upon collateral amount.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `borrowerCommitment()` → `_depositToken(CollateralToken, msg.sender, expectedDepositAmount)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `callLoan(uint256 callbackPeriodInHours, uint256 gracePeriodInHours)`

Calls loan.

Inputs

- `callbackPeriodInHours`:
  - **Control**: Full.
  - **Constraints**: >= MIN_CALLBACK_PERIOD.
  - **Impact**: Callback period in hours.
- `gracePeriodInHours`:
  - **Control**: Full.
  - **Constraints**: >= MIN_GRACE_PERIOD.
  - **Impact**: Grace period in hours.

Branches and code coverage (including function calls)

Intended branches

- Set the loan state CALLED and set the deadline.
  - □ Test coverage

Negative behaviour

- Should not be callable if the loan is not active or funded.
  - □ Negative test
- Should not be callable if the loan is already called.
  - □ Negative test
- Should not be callable by anyone other than the lender.
  - □ Negative test

Function: `changeOracle(IBasicPriceOracle newOracle)`

Changes oracle.

Inputs

- `newOracle`:
  - **Control**: Full.
  - **Constraints**: != PrevOracle.
  - **Impact**: The address of the oracle.

Branches and code coverage (including function calls)

Intended branches

- Changes the oracle.
  - □ Test coverage

Negative behavior

- Does not change oracle if loan has been called.
  - □ Negative test

Function call analysis

- `changeOracle` → `isSecured()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Checks if call is secured to check for deadline.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `claimCollateral()`

Claims collateral and allows the borrower to claim their collateral if the loan is not funded.

Branches and code coverage (including function calls)

Intended branches

- Check if the user was returned their collateral.
  - ☐ Test coverage

Negative behavior

- Do not allow for claiming of collateral if not in the FUNDING_REQUIRED stage.
  - ☐ Negative test

Function call analysis

- `claimCollateral → isSecured()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Tells whether loan needs collateral.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `claimCollateral → collateralToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: The amount of collateral in the contract.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `claimCollateral → collateralToken.transfer(borrower, currentBalance)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `claimCollateral → collateralToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Post-transfer balance used for checking if the transfer was successful.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `claimPrincipal()`

Allows the lender to claim returned principal.

Branches and code coverage (including function calls)

Intended branches

- Ensures the lender received their principal.
  - ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the lender.
  - ☐ Negative test
- Should not be callable twice
  - ☐ Negative test
- Should not be callable if loan is neither CLOSED nor ACTIVE
  - ☐ Negative test

Function call analysis

- `claimPrincipal → _withdrawPrincipalTokens(currentBalanceAtContract, lender)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `claimPrincipal → _withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `claimPrincipal → _withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr) → principtalToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Balance of principal in the open loan.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

- `claimPrincipal` → `_withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr)` → `principtalToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Pre-transfer balance of principal in the contract.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `claimPrincipal` → `_withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr)` → `principtalToken.balanceOf(recipientAddr)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Pre-transfer balance of principal of the recipient's `addr`.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `claimPrincipal` → `_withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr)` → `principalToken.transfer(recipientAddr, amountInPrincipalTokens)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `claimPrincipal` → `_withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr)` → `principtalToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Post-transfer balance of the contract.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `claimPrincipal` → `_withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr)` → `principtalToken.balanceOf(recipient)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Post-transfer balance of the recipient.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `fundLoan()`

The lender funds the loan.

Branches and code coverage (including function calls)

Intended branches

- Check that loan has been funded, state has changed, and tokens have been taken.
  - □ Test coverage

Negative behavior

- Do not allow for funding of loan if not in the FUNDING_REQUIRED stage.
  - □ Negative test
- Do not allow for funding of loan if the loan is already funded.
  - □ Negative test
- Do not allow for funding of loan if caller is not the lender.
  - □ Negative test

Function call analysis

- `fundLoan → _depositToken(principalToken, msg.sender)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `liquidate()`

Liquidates a called loan.

Branches and code coverage (including function calls)

Intended branches

- Check that liquidation works according to the grace period and transfers tokens to the lender.
  - □ Test coverage

Negative behavior

- Check that loan cannot be liquidated when not called.
  - □ Negative test

Function call analysis

- `_transferPrincipalAndCollateral`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `repayInterests()`

Borrower repays the interest on the loan.

Branches and code coverage (including function calls)

Intended branches

- Ensures repayments accumulate.
  - □ Test coverage

Negative behavior

- Ensure interest cannot be payed if not owed.
  - □ Negative test

Function call analysis

- `repayInterests` → `_enforceMaintenanceRatio()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok (means collateral ratio is not high enough / unhealthy loan).
- `repayInterests` → `getDebt()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Returns the interest owed, the minPayment.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok (means collateral ratio is not high enough / unhealthy loan).
- `repayInterests` → `_depositToken`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.

  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok (means collateral ratio is not high enough / unhealthy loan).

Function: `repayPrincipal(uint256 paymentAmountInTokens)`

Repays the principal portion of the loan (borrower).

Inputs

- `paymentAmountInTokens`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount of principal to payback.

Branches and code coverage (including function calls)

Intended branches

- Repayments accumulate.
  - □ Test coverage

Negative behavior

- Cannot pay back after loan is closed.
  - □ Test coverage

Function call analysis

- `repayPrincipal → _enforceMaintenanceRatio()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok (means collateral ratio is not high enough / unhealthy loan).
- `repayPrincipal → getDebt()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: `debtAmount`, interest owed, and max payment amount.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `repayPrincipal → _depositToken(principalToken, msg.sender, paymentAmountInTokens)`:
  - **What is controllable?**: `paymentAmountInTokens`.

  - **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?**: Ok.
- `repayPrincipal → _depositToken(principalToken, msg.sender, paymentAmountInTokens)`:
  - **What is controllable?**: `paymentAmountInTokens`.
  - **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?**: Ok.
- `repayPrincipal → _closeLoan()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `withdraw()`

The borrower withdraws the funds.

Branches and code coverage (including function calls)

Intended branches

- Check that the borrower is able to withdraw the funds and receives them.
  - □ Test coverage (tests commented)
- Check that the loan is active after withdrawal.
  - □ Test coverage (tests commented)

Negative behavior

- Do not allow for withdrawal of funds if not in the FUNDED stage.
  - □ Negative test
- Should not be callable by anyone other than the borrower.
  - □ Negative test

Function call analysis

- `withdraw → _enforceMaintenanceRatio`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Can

    prevent borrower from withdrawing.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `withdraw → _withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `withdraw → _withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr) → principtalToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Balance of principal in the open loan.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `withdraw → _withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr) → principtalToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Pre-transfer balance of principal in the contract.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `withdraw → _withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr) → principtalToken.balanceOf(recipientAddr)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Pre-transfer balance of principal of the recepient's `addr`.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `withdraw → _withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr) → principalToken.transfer(recipientAddr, amountInPrincipalTokens)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `withdraw → _withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr) → principtalToken.balanceOf(this)`:

  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Post-transfer balance of the contract.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `withdraw → _withdrawPrincipalTokens(amountInPrincipalTokens, recipientAddr) → principtalToken.balanceOf(recipient)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Post-transfer balance of the recipient.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

5.19 File: ReceiptToken.sol

Function: `changeMaxSupply(uint256 newValue)`

Changes the max supply of the token.

Inputs

- `newValue`:
  - **Control**: Only the owner can call this function.
  - **Constraints**: Should be greater than the current supply.
  - **Impact**: More tokens can be minted.

Branches and code coverage (including function calls)

Intended branches

- Should update the max supply.
  ☑ Test coverage
- Ensure that the new value is greater than the current supply.
  □ Test coverage

Negative behaviour

- Should not be able to call this function if caller is not the owner.
  □ Negative test

Function: `initialize(address newOwner, uint256 initialMaxSupply)`

Initializes the contract, which is upgradable. It basically calls `_initReceiptToken(newOwner, initialMaxSupply)`, so we will analyze that here.

Inputs

- `newOwner`:
  - **Control**: Anyone can call this function; it can only be called once.
  - **Constraints**: N/A.
  - **Impact**: Sets the owner of the contract.
- `initialMaxSupply`:
  - **Control**: Anyone can call this function; it can only be called once.
  - **Constraints**: N/A.
  - **Impact**: Sets the initial max supply of the token.

Branches and code coverage (including function calls)

Intended branches

- Should initialize all the variable, as well as set the initialized flag.
  - □ Test coverage
- Ensure that all initializer functions from underlying contracts are called.
  - □ Test coverage

Negative behavior

- Should not be able to call this function multiple times.
  - □ Negative test

5.20 File: SpotMarginAccount.sol

Function: `PartialUnwind(IERC20 token, uint256 amount, byte[] swapCallData)`

Partial unwinding of the position.

Inputs

- `token`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Token to unwind.

- `amount`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount to unwind.
- `swapCallData`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Swap bytes for the Paraswap exchange.

Branches and code coverage (including function calls)

Intended branches

- Unwinds according to amounts specified.
  ☑ Test coverage

Negative behaviour

- Cannot unwind more than the entire position.
  □ Test coverage

Function call analysis

- `partialUnwind → _partialUnwind(token, amount, swapCallData)`:
  - **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?**: Ok.
- `partialUnwind → _partialUnwind(token, amount, swapCallData) → _swap(token, amount, swapCallData)`:
  - **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?**: Ok.
- `partialUnwind → _partialUnwind(token, amount, swapCallData) → loanContract.principalToken()`:
  - **What is controllable?**: Everything.
  - **If return value controllable, how is it used and how can it go wrong?**: The principal token.
  - **What happens if it reverts, reenters, or does other unusual control flow?**:

    Ok.

- `partialUnwind → _partialUnwind(token, amount, swapCallData) → principalToken.balanceOf(this)`:
- **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Balance of principal token post swap.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `PartialUnwind → repayLoanInterest()`:
  - **What is controllable?**: Nothing.
  - **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 cannot close their debts.
- `PartialUnwind → repayLoanPrincipal()`:
  - **What is controllable?**: Nothing.
  - **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 cannot close their debts.

Function: `acceptLoan()`

Accepts the loan (borrower puts down collateral).

Branches and code coverage (including function calls)

Intended branches

- Takes collateral from the spot account and changes the loan state to FUNDING_REQUIRED.
  - □ Test coverage

Negative behaviour

- Should not be callable by anyone other than the operator.
  - □ Negative test
- Should not be callable twice.
  - □ Negative test

Function call analysis

- `acceptLoan() -> _acceptLoan()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `acceptLoan() -> _acceptLoan() -> loanContract.principalToken()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: `PrincipalToken`.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `acceptLoan() -> _acceptLoan() -> token.approve(loanContract, type(uint256).max)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `acceptLoan() -> _acceptLoan() -> loanContract.borrowerCommitment()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `deployTwap(address ownerAddr, address traderAddr, address depositorAddr, IERC20 sellingToken, IERC20 buyingToken)`

Deploys a TWAP order.

Inputs

- `ownerAddr`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Owner.
- `traderAddr`:
  - **Control**: Full.

  - **Constraints**: None.
  - **Impact**: Trader.
- `depositorAddr`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Depositor.
- `sellingToken`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Sell token (base/quote).
- `buyingToken`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Buy token (base/quote).

Branches and code coverage (including function calls)

Intended branches

- Ensure it creates a new TWAP, initializes it, and transfers ownership correctly.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the operator.
  - □ Negative test
- Should not be callable unless the contract was initialized.
  - □ Negative test

Function call analysis

- `deployTwap → _deployTwap`:
  - **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?**: Ok.
- `deployTwap → _deployTwap → new TwapOrder()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: The `TwapOrder` instance.

  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `deployTwap → _deployTwap → instance.initialize(traderAddr, depositorAddr, sellingToken, buyingToken)`:
  - **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?**: Ok.
- `deployTwap → _deployTwap → instance.transferOwnership(ownerAddr)`:
  - **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?**: Ok.

Function: `deposit(address token, uint256 amount)`

Deposits token into the SpotMarginAccount.

Inputs

- `token`:
  - **Control**: Full.
  - **Constraints**: != 0.
  - **Impact**: The token.
- `amount`:
  - **Control**: Full.
  - **Constraints**: > 0.
  - **Impact**: The amount to deposit.

Branches and code coverage (including function calls)

Intended branches

- Deposits the specificed amount of tokens.
  - □ Test coverage

Negative behavior

- Deposits more than the specificed amount of tokens.
  - □ Negative test

Function call analysis

- deposit → _deposit:

- **What is controllable?**: token.
  - **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.
- deposit → _deposit → token.transferFrom(msg.sender, this, amount):
  - **What is controllable?**: amount and token.
  - **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.

Function: `fullUnwind(IERC20 token, uint256 amount, byte[] swapCallData)`

Fully unwinds a position.

Inputs

- token:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Token to unwind.
- amount:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Amount to unwind.
- swapCallData:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Swap bytes for the PARASWAP exchange.

Branches and code coverage (including function calls)

Intended branches

- Unwinds the position fully, pays interest first and principal second.
  - □ Test coverage

Negative behavior

- Does not unwind the position.
  - □ Negative test

Function call analysis

- `fullUnwind → _fullUnwind(token, amount, swapCallData)`:
  - **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?**: Ok.
- `fullUnwind → _fullUnwind(token, amount, swapCallData) → _swap(token, amount, swapCallData)`:
  - **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?**: Ok.
- `fullUnwind → _fullUnwind(token, amount, swapCallData) → loanContract.principalToken()`:
  - **What is controllable?**: Everything.
  - **If return value controllable, how is it used and how can it go wrong?**: The principal token.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `fullUnwind → _fullUnwind(token, amount, swapCallData) → principalToken.balanceOf(this)`:

- **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Balance of principal token post swap.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `glpUnwind → repayLoanInterest()`:
  - **What is controllable?**: Nothing.
  - **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 cannot close their debts.
- `glpUnwind → repayLoanPrincipal()`:
  - **What is controllable?**: Nothing.

  - **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 cannot close their debts.

Function: `transferMargin(IERC20 token, address toSubAccount, uint256 marginAmount)`

Transfers margin.

Inputs

- `token`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The token.
- `toSubAccount`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: `subAccount`.
- `marginAmount`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Margin.

Branches and code coverage (including function calls)

Intended branches

- Transfers to the specific sub account.
  - □ Test coverage

Negative behavior

- Should not transfer to the wrong account.
  - □ Negative test

Function call analysis

- `transferMargin(token, toSubAccount, marginAmount) → _transferMargin(token, toSubAccount, marginAmount)`:
  - **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?**: Ok.
- `transferMargin(token, toSubAccount, marginAmount) → _transferMargin(token, toSubAccount, marginAmount) → counterPartyRegistry.getCounterParty(toSubAccount)`:
  - **What is controllable?**: `toSubAccount`.
  - **If return value controllable, how is it used and how can it go wrong?**: Check whether the counter party exists.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `transferMargin(token, toSubAccount, marginAmount) → _transferMargin(token, toSubAccount, marginAmount) → counterPartyRegistry.getMaxMarginTransferAmount(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Check if the transfer amount is > than the max.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `transferMargin(token, toSubAccount, marginAmount) → _transferMargin(token, toSubAccount, marginAmount) → token.approve(toSubAccount, marginAmount)`:
  - **What is controllable?**: Everything (including token).
  - **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?**: Ok.
- `transferMargin(token, toSubAccount, marginAmount) → _transferMargin(token, toSubAccount, marginAmount) → token.transfer(toSubAccount, marginAmount)`:
  - **What is controllable?**: Everything (including token).
  - **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?**: Ok.
- `transferMargin(token, toSubAccount, marginAmount) → _transferMargin(token, toSubAccount, marginAmount) → token.approve(toSubAccount, 0)`:
  - **What is controllable?**: `toSubAccount`.
  - **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?**: Ok.

Function: `transferOriginationFee(IERC20 token, uint256 amount)`

Transfers the origination fee to the fee collector.

Inputs

- `token`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: tThe token to transfer.
- `amount`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The amount of fee.

Branches and code coverage (including function calls)

Intended branches

- Updates the swapContractManager and transfers the origination fee.
  ☐ Test coverage

Negative behaviour

- Should not be callable by anyone other than the swap contract manager.
  ☐ Negative test
- Should not be callable if contract was not initialized.
  ☐ Negative test

Function call analysis

- `transferOrigniationFee(token, amount) → _transferOriginationFee(token,amount)`:
  - **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?**: Ok.
- `transferOrigniationFee(token, amount) → _transferOriginationFee(token,amount) → counterPartyRegistry.getSwapContractManager(msg.sender)`:

- **What is controllable?**: Nothing.
    - **If return value controllable, how is it used and how can it go wrong?**: Check if `msg.sender` is a valid swap contract manager.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok
- `transferOrigniationFee(token, amount) → _transferOriginationFee(token,amount) →token.approve(msg.sender, 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?**: Ok.
- `transferOrigniationFee(token, amount) → _transferOriginationFee(token,amount) →token.transfer(feeCollector, 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?**: Ok.
- `transferOrigniationFee(token, amount) → _transferOriginationFee(token,amount) → token.approve(msg.sender, 0)`:
    - **What is controllable?**: Nothing.
    - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `withdrawToOwner(address token, uint256 amount)`

Withdraws to owner.

Inputs

- `token`:
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: The tokens to withdraw from the account.
- `amount`:
    - **Control**: Full.
    - **Constraints**: None.

    - **Impact**: The amount of tokens to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Withdraws tokens to owner.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the operator.
  - □ Negative test

Function call analysis

- `withdrawToOwner → _withdrawToOwner()`:
  - **What is controllable?**: `token` and `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?**: Ok.
- `withdrawToOwner → _withdrawToOwner() → token.transfer(owner, amount)`:
  - **What is controllable?**: `token` and `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?**: Ok.

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

Withdraw from account.

Inputs

- `token`:
  - **Control**: Full.
  - **Constraints**: != 0.
  - **Impact**: The token to withdraw from the account.
- `amount`:
  - **Control**: Full.
  - **Constraints**: > 0.
  - **Impact**: The amount of tokens to withdraw.

Branches and code coverage (including function calls)

Intended branches

- Withdraws the correct amount.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  - □ Negative test
- Should not be callable if contract was not initialized.
  - □ Negative test

Function call analysis

- `withdraw → _withdraw()`:
  - **What is controllable?**: `token` and `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?**: Ok.
- `withdraw → _withdraw() → token.transfer(this, msg.sender, amount)`:
  - **What is controllable?**: `token` and `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?**: Ok.

5.21 File: TotalReturnSwapContract.sol

Function: `closePosition(uint256 receiverMarginBips, uint256 payerMarginBips, uint256 newUnderlyingPrice)`

Closes the total return contract position.

- `receiverMarginBips`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Difference in receiver margin in bips.
- `payerMarginBips`:
  - **Control**: Full.

  - **Constraints**: None.
  - **Impact**: Difference in payer margin in bips.
- `newUnderlyingPrice`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The new price of the asset in the contract.

Branches and code coverage (including function calls)

Intended branches

- Close the position by transferring the remaining tokens.
  - ☑ Test coverage
- Set the notional value of the contract to 0.
  - □ Test coverage

Negative behavior

- Should not shadow the `newUnderlyingPrice` variable.
  - □ Negative test
- Should not be callable multiple times.
  - □ Negative test
- Should not be callable by anyone other than the operator.
  - □ Negative test

Function call analysis

- `closePosition → updateVariationMarginBips()`:
  - **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?**: Ok.
- `closePosition → updatePositions()`:
  - **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?**: Ok.
- `closePosition → updateNotionalValue()`:
  - **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?**: Ok.

Function: `updateNotionalValue(uint256 newInitialUnderlyingPrice, uint256 newSwapContractUnits)`

Updates the value of the underlying asset of the contract.

Inputs

- `newInitialUnderlyingPrice`:
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: The new value of the asset.
- `newSwapContractUnits`:
    - **Control**: Full.
    - **Constraints**: None.
    - **Impact**: The units (units x value = total price).

Branches and code coverage (including function calls)

Intended branches

- Updates the price of a unit.
    ☐ Test coverage
- Updates the number of units.
    ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the operator.
    ☐ Negative test

Function call analysis

- `updateNotionalValue → _updateSwapContractManagerData(newInitialUnderlyingPrice, newSwapContractUnits)`:
    - **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?**: Ok.

- updateNotionalValue → _updateSwapContractManagerData(newInitialUnderlyingPrice, newSwapContractUnits) → swapContractManager.updateSwapContractNotionalValue(swapIndex, newInitialUnderlyingPrice, newSwapContractUnits):
  - **What is controllable?**: newInitialUnderlyingPrice and newSwapContractUnits.
  - **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?**: Ok.

Function: `updatePositions(uint256 underlyingPrice)`

Updates the positions.

Inputs

- underlyingPrice:
  - **Control**: Full.
  - **Constraints**: > 0.
  - **Impact**: Update positions based on the price.

Branches and code coverage (including function calls)

Intended branches

- Updates the positions with the new price and transfers the margins.
  ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the operator.
  ☐ Negative test

Function call analysis

- updatePositions → _updatePositionGreater():
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- updatePositions → _updatePositionLesser():
  - **What is controllable?**: Nothing.

  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `updatePositions → _updatePositionGreater() → _updateMargins(swapContractData.payer/receiverSubAccount, swapContractData.payer/receiverSubAccount, requiredReceiver/PayerMargin)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `updatePositions → _updatePositionGreater() → _updateMargins(swapContractData.payer/receiverSubAccount, swapContractData.payer/receiverSubAccount, requiredReceiver/PayerMargin)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `updateVariationMarginBips(uint256 receiverMarginBips, uint256 payerMarginBips)`

Updates the variation margin bips.

Inputs

- `receiverMarginBips`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Receiver margin bips.
- `payerMarginBips`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Payer margin bips.

Branches and code coverage (including function calls)

Intended branches

- Check that the relevant state variables were set.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the operator.
  - □ Negative test

5.22 File: TwapOrder.sol

Function: `cancelOrder()`

Cancels the order and changes the state such that other functions cannot be used.

Branches and code coverage (including function calls)

Intended branches

- Cancels the order and returns the remaining tokens.
  - □ Test coverage

Negative behavior

- Ensure the order can no longer be used.
  - □ Negative test

Function call analysis

- `cancelOrder → _closeOrder()`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `cancelOrder → _closeOrder() → sellingToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Amount of selling tokens left in the `TwapOrder` to return to the depositor.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `cancelOrder → _closeOrder() → buyingToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**:

    Amount of buying tokens left in the `TwapOrder` to return to the depositor.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `cancelOrder → _closeOrder() → sellingToken.transfer(depositorAddress, sellingTokenBalance)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `cancelOrder → _closeOrder() → buyingToken.transfer(depositorAddress, buyingTokenBalance)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `initialize(address traderAddr, address depositorAddr, IERC20 sellingToken, IERC20 buyingToken)`

Initializes the TWAP order.

Inputs

- `traderAddr`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Address of the trader.
- `depositorAddr`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Address of the depositor.
- `sellingToken`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Selling tokens.
- `buyingToken`:
  - **Control**: Full.
  - **Constraints**: None.

    - **Impact**: Buying tokens.

Branches and code coverage (including function calls)

Intended branches

- Check that all relevant state variables were set and initialized.
  - ☐ Test coverage

Negative behavior

- Should not be callable multiple times.
  - ☐ Negative test

Function call analysis

- `initialize` → `_initializationCompleted`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `openOrder(uint256 durationInMins, uint256 targetQty, uint256 chunkSize, uint256 maxPriceLimit)`

Opens an order.

Inputs

- `durationInMins`:
  - **Control**: Full.
  - **Constraints**: >= 5 minutes.
  - **Impact**: The duration in minutes.
- `targetQty`:
  - **Control**: Full.
  - **Constraints**: > 0.
  - **Impact**: Target amount.
- `chunkSize`:
  - **Control**: Full.
  - **Constraints**: > 0.
  - **Impact**: `chunkSize`.
- `maxPriceLimit`:

  - **Control**: Full.
  - **Constraints**: > 0.
  - **Impact**: Slippage check.

Branches and code coverage (including function calls)

Intended branches

- Opens an order and approves the Augustus token transfer proxy.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the operator.
  - □ Negative test
- Should not be called if contract was not initialized.
  - □ Negative test
- Should not be called if the order is already open.
  - □ Negative test

Function call analysis

- `openOrder` → `_approveProxy`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Discarded.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

Function: `swap(uint256 sellQty, uint256 buyQty, byte[] payload)`

The TWAP swap order.

Inputs

- `sellQty`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Max sell qty / limit/ slippage.
- `buyQty`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Max buy qty / limit/ slippage.

- `payload`:
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: Byte payload to be sent to Augustus.

Branches and code coverage (including function calls)

Intended branches

- Check that swap works as intended.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the trader.
  - □ Negative test
- Should not be callable if the contract was not initialized.
  - □ Negative test

Function call analysis

- `sellingToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Pre-swap selling token balance.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `BuyingToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Pre-swap buying token balance.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `AUGUSTUS_SWAPPER_ADDR.call(payload)`:
  - **what is controllable?**: `payload`.
  - **if return value controllable, how is it used and how can it go wrong?**: Used to check the low-level call for reversion.
  - **what happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `sellingtoken.balanceof(this)`:
  - **what is controllable?**: Nothing.
  - **if return value controllable, how is it used and how can it go wrong?**: Post-

swap selling token balance.
  - **what happens if it reverts, reenters, or does other unusual control flow?**: Ok.
- `BuyingToken.balanceOf(this)`:
  - **What is controllable?**: Nothing.
  - **If return value controllable, how is it used and how can it go wrong?**: Post-swap buying token balance.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Ok.

5.23 File: Vault.sol

Function: `changeApr(uint256 newApr)`

Allows the owner to change the APR.

Inputs

- `newApr`:
  - **Control**: The owner has full control over this input.
  - **Constraints**: The `newApr` must be > 0.
  - **Impact**: The APR will be updated to the `newApr`.

Branches and code coverage (including function calls)

Intended branches

- The APR should be updated to the `newApr`.
  ☐ Test coverage
- The `OnAprChanged` event should be emitted.
  ☐ Test coverage
- Assure that the APR is > 100 and APR < 10000.
  ☐ Test coverage

Negative behavior

- Should not allow anyone to call this function except the owner.
  ☐ Negative test

Function: `claimDailyInterest()`

Allows the owner or controller to claim the daily interest promised per APR.

Branches and code coverage (including function calls)

Intended branches

- The daily interest should be transferred from the yield reserve to the vault.
  - ☐ Test coverage
- The `OnDailyInterestClaimed` event should be emitted.
  - ☐ Test coverage
- The balance of the vault should be updated to account for the newly claimed interest.
  - ☐ Test coverage

Negative behavior

- Should not allow anyone to call this function except the owner or controller.
  - ☐ Negative test
- Should not be callable multiple times in the same day.
  - ☐ Negative test

Function call analysis

- `IDeployable(yieldReserveAddress).claim(dailyInterestAmount)`:
  - **What is controllable?**: Technically, the `yieldReserveAddress` is controllable, but it is set in the constructor and cannot be changed.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: Means that the yield reserve does not have enough funds to pay the interest, and the vault will be left with less funds than it should have.

Function: `deposit(uint256 depositAmount)`

Allows a user to deposit funds in the vault and receive receipt tokens in exchange.

Inputs

- `depositAmount`:
  - **Control**: Full control (can be set by anyone).
  - **Constraints**: Must be `>= minDepositAmount`. Also, the `msg.sender` must have enough funds to cover the deposit.
  - **Impact**: The deposit amount will be added to the total amount deposited in the current period.

Branches and code coverage (including function calls)

Intended branches

- Ensure that the deposit amount is `>=` `minDepositAmount`.
  - □ Test coverage
- Ensure that the `msg.sender` has enough funds to cover the deposit.
  - □ Test coverage
- Wake up the reentrancy guard.
  - □ Test coverage
- Refresh the current timelime, if needed.
  - □ Test coverage
- Calculate the number of receipt tokens to mint. It is important that the token price is up to date and that it is not abused by the owner.
  - □ Test coverage
- The underlying balance of the vault should increase by `depositAmount`.
  - □ Test coverage
- The total amount deposited in the current period should increase by `depositAmount`.
  - □ Test coverage
- The underlying balance of the `msg.sender` should decrease by `depositAmount`.
  - □ Test coverage
- The user should receive `receiptTokensToMint` receipt tokens, so their balance should increase by `receiptTokensToMint`.
  - □ Test coverage
- The total supply of receipt tokens should increase by `receiptTokensToMint`.
  - □ Test coverage

Negative behavior

- Should not allow user to pay less than `minDepositAmount`.
  - □ Negative test
- Should not allow user to mention a deposit amount that is greater than the amount of funds they have on their wallet.
  - □ Negative test
- Should not allow the owner to abuse the vault by setting a token price that is too low or too high.
  - □ Negative test

Function call analysis

- `underlyingTokenInterface.balanceOf(msg.sender)`:

- **What is controllable?**: The `msg.sender` can be controlled by the user.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value is used to ensure that the user has enough funds to cover the deposit. Should not be entirely controllable by the user, as the user can only spend funds that they have on their wallet, and they should not have minting rights for the underlying token.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The function will revert due to reentrancy mutex.
- `underlyingTokenInterface.allowance(msg.sender, address(this))`:

- **What is controllable?**: The `msg.sender` can be controlled by the user.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value is used to ensure that the user has approved this contract to spend the amount specified. Should not be entirely controllable by the user, as the user can only spend funds that they have on their wallet, and they should not have minting rights for the underlying token.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The function will revert due to reentrancy mutex.
- `_receiptToken.canMint(numberOfReceiptTokens)`:

- **What is controllable?**: The `numberOfReceiptTokens` can be controlled by the user.
  - **If return value controllable, how is it used and how can it go wrong?**: The return value is used to ensure that the user can receive the number of receipt tokens they are requesting. Should not be entirely controllable by the user, as the user can only spend funds that they have on their wallet, and they should not have minting rights for the underlying token, and they should not have the right to alter the token price.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The function will revert due to reentrancy mutex.
- `underlyingTokenInterface.balanceOf(address(this));`:

- **What is controllable?**: The balance of the vault can be artificially inflated by anyone that transfers funds to the vault.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.
- `underlyingTokenInterface.transferFrom(msg.sender, address(this), depositAmount)`:

- **What is controllable?**: The `msg.sender` and `depositAmount` can be controlled by the user.
  - **If return value controllable, how is it used and how can it go wrong?**: The return tells if the transfer was successful. Should not be entirely control-

lable by the user, as the user can only spend funds that they have on their wallet, and they should not have minting rights for the underlying token.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: The function will revert due to reentrancy mutex.
- `_receiptToken.mint(msg.sender, numberOfReceiptTokens)`:
    - **What is controllable?**: The `msg.sender` can be controlled by the user.
    - **If return value controllable, how is it used and how can it go wrong?**: N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: The function will revert due to reentrancy mutex.

Function: `emergencyWithdraw(address destinationAddr)`

Allows the owner to withdraw all funds from the vault.

Inputs

- `destinationAddr`:
    - **Control**: The owner has full control over this input.
    - **Constraints**: The `destinationAddr` must be != 0 and != the vault address.
    - **Impact**: The funds will be sent to the `destinationAddr`.

Branches and code coverage (including function calls)

Intended branches

- The balance of the vault should become 0.
    - □ Test coverage
- The funds should be sent to the `destinationAddr`.
    - □ Test coverage
- The `emergencyWithdraw` event should be emitted.
    - □ Test coverage
- Assumes the owner can return the funds and the vault should work as expected after.
    - □ Test coverage
- Expects that this function will lock the vault and other functions will not work.
    - □ Test coverage

Negative behavior

- Should not allow anyone to call this function except the owner.
    - □ Negative test

Function call analysis

- underlyingTokenInterface.balanceOf(address(this)):

- **What is controllable?**: The balance can be artificially increased by manually transferring tokens to the vault.
  - **If return value controllable, how is it used and how can it go wrong?**: Return used to know how much can be withdrawn.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.
- underlyingTokenInterface.transfer(destinationAddr, currentBalance):
  - **What is controllable?**: The destinationAddr can be controlled by the caller.
  - **If return value controllable, how is it used and how can it go wrong?**: Transfer did not work.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: The function will revert due to reentrancy mutex. Should not be a huge issue though, since the entire balance is transferred anyway.

Function: initialize(address ownerAddr, address controllerAddr, address receiptTokenInterface, address eip20Interface, uint256 initialApr, uint256 initialTokenPrice, uint256 initialMinDepositAmount, uint256 flatFeePerc, address feesAddr)

Function that initializes the contract. Its code is written in _initVault basically.

Inputs

- ownerAddr:
  - **Control**: Full control; can be set by anyone.
  - **Constraints**: Must be a valid address, different than controllerAddr.
  - **Impact**: The owner of the contract will be set.
- controllerAddr:
  - **Control**: Full control; can be set by anyone.
  - **Constraints**: Must be a valid address, different than ownerAddr.
  - **Impact**: The controller of the contract will be set.
- receiptTokenInterface:
  - **Control**: Full control; can be set by anyone.
  - **Constraints**: N/A.
  - **Impact**: The receipt token interface will be set.
- eip20Interface:
  - **Control**: Full control; can be set by anyone.

  - **Constraints**: The inherent constraints of the `IERC20` interface.
  - **Impact**: The underlying token interface will be set.
- `initialApr`:
  - **Control**: Full control; can be set by anyone.
  - **Constraints**: N/A; it should have some constraints though (e.g., > 100 and < 10,000).
  - **Impact**: The APR will be set.
- `initialTokenPrice`:
  - **Control**: Full control; can be set by anyone.
  - **Constraints**: N/A; should have some constraints though (e.g., > 0).
  - **Impact**: The initial token price will be set.
- `initialMinDepositAmount`:
  - **Control**: Full control; can be set by anyone.
  - **Constraints**: Checked that it is > 0.
  - **Impact**: The initial minimum deposit amount will be set.
- `flatFeePerc`:
  - **Control**: Full control; can be set by anyone.
  - **Constraints**: N/A; should have some constraints though (e.g., > 0 and < 100).
  - **Impact**: The flat fee percentage will be set.
- `feesAddr`:
  - **Control**: Full control; can be set by anyone.
  - **Constraints**: Checked that != address(0).
  - **Impact**: The fees address will be set.

Branches and code coverage (including function calls)

Intended branches

- Should ensure that `receiptTokenInterface` is not `eip20Interface`.
  □ Test coverage
- Ensure APR is > 100 and < 10,000.
  □ Test coverage
- Ensure `initialMinDepositAmount` > 0.
  □ Test coverage
- Ensure `feesAddr` != address(0).
  □ Test coverage
- All variables should be initialized.
  □ Test coverage

Negative behavior

- Ensure that `_records[currentPeriod]` does not already exist. That could happen if it is a new year whose period overlaps with the existing period.
  - □ Negative test
- Ensure that it cannot be called multiple times.
  - □ Negative test

Function: `lockCapital()`

Moves the deployable capital from the vault to the yield reserve.

Branches and code coverage (including function calls)

Intended branches

- Deployable capital should be transferred to the yield reserve.
  - □ Test coverage
- The `OnCapitalLocked` event should be emitted.
  - □ Test coverage
- Should update some states of the vault to account for how much has been deployed.
  - □ Test coverage

Negative behavior

- Should not allow anyone to call this function except the owner.
  - □ Negative test
- Should not allow the vault to be left with < 10% of deposited amount.
  - □ Negative test
- Should not allow `lock` -> `withdraw` -> `lock` again to be abused by the owner.
  - □ Negative test

Function call analysis

- `underlyingTokenInterface.transfer(yieldReserveAddress, maxDeployableAmount)`:
  - **What is controllable?**: N/A.
  - **If return value controllable, how is it used and how can it go wrong?**: N/A.
  - **What happens if it reverts, reenters, or does other unusual control flow?**: N/A.

Function: `setFeeAddress(address addr)`

Sets the address for collecting fees.

Inputs

- `addr`:
  - **Control**: Full control; can be set either by the owner or the controller.
  - **Constraints**: Must be a valid address, different than `address(0)` and different than `address(this)`.
  - **Impact**: The fees address will be set.

Branches and code coverage (including function calls)

Intended branches

- Ensure that the fees address is different than `address(0)` and different than `address(this)`.
  - □ Test coverage
- Set the fees address to `addr`.
  - □ Test coverage

Negative behavior

- Should not DOS the vault. Change this design to redeem the fees as a separate transaction, rather than transferring them to the fees address.
  - □ Negative test

Function: `setFlatWithdrawalFee(uint256 newFeeWithMultiplier)`

Sets a new flat fee for withdrawals.

Inputs

- `newFeeWithMultiplier`:
  - **Control**: Full control; can be set either by the owner or the controller.
  - **Constraints**: N/A.
  - **Impact**: The flat fee for withdrawals will be set.

Branches and code coverage (including function calls)

Intended branches

- Ensure that the new fee is >= 0.
  - □ Test coverage

- Set the new fee to `newFeeWithMultiplier`.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner or the controller.
  - □ Negative test

Function: `setInvestmentPercent(uint8 newPercent)`

Allows the owner to update the investment percent.

Inputs

- `newPercent`:
  - **Control**: The owner has full control over this input.
  - **Constraints**: The `newPercent` must be > 0 and < 100.
  - **Impact**: The investment percent will be updated to the `newPercent`.

Branches and code coverage (including function calls)

Intended branches

- The investment percent should be updated to the `newPercent`.
  - □ Test coverage
- The `OnInvestmentPercentChanged` event should be emitted.
  - □ Test coverage
- Ensure that the `newPercent` is > 0 and < 100.
  - □ Test coverage

Negative behavior

- Should not allow anyone to call this function except the owner.
  - □ Negative test

Function: `setMinDepositAmount(uint256 minAmount)`

Sets the minimum allowed deposit amount.

Inputs

- `minAmount`:
  - **Control**: Full control; can be set either by the owner or the controller.
  - **Constraints**: Must be > 0.

  - **Impact**: The minimum deposit amount will be set.

Branches and code coverage (including function calls)

Intended branches

- Ensure that the minimum deposit amount is > 0.
  - ☐ Test coverage
- Set the minimum deposit amount to `minAmount`.
  - ☐ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner or the controller.
  - ☐ Negative test

Function: `setTokenPrice(uint256 newTokenPrice)`

Allows the owner to set the token price.

Inputs

- `newTokenPrice`:
  - **Control**: The owner has full control over this input.
  - **Constraints**: The `newTokenPrice` must be > 0.
  - **Impact**: The token price will be updated to the `newTokenPrice`.

Branches and code coverage (including function calls)

Intended branches

- The token price should be updated to the `newTokenPrice`.
  - ☐ Test coverage
- The `OnTokenPriceChanged` event should be emitted.
  - ☐ Test coverage

Negative behavior

- Should not impact the current states of withdrawals and deposits.
  - ☐ Negative test
- Should not allow anyone to call this function except the owner.
  - ☐ Negative test
- Should not be abused by the owner to manipulate the token price so that the users can withdraw less or deposit more than they should.
  - ☐ Negative test

Function: `setTotalDepositedAmount(uint256 newAmount)`

Change the total amount deposited in the vault.

Inputs

- `newAmount`:
  - **Control**: Full control; can be set only by the owner.
  - **Constraints**: Must be > 0.
  - **Impact**: The total amount deposited in the vault will be set.

Branches and code coverage (including function calls)

Intended branches

- Ensure that the new amount is > 0.
  - □ Test coverage
- Ensure that the total amount deposited is 0.
  - □ Test coverage
- Set the total amount deposited to `newAmount`.
  - □ Test coverage

Negative behavior

- Should not be callable by anyone other than the owner.
  - □ Negative test
- Should not be used in a non-migration scenario.
  - □ Negative test

Function: `setYieldReserveAddress(address addr)`

Sets the address of the yield reserve.

Inputs

- `addr`:
  - **Control**: Full control; can be set either by the owner or the controller.
  - **Constraints**: Must be a valid address, different than `address(0)` and different than `address(this)`.
  - **Impact**: The yield reserve address will be set.

Branches and code coverage (including function calls)

Intended branches

- Ensure that the yield reserve itself has set the vault as its vault.
  - □ Test coverage

Negative behavior

- Should not impact any other states of the vault nor the states of the previous yield reserve.
  - □ Negative test
- Should not leave the vault in an inconsistent state.
  - □ Negative test

Function: `withdraw(uint256 receiptTokenAmount)`

Allows a user to withdraw their underlying token from the vault, burning the receipt token in the process.

Inputs

- `receiptTokenAmount`:
  - **Control**: Fully controlled by the user.
  - **Constraints**: Must be greater than 0, and the user must have enough receipt tokens to cover the withdrawal.
  - **Impact**: The receipt token balance of the user will be reduced by the amount withdrawn.

Branches and code coverage (including function calls)

Intended branches

- Ensure the user has enough receipt tokens to cover the withdrawal.
  - □ Test coverage
- Burn the receipt tokens from the user's balance, so their receipt token balance will be reduced by the amount withdrawn.
  - □ Test coverage
- Transfer the underlying token from the vault to the user's wallet, calculating the amount to transfer based on the number of receipt tokens they are burning and the current token price.
  - □ Test coverage
- Emit an event to indicate that the user has withdrawn from the vault.
  - □ Test coverage
- Update the total deposited amount for the current period.
  - □ Test coverage
- The underlying token balance of the vault should be reduced by the amount

withdrawn.
    - □ Test coverage

Negative behavior

- The user attempts to withdraw zero receipt tokens. This should fail.
    - □ Negative test
- The user attempts to withdraw more receipt tokens than they have. This should fail.
    - □ Negative test
- The user attempts to withdraw more receipt tokens than the vault has. This should fail.
    - □ Negative test
- The fee tranfer should never fail; it will lead to DOS. Switch to a redeem function for the fee.
    - □ Negative test

Function call analysis

- `_receiptToken.balanceOf(msg.sender)`:

- **What is controllable?**: The `msg.sender` can be controlled by the user.
    - **If return value controllable, how is it used and how can it go wrong?**: The return value is used to determine if the user has enough receipt tokens to cover the withdrawal.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: The function will revert due to reentrancy mutex.
- `_receiptToken.burn(msg.sender, receiptTokenAmount)`:
    - **What is controllable?**: The `msg.sender` and `receiptTokenAmount` can be controlled by the user.
    - **If return value controllable, how is it used and how can it go wrong?**: N/A.
    - **What happens if it reverts, reenters, or does other unusual control flow?**: The function will revert due to reentrancy mutex.

6 Audit Results

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

During our audit, we discovered five findings. Of these, one was high risk, two were medium risk, and two were low risk. Fractal acknowledged all findings and implemented fixes.

6.1 Disclaimers

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 additional code added to the assessed 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 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.