Zellic

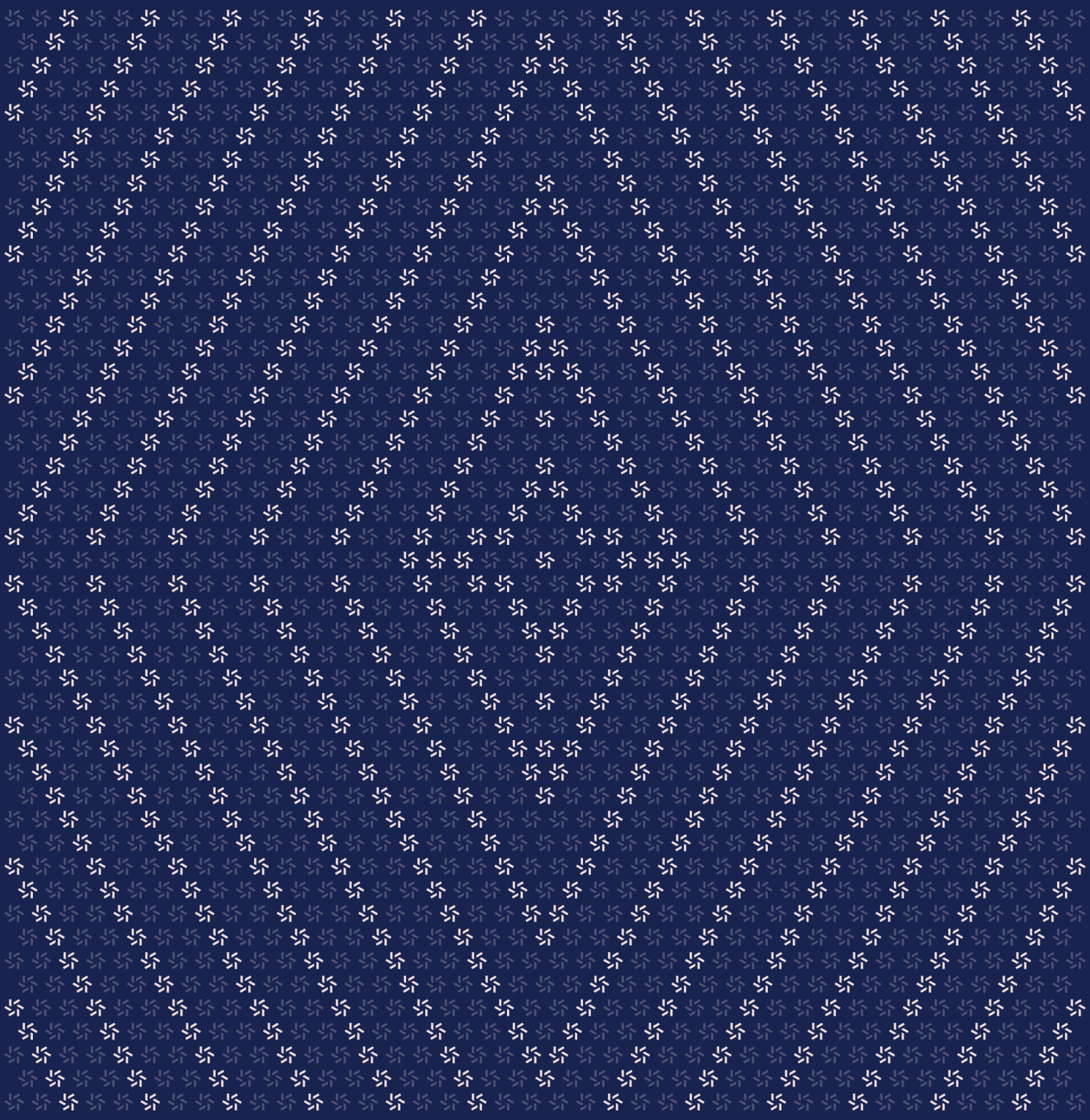
Prepared for
Cal Bera
Berachain

Prepared by
Syed Faraz Abrar
William Bowling
Seunghyeon Kim
Jisub Kim
Jaeeu Kim
Seungjun Kim
Zellic

February 22, 2024

Berachain BTS

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. Executive Summary

Zellic conducted a security assessment for Berachain from November 27th, 2023 to February 9th, 2024. During this engagement, Zellic reviewed Berachain BTS's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1. Goals of the Assessment

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

- Could an on-chain attacker drain from the vault?
- Could an on-chain attacker make the contract DOS?
- Could an attacker RCE on the off-chain services?
- Could an attacker SQLi on the off-chain services?

1.2. Non-goals and Limitations

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

- Price oracle
- 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.

During this assessment, incompleted price-oracle coding prevented us from auditing price-oracle–related vulnerabilities.

1.3. Results

During our assessment on the scoped Berachain BTS contracts, we discovered 30 findings. Eight critical issues were found. Four were of high impact, seven were of medium impact, seven were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	8
High	4
Medium	7
Low	7
Informational	4

2. Introduction

2.1. About Berachain Berps (BTS)

Berachain Berps(perpetual futures contract trading), which known as Berachain Trading System, is a liquidity-efficient, robust, and easy-to-use decentralized leveraged trading platform. It allows for low trading fees, a wide range of leverages, and pairs of up to 100x.

Berps also revolves around the native ERC-20 stablecoin HONEY. It is the main and only token used to open any positions. To get HONEY, one can do so by getting it at the main Berachain Honey Swap.

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:

Berachain BTS Contracts

Repository	https://github.com/berachain/bts ↗
Version	bts: `c529869a07fc358133c27ddfaa1d680c1b38bb86`
Programs	• contracts/src/* • services/*
Types	Solidity, Go
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with six consultants for a total of twelve person-weeks. The assessment was conducted over the course of seven calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Syed Faraz Abrar
Engineer
faith@zellic.io

William Bowling
Engineer
vakzz@zellic.io

Seunghyeon Kim
Engineer
seungyhyeon@zellic.io

Jisub Kim
Engineer
jisub@zellic.io

Jaeeu Kim
Engineer
jaeeu@zellic.io

Seungjun Kim
Engineer
seungjun@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

On December 7th, 2023, Berachain advised us to pause the audit of the BTS codebase due to concerns about its audit readiness. Instead, they proposed shifting our focus to their lending code. We initiated the lending audit on December 11th, 2023. The plan was to assess it for two weeks and then revisit the BTS audit post-Christmas break. The BTS audit recommenced on January 9th, 2024.

November 27, 2023	Kick-off call
November 27, 2023	Start of primary review period
December 7, 2023	Berachain requested us to postpone the audit
January 9, 2024	Start of resuming review period
February 9, 2024	End of resuming review period

3. Detailed Findings

3.1. Drain all tokens using `distributeFees()`

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

Description

In PoLRewarder, the `distributeFees()` function is marked as external and can be invoked by anyone, potentially resulting in the unauthorized transfer of all HONEY tokens from the sender who has approved the PoLRewarder.

```
function distributeFees(address sender, uint256 amount) external {
    SafeERC20Upgradeable.safeTransferFrom(
        feeAsset, sender, distributionModule, amount
    );
}
```

TradingStorage, which contains all balance used for trading, approves `rewarder` with `type(uint256).max`. Therefore, all tokens in TradingStorage can be drained and distributed as rewards by calling `distributeFees()`.

```
contract TradingStorage is Initializable, ITradingStorage {
    function initialize(
        ERC20 _honey,
        address _gov,
        address _bot,
        address _pairStorage,
        address _vault,
        address _trading,
        address _callbacks
    ) external initializer {
        // [...]
        honey.approve(address(vault.rewarder()), type(uint256).max);
        // [...]
    }
```

Impact

A malicious user has the ability to invoke `distributeFees()` and drain the balance of a sender who has granted approval to PoLRewarder.

Recommendations

Add an `onlyOwner` modifier to validate the caller of `distributeFees()`. This modification ensures that only the designated owner has the authority to execute the function, mitigating the risk of unauthorized access and potential fund drainage by malicious users.

Remediation

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

- 316ee1d5 ↗
- 5a84382e ↗

3.2. Claim vault's yield could be done by anyone

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

Description

BGT rewards are claimed using `BToken.claimBGT()`. When a user invokes `BToken.claimBGT()`, a call is made to `PoLRewarder.harvestRewards()`, which has an `onlyOwner` modifier and can be called externally. The `harvestRewards()` function uses the `msg.sender`'s `accBGT` to determine how many rewards the user can claim or harvest.

```
function harvestRewards(uint256 amount, address recipient)
    external
    onlyOwner
{
    address sender = _msgSender();
    updateGlobalBGT();

    updateUserBGT(sender, 0, false);
    users[sender].accBGT -= amount;

    Cosmos.Coin[] memory rewards =
        rewardsModule.withdrawDepositorRewardsTo(vault, recipient, amount);
    require(rewards.length == 1, "expected only 1 coin from Rewards
    Module");
    require(
        rewards[0].amount == amount,
        "unexpected withdraw amount from Rewards Module"
    );

    updateGlobalBGT(); // update again to reset the available BGT amount
        // after withdrawal
}
```

However, the `msg.sender` in this case will always be the BToken vault since it is an external call through `claimBGT()`. And when `updateUserBGT`is called, `feeAsset.balanceOf(receiver)` will take into account all HONEY tokens deposited by all users as it is checking the balance of the vault itself. This allows an attacker to claim rewards for all tokens deposited into the vault.

```
contract BToken is
    function claimBGT(uint256 amount, address recipient) external {
        rewarder.harvestRewards(amount, recipient);
    }
```

Impact

A malicious user can call `claimBGT()` to claim rewards for all tokens deposited into the vault.

Recommendations

Make sure only the BToken vault can call `harvestRewards()` with the appropriate `claimer` address.

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit 5a84382e ↗.

3.3. Gain more BGT tokens due to using wrong balance of assets

Target	PoLRewarder		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In PoLRewarder, the `onTransfer()` function is called when vault tokens are transferred. The `onTransfer()` function calls `updateUserBGT()`, which updates a user's accrued and debt balance of BGT using `accBGT` and `debtBGT`. At this moment, a share is a vault token that is returned from depositing `feeAssets(HONEY)`. However, `userShares` is not the balance of vault tokens; they use `feeAsset(HONEY)` as shares.

In this case, a user can continuously increase their `accBGT` by transferring vault tokens and `feeAssets` with their other address. For example, the user can use two addresses to hold `feeAssets(HONEY)`. The user can get `accBGT` and `debtBGT` by transferring vault tokens when the user holds `feeAssets`. After that, the user can set the balance of `feeAssets(HONEY)` as zero by transferring the whole balance to the other address, then triggering transfer of the vault token; this results in `debtBGT` being zero. Only `accBGT` will be constantly increase without `debtBGT` by repeating this.

In addition, a user who has `feeAssets(HONEY)` can increase `accBGT` without any vault token share, using `vault.transfer(address, 0)` to trigger `onTransfer()`.

```
function onTransfer(address from, address to, uint256 shares)
    external
    onlyOwner
{
    updateGlobalBGT();
    updateUserBGT(from, shares, false);
    updateUserBGT(to, shares, true);
}
```

```
// updates the BGT accrued and debt for receiver
function updateUserBGT(address receiver, uint256 sharesDelta, bool isMint)
    private
{
    User storage user = users[receiver];
    uint256 userShares = feeAsset.balanceOf(receiver);
    if (userShares > 0) {
```

```
            // set aside the receiver's accrued BGT
            user.accBGT +=
                ((userShares * accBGTPerShare) / PRECISION) - user.debtBGT;
        }
        user.debtBGT = (
            (isMint ? userShares + sharesDelta : userShares - sharesDelta)
                * accBGTPerShare
        ) / PRECISION;
    }
```

Impact

An attacker could continuously increase their `accBGT`, transferring `vault` and `feeAssets`. This could cause an attacker to get more BGT tokens as reward when harvesting them.

Recommendations

After consulting with the client, we found out that the defect in the code was caused by incorrect use between `vault` and `feeAsset`. Correct misuse of tokens.

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit `135fdd94` ↗.

3.4. Unsafe cast with update target price may lead to fund loss

Target	Trading		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Traders can set their target price. If the price reaches the target price, a bot automatically executes this trade. Traders use `updateTp()` to set their new target-price value after opening a trade. However, there is no boundary check on the `newTp` value, so that `newTp` could be set to `type(uint256).max`.

If the target price is `type(uint256).max`, this value is casted to `int256` in `currentPercentProfit()`, which is used for calculating profit. This results in a very high profit percent.

```
function updateTp(uint256 pairIndex, uint256 index, uint256 newTp)
    external
    notContract
    notDone
{
    address sender = _msgSender();

    ITradingStorage.Trade memory t =
        storageT.openTrades(sender, pairIndex, index);
    require(t.leverage > 0, "NO_TRADE");

    storageT.updateTp(sender, pairIndex, index, newTp);
    storageT.callbacks().setTpLastUpdated(
        sender,
        pairIndex,
        index,
        ITradingCallbacks.TradeType.MARKET,
        block.number
    );
    ITradingStorage.TradeInfo memory i =
        storageT.openTradesInfo(sender, pairIndex, index);

    emit TpUpdated(sender, pairIndex, index, newTp, i.openTime);
}
```

And with a short position, closing in target price could bypass all conditions of `TradingCall-`

backs.

```
function executeNftCloseOrderCallback(
        AggregatorAnswer memory a,
        ITradingStorage.PendingNftOrder memory o
    ) external onlyTrading notDone {
        // [...]
        Values memory v;

        if (cancelReason == CancelReason.NONE) {
            // [...]
            v.price = pairsStored.guaranteedSlEnabled(t.pairIndex)
                ? o.orderType == ITradingStorage.LimitOrder.TP
                    ? t.tp
                    : o.orderType == ITradingStorage.LimitOrder.SL ? t.
                        sl : a.price
                : a.price; // [1]

            // [...]
            if (o.orderType == ITradingStorage.LimitOrder.LIQ) {
                // [...]
            } else {
                // NFT reward in DAI
                v.reward1 = (
                    (
                        o.orderType == ITradingStorage.LimitOrder.TP &&
                            t.tp > 0
                            && (t.buy ? a.price >= t.tp : a.price <= t.
                                tp) // [2]
                    )
                        || (
                            o.orderType == ITradingStorage.LimitOrder.SL &&
    t.sl > 0
                                && (t.buy ? a.price <= t.sl : a.price
    >= t.sl)
                        )
                )
                    ? (v.levPosDai
    * pairsStored.pairNftLimitOrderFeeP(t.pairIndex))
                        / 100 / PRECISION
                    : 0;
            }
            // [...]
```

```
            cancelReason =
                v.reward1 == 0 ? CancelReason.NOT_HIT : CancelReason.NONE;

            // If can be triggered
            if (cancelReason == CancelReason.NONE) {
                v.profitP = currentPercentProfit(
                    t.openPrice, v.price, t.buy, t.leverage
                ); // [3]

                uint256 daiSentToTrader = unregisterTrade(
                    t,
                    false,
                    v.profitP,
                    v.posDai,
                    i.openInterestDai,
                    o.orderType == ITradingStorage.LimitOrder.LIQ
                        ? v.reward1
                        : (v.levPosDai
    * pairsStored.pairCloseFeeP(t.pairIndex))
                            / 100 / PRECISION,
                    v.reward1
                );
            // [...]
```

The value of `v.price` is set to the target price when using `LimitOrder.TP` at [1]. And `v.reward1` will be set since `a.price(oracle price) <= t.tp(type(uint256).max)` will be always true at [2], then the trade will be executed.

Finally, `type(uint256).max` could make the cast overflow in `currentPercentProfit()`, which is called at [3] and will end up returning `maxPnlP(900%)`, which is 900% profit.

```
function currentPercentProfit(
    uint256 openPrice,
    uint256 currentPrice,
    bool buy,
    uint256 leverage
) private pure returns (int256 p) {
    int256 maxPnlP = int256(MAX_GAIN_P) * int256(PRECISION);

    p = (
        (
            buy
                ? int256(currentPrice) - int256(openPrice)
                : int256(openPrice) - int256(currentPrice)
        ) * 100 * int256(PRECISION) * int256(leverage)
    ) / int256(openPrice);
```

```
    p = p > maxPnlP ? maxPnlP : p;
}
```

Impact

An attacker can create a malicious trade such that they always make 900% returns instantly. They could use this to drain the protocol.

Recommendations

Add a boundary check to target prices that do not make the cast overflow/underflow, or make sure a new target price is within an appropriate range.

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit d2cba636 ↗.

3.5. Invalid `pairIndex` can be valid

Target	Trading		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In `pkg/schemas/data/open_trades.go`, the `OpenTrade` struct that the limitbot uses to track open trades stores `PairIndex` as a `uint64`. However, it is a `uint256` on the Solidity side of things, and there is nothing preventing a limit order with an invalid pair index from being submitted.

The `PairIndex` of `OpenTrade` is declared as `uint64`.

```
type OpenTrade struct {
    PairIndex uint64   `json:"pairIndex"`
    Index *big.Int `json:"index"`
    Trader string   `json:"trader"`
    Leverage *big.Int `json:"leverage"`
    Buy bool     `json:"buy"`
    PositionSize *big.Int `json:"positionSize"`
}
```

But `pairIndex` of `Trade`, which is used in the smart contract, is declared as `uint256`.

```
struct Trade {
    address trader;
    uint256 pairIndex;
    uint256 index; // don't need, will auto-fill
    // [...]
}
```

Impact

An invalid pair index can be truncated down to a valid pair index — for example, if `uint64.max + 1` will be truncated in limitbot, but it is not truncated in the smart contract. So that pair index will be `1` in limitbot and `18446744073709551617` in the smart contract. This can lead to inconsistency in the limitbot.

Recommendations

Include a check on `openTrade` to require the `pairIndex` on the given trade is valid.

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit `8c715652` ↗.

3.6. Unsafe cast with reversal order may lead to fund loss

Target	TradingCallbacks		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In Trading, there is `REVERSAL` order type, which is used by a trader who predicted a trend shift in the market. The conditional expression is a little different from the others.

A malicious trader can exploit the system by utilizing the `REVERSAL` type in a limit order. In a `REVERSAL` limit order, the trader can set an exceptionally large value for their `openPrice`. This, combined with a large take profit (TP) or stop loss (SL), has the capability to bypass all checks in the `executeNftOpenOrderCallback()` function. Consequently, this could lead to the `currentPercentProfit()` function returning the maximum profit.

```
function executeNftOpenOrderCallback(
    AggregatorAnswer memory a,
    ITradingStorage.PendingNftOrder memory n
) external onlyTrading notDone {
    // [...]

        a.price = priceAfterImpact; // [1]

        cancelReason = (
            t == Trading.OpenLimitOrderType.LEGACY
                ? (a.price < o.minPrice || a.price > o.maxPrice)
                : (
                    t == Trading.OpenLimitOrderType.REVERSAL
                        ? (o.buy ? a.price > o.maxPrice : a.price < o.
                            minPrice) // [2]
                        : (o.buy ? a.price < o.minPrice : a.price >
    o.maxPrice)
                )
        )
            ? CancelReason.NOT_HIT
            : (
                !withinExposureLimits(
                    o.pairIndex, o.buy, o.positionSize, o.leverage
                )
```

```
                    ? CancelReason.EXPOSURE_LIMITS
                    : priceImpactP * o.leverage >
    pairInfos.maxNegativePnlOnOpenP()
                        ? CancelReason.PRICE_IMPACT
                        : !withinMaxLeverage(o.pairIndex, o.leverage)
                            ? CancelReason.MAX_LEVERAGE
                            : CancelReason.NONE
            );

        if (cancelReason == CancelReason.NONE) {
            (
                ITradingStorage.Trade memory finalTrade,
                uint256 tokenPriceDai,
                uint256 openFee
            ) = registerTrade(
                ITradingStorage.Trade(
                    o.trader,
                    o.pairIndex,
                    0,
                    0,
                    o.positionSize,
                    t == Trading.OpenLimitOrderType.REVERSAL // [3]
                        ? o.maxPrice // o.minPrice = o.maxPrice in that case
                        : a.price,
                    o.buy,
                    o.leverage,
                    o.tp,
                    o.sl
                )
            );
```

The value of `a.price` is set at [1], which is tracked by `priceAfterImpact` obtained from the oracle price. In the case of a long position, `o.maxPrice(openPrice)`, which is larger than `a.price`, could bypass the condition at [2]. After bypassing, `registerTrade()` is called with `o.maxPrice(openPrice)` at [3].

A large `openPrice` coupled with substantial TP/SL values, which are casted to negative, has the potential to cause an overflow in the `currentPercentProfit()` function. This overflow results in the function returning `maxPnlP(900%)`.

```
function currentPercentProfit(
    uint256 openPrice,
    uint256 currentPrice,
    bool buy,
    uint256 leverage
) private pure returns (int256 p) {
```

```
        int256 maxPnlP = int256(MAX_GAIN_P) * int256(PRECISION);

        p = (
            (
                buy
                    ? int256(currentPrice) - int256(openPrice)
                    : int256(openPrice) - int256(currentPrice)
            ) * 100 * int256(PRECISION) * int256(leverage)
        ) / int256(openPrice);

        p = p > maxPnlP ? maxPnlP : p;
    }
```

Impact

An attacker can create a malicious trade such that they always make 900% returns instantly. They could use this to drain the protocol.

Recommendations

Consider adding a check to ensure that `openPrice` and the values for TP/SL do not result in a casting overflow.

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit 2052c7d4 ↗.

3.7. Wrong balance of assets is used when accounting rewards

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

Description

In PolRewarder, it currently considers the receiver address's `feeAsset` (i.e., HONEY) token balance. The intended behavior is to use the receiver's BToken (i.e., bHONEY) balance instead, as that determines how many HONEY tokens the receiver deposited into the vault.

For example, in `updateGlobalBGT()`,

```
// updates the BGT available to the vault contract
function updateGlobalBGT() private {
    (uint256 availableBGT, uint256 newAvailableBGT) = getAvailableBGT();
    globalAvailableBGT = availableBGT;
    uint256 supply = feeAsset.totalSupply();
    if (supply > 0) {
        accBGTPerShare += (newAvailableBGT * PRECISION) / supply;
    }
}
```

Impact

This leads to inaccuracies in reward accounting.

Recommendations

Use `vault.balanceOf()` and `vault.totalSupply()` instead of the `feeAsset.balanceOf()` and `feeAsset.totalSupply()`.

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit `135fdd94` ↗.

3.8. The value of `honeyLeftInStorage` must be wrong

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

Description

The unused variables `reward2` and `reward3` are incorrectly used in the calculation of `honeyLeftInStorage`.

```
function unregisterTrade(
    ITradingStorage.Trade memory trade,
    bool marketOrder,
    int256 percentProfit, // PRECISION
    uint256 currentHoneyPos, // 1e18
    uint256 openInterestHoney, // 1e18
    uint256 closingFeeHoney, // 1e18
    uint256 limitFeeHoney // 1e18 (= SSS reward if market order)
) private returns (uint256 honeySentToTrader) {
    // [...]

    // 4.1 If collateral in storage (opened after update)
    if (trade.positionSizeHoney > 0) {
        Values memory v; // [1]

        v.reward1 =
            marketOrder ? limitFeeHoney + closingFeeHoney : closingFeeHo
                ney; // [2]
        transferFromStorageToAddress(address(this), v.reward1);
        vault.distributeReward(v.reward1);
        emit HoneyVaultFeeCharged(
            trade.trader, trade.pairIndex, trade.index, v.reward1
        );

        uint256 honeyLeftInStorage = currentHoneyPos - v.reward3 - v.
            reward2; // [3]

        if (honeySentToTrader > honeyLeftInStorage) {
            vault.sendAssets(
```

```
                honeySentToTrader - honeyLeftInStorage, trade.trader
            );
            transferFromStorageToAddress(trade.trader, honeyLeftInStorage);
        } else {
            uint256 amountHoney = honeyLeftInStorage - honeySentToTrader;
            transferFromStorageToAddress(address(this), amountHoney);
            vault.receiveAssets(amountHoney, trade.trader);
            transferFromStorageToAddress(trade.trader, honeySentToTrader);
        }

        // 4.2 If collateral in vault (opened before update)
    } else {
        vault.sendAssets(honeySentToTrader, trade.trader);
    }
}
```

The variable `v` is declared at [1], and `v.reward1` is calculated as the fee, which is subsequently transferred to the vault at [2]. However, `v.reward1` is not utilized in the calculation of `honeyLeftInStorage` at [3].

Furthermore, `v.reward2` and `v.reward3` are declared but not assigned any values, causing them to always be zero in this calculation.

Impact

If there is a mismatch in `honeyLeftInStorage`, it could lead to a revert for the last trader. This is because the balance is less than expected, resulting in an insufficient balance to return the entire amount to the trader.

Recommendations

Use the `v.reward1` in calculating `honeyLeftInStorage` instead of `reward2` and `reward3`.

```
-     uint256 honeyLeftInStorage = currentHoneyPos - v.reward3 - v.reward2;
+     uint256 honeyLeftInStorage = currentHoneyPos - v.reward1;
```

Remediation

This issue has been acknowledged by Berachain.

3.9. Trade-key collision in limitbot

Target	limitbot		
Category	Code Maturity	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The limitbot is responsible for executing the `LIMIT` order in the protocol. For example, limitbot detects the user's set price and executes open or close.

The `TradeKeyFor()` function is used by the limitbot to create a unique key for each trade. However, when packing all the variables for the final trade key, it has similar behavior to `abi.encodePacked()` in Solidity. In this case, certain values of `pairIndex` combined with `index` can cause key clashes. For example, `pairIndex = 0x11, index = 0x1` clashes with `pairIndex = 0x1, index = 0x11`.

Impact

There might be a collision between the `pairIndex` and `index` pair.

Recommendations

Consider padding all values to their full length or implementing an equivalent measure to prevent this.

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit `856924e1` ↗.

3.10. Function `distributePotentialReward` uses wrong modifier

Target	Referrals, TradingStorage		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

In Referrals, `distributePotentialReward()` is called from `registerTrade()`. The `distributePotentialReward()` function is used for transferring the referral fee to the referrer, if the referrer of the trader exists.

```
function registerTrade(ITradingStorage.Trade memory trade)
    private
    returns (ITradingStorage.Trade memory, uint256, uint256)
{
    // [...]

    // 1. Charge referral fee (if applicable) and send HONEY amount to vault
    if (referrals.getTraderReferrer(trade.trader) != address(0)) {
        // [...]

        v.reward1 = referrals.distributePotentialReward(
            trade.trader,
            v.levPosHoney,
            pairsStored.pairOpenFeeP(trade.pairIndex)
        );
        // [...]
```

In `Referrals.distributePotentialReward()`, when `referrerRewardValueHoney` is calculated, it calls `storageT.transferHoney()`. However, it does not meet the conditions of the `onlyTrading` modifier, reverting as a result.

```
function distributePotentialReward(
    address trader,
    uint256 volumeHoney,
    uint256 pairOpenFeeP
) external onlyCallbacks returns (uint256) {
    // [...]

    uint256 referrerRewardValueHoney = (
```

```
            volumeHoney * getReferrerFeeP(pairOpenFeeP, r.volumeReferredHoney)
        ) / PRECISION / 100;
        storageT.transferHoney(
            address(storageT), referrer, referrerRewardValueHoney
        );

        // [...]
    }
```

```
modifier onlyTrading() {
    require(isTradingContract[msg.sender]);
    _;
}

// [...]

function transferHoney(address _from, address _to, uint256 _amount)
    external
    onlyTrading
{
    if (_from == address(this)) {
        honey.transfer(_to, _amount);
    } else {
        honey.transferFrom(_from, _to, _amount);
    }
}
```

Impact

The function of `Referrals.distributePotentialReward()` does not work properly. In this case, every trade with a referrer reverts.

Recommendations

Change the modifier for `storageT.transferHoney()` or add `new role` for this.

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit `dbb113b9` ↗.

3.11. Stale oracle price could be used

Target	Trading		
Category	Protocol Risks	**Severity**	High
Likelihood	Low	**Impact**	High

Description

In `Trading.getPrice()`, the function uses `feedPrice1(answer)` from the oracle provider, but there are no checks for the timestamp in the return value of `latestRoundData()`. This absence of timestamp validation may pose a risk, as it does not ensure that the retrieved price data is recent or within an expected time frame.

Impact

The vulnerability allows for the potential retrieval of a stale oracle price, introducing the risk of inaccurate pricing and compromising the reliability of the trading system.

Recommendations

To mitigate the risk of obtaining outdated prices, incorporate a check on the timestamp of when the round was updated, using the `updatedAt` parameter from the Chainlink oracle's `latestRoundData()` ↗ response. This measure ensures that only recent and relevant price data is considered.

For example,

```
(, int256 feedPrice1,,uint ts,) =
            AggregatorV3Interface(f.feed1).latestRoundData();
+    require(block.timestamp - ts <= 10s, "ORACLE_HEALTHY")
```

Remediation

This issue has been acknowledged by Berachain.

3.12. Referral cannot be registered

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

Description

The logic for setting `r.active` to true, which registers a referrer, is currently absent. This absence may impact the expected behavior related to referrer registration.

```
function registerPotentialReferrer(address referrer) external {
        ReferrerDetails storage r = _referrerDetails[referrer];
        if (
            referrerByTrader[msg.sender] != address(0) || referrer ==
    address(0)
                || msg.sender == referrer
                || !r.active
        ) {
            return;
        }

        referrerByTrader[msg.sender] = referrer;
        r.tradersReferred.push(msg.sender);

        emit ReferrerRegistered(msg.sender, referrer);
    }
```

Impact

Users cannot register a `referrer`.

Recommendations

Remove the `!r.active` check or set `r.active = true` after the other checks.

Remediation

This issue has been acknowledged by Berachain.

3.13. Anyone can forge any events

Target	service/indexer		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `HoneyVaultWatcher` watches transactions and stores transactions' event data to the database. In `processBlock()`, `HoneyVaultWatcher` appends transaction receipts and calls `processReceipts()` with it. The `processReceipts()` function check emits an event and adds data to the database.

But, `HoneyVaultWatcher` does not check that the event came from the `vault`, so anyone can forge any of the events.

```
func (w *HoneyVaultWatcher) processBlock(sCtx *sdk.Context, height uint64)
    error {
    // [...]
    if len(block.Transactions()) > 0 {
        // get receipts for relevant txns
        receipts := make([]*coretypes.Receipt, 0)
        for _, tx := range block.Transactions() {
            receipt, err := sCtx.Chain().TransactionReceipt(sCtx,
    tx.Hash())
            if err != nil {
                sCtx.Logger().Error(
                    "Failed to retrieve transaction receipt",
                    "hash", tx.Hash().Hex(), "err", err,
                )
                return err
            }

            // failed txns not included
            if receipt.Status == 1 {
                receipts = append(receipts, receipt)
            }
        }
        // [...]

    if err = w.processReceipts(sCtx, receipts, block.Time(),
```

```
            currEpoch, tx); err != nil {
                sCtx.Logger().Error("processing receipts", "err", err)
                return err
            }
            // [...]
}
```

```
func (w *HoneyVaultWatcher) processReceipts(
    sCtx *sdk.Context, receipts []*coretypes.Receipt, timestamp uint64,
    currEpoch uint64, tx db.Tx,
) error {
    for _, receipt := range receipts {
        for _, log := range receipt.Logs {
            switch log.Topics[0].Hex() {
            case utils.WithdrawRequested:
                withdrawRequested,
    err := w.bTokenContract.ParseWithdrawRequested(*log)
                if err != nil {
                    sCtx.Logger().Error("failed to parse WithdrawRequested
    event", "err", err)
                    return err
                }

                if err = w.db.InsertHoneyWithdrawal(
                    sCtx, timestamp, request,
                    withdrawRequested.Sender, withdrawRequested.Owner,
                    withdrawRequested.Shares,
                    withdrawRequested.CurrEpoch,
    withdrawRequested.UnlockEpoch,
                    tx,
                ); err != nil {
                    sCtx.Logger().Error("failed to insert honey withdrawal
    request", "err", err)
                    return err
                }
            // [...]
        }
    }

    sCtx.Logger().Info("successfully processed withdrawals events")
    return nil
}
```

Impact

An attacker can forge the events from other smart contracts and mess up the indexer, which would likely lead to incorrect information. For example, if there is another system to use `WithdrawRequested` from the indexer's data to calculate pending withdrawal, forged events with huge withdrawal could poison the system.

Recommendations

All indexing jobs should be filtered by the contract address before being fed into processing. Add checking contract address of events. This ensures the indexer is not poisoned by a forged event.

For example, as below.

```
// [...]
if receipt.Status == 1 {
    for _, log := range receipt.Logs {
        if (log.Address == honeyVaultAddress) {
            receipts = append(receipts, log)
        }
    }
}
// [...]
```

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit 04dd976b ↗.

3.14. The value of `simplifiedTradeId` could be wrong when `delegatecall`

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

Description

In the `Trading.openTrade()` function, at the end when a limit order is created, it sets `simplifiedTradeId.trader = msg.sender`. This does not work correctly in the case where a delegatee is performing this call to `openTrade()` for a delegator.

```
function openTrade(
    ITradingStorage.Trade memory t,
    OpenLimitOrderType orderType,
    uint256 slippageP
) external notContract notDone {
    require(!isPaused, "PAUSED");
    require(t.openPrice * slippageP < type(uint256).max, "OVERFLOW");

    IPairsStorage pairsStorage = storageT.pairsStorage();

    address sender = _msgSender();
    // [...]
    ITradingCallbacks.SimplifiedTradeId memory simplifiedTradeId;
    simplifiedTradeId.trader = msg.sender;
```

In that case, `msg.sender` would end up being the delegatee's address, whereas the HONEY tokens are originally transferred from the sender address, which is set to `_msgSender()`, which in this case would be set to `senderOverride == original delegator's address`.

Impact

A delegatee can use their delegator's funds to open a trade for themselves.

Recommendations

Change the line of code to `simplifiedTradeId.trader = sender` instead to prevent delegatees from being able to open trades using their delegator's tokens.

```
simplifiedTradeId.trader = msg.sender;
simplifiedTradeId.trader = sender;
```

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit d4720a90 ↗.

3.15. Bypass closing fee

Target	TradingCallbacks		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In TradingCallbacks, `canExecuteTimeout` is used for delay time-out for updating each limit order. For example, if `canExecuteTimeout` is three, a user must wait three blocks until executing the next update.

When `canExecuteTimeout` is set to zero, traders could bypass their closing fee. The trader could trigger update of TP/SL to `a.price`, which is the oracle price, and execute TP/SL without any time-out delay.

```
function setCanExecuteTimeout(uint256 _canExecuteTimeout)
    external
    onlyGov
{
    if (_canExecuteTimeout > MAX_EXECUTE_TIMEOUT) {
        revert WrongParams();
    }
    canExecuteTimeout = _canExecuteTimeout;
    emit CanExecuteTimeoutUpdated(_canExecuteTimeout);
}
```

In Trading, `executeLimitOrder()` calls `canExecute()` to check time-out passed. And internally, they check `currentBlock` is bigger than time-out, which is `last updated(tp/sl/open) block number + canExecuteTimeout`.

```
function executeLimitOrder(
    ITradingStorage.LimitOrder orderType,
    address trader,
    uint256 pairIndex,
    uint256 index
) external notDone {
    require(
        canExecute(
            orderType,
            ITradingCallbacks.SimplifiedTradeId(
```

```
                trader,
                pairIndex,
                index,
                orderType == ITradingStorage.LimitOrder.OPEN
                    ? ITradingCallbacks.TradeType.LIMIT
                    : ITradingCallbacks.TradeType.MARKET
            )
        ),
        "IN_TIMEOUT"
    );
    // [...]
```

```
function canExecute(
    ITradingStorage.LimitOrder orderType,
    ITradingCallbacks.SimplifiedTradeId memory id
) private view returns (bool) {
    if (orderType == ITradingStorage.LimitOrder.LIQ) return true;

    uint256 b = block.number;
    address cb = storageT.callbacks();

    if (orderType == ITradingStorage.LimitOrder.TP) {
        return !cb.isTpInTimeout(id, b);
    }
    if (orderType == ITradingStorage.LimitOrder.SL) {
        return !cb.isSlInTimeout(id, b);
    }

    return !cb.isLimitInTimeout(id, b);
}
```

```
function isLimitInTimeout(
    address _callbacks,
    ITradingCallbacks.SimplifiedTradeId memory id,
    uint256 currentBlock
) external view returns (bool) {
    (ITradingCallbacks callbacks, ITradingCallbacks.LastUpdated memory l,) =
    _getTradeLastUpdated(
        _callbacks, id.trader, id.pairIndex, id.index, id.tradeType
    );

    return currentBlock < l.limit + callbacks.canExecuteTimeout();
}
```

Impact

The trader has the option to execute immediate trades using take profit or stop loss. This creates a similar effect to market trading but without incurring the market closing fee.

Traders may prefer limit orders over market orders because the closing fee for market orders is higher than that for limit orders. As a result, they strategically choose the order type as a limit to optimize their trading fees.

Recommendations

Consider implementing a check to ensure that the new take profit is greater than `a.price` and the new stop loss is smaller than `a.price`.

For example, like this.

```
function updateSlCallback(AggregatorAnswer memory a, PendingSl memory o)
    external
    onlyTrading
    notDone
{
    // [...]

    if (cancelReason == CancelReason.NONE) {
        // [...]

        cancelReason = a.price == 0
            ? CancelReason.MARKET_CLOSED
            : (
                (t.buy != o.buy || t.openPrice != o.openPrice)
                    ? CancelReason.WRONG_TRADE
                : (t.buy ? o.newSl > a.price : o.newSl < a.price)
                : (t.buy ? o.newSl >= a.price : o.newSl <= a.price)
                        ? CancelReason.SL_REACHED
                        : CancelReason.NONE
            );
```

Remediation

This issue has been acknowledged by Berachain.

3.16. Absent mapping check

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

Description

Functions using pair index as parameters do not check valid pair index. Calling `guaranteedSlEnabled()` with an incorrect `_pairIndex` will always return `true`. This is because the type of pairs is `mapping(uint256 => Pair)`, and accessing `pairs[_pairIndex]` returns the default `Pair` when `_pairIndex` is not mapped.

The default `Pair` contains default values of each members, and in this case, the `groupIndex` is of type `uint256` with a default value of zero. Therefore, `pairs[_pairIndex].groupIndex` will return zero.

```
function guaranteedSlEnabled(uint256 _pairIndex)
    external
    view
    returns (bool)
{
    return pairs[_pairIndex].groupIndex == 0; // crypto only
}
```

```
struct Pair {
    string from;
    string to;
    Feed feed;
    uint256 spreadP; // PRECISION
    uint256 groupIndex;
    uint256 feeIndex;
}
```

Similarly, calling `pairCloseFeeP()` with an incorrect `_pairIndex` will always return zero indexed fees' `closeFeeP`.

```
function pairCloseFeeP(uint256 _pairIndex)
    external
    view
```

```
    returns (uint256)
{
    return fees[pairs[_pairIndex].feeIndex].closeFeeP;
}
```

Impact

An attacker can potentially deceive the trading system by utilizing getter functions with an incorrect pair index. This manipulation could lead to confusion by returning data that is inaccurate.

Recommendations

Add a check to ensure that the index is mapped already using `isPairListed`.

Remediation

This issue has been acknowledged by Berachain.

3.17. Value of `totalDeposited` value can be wrong

Target	BToken, TradingCallbacks		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `BToken.initializeV2()` function lacks an access-control modifier, allowing anyone to call it. This absence of access control may lead to inconsistencies in the `totalDeposited` state variable.

```
function initializeV2() external reinitializer(2) {
    storeAccBlockWeightedMarketCap();
    totalDeposited += totalRewards;
}
```

Additionally, the `BToken.initializeV2()` function performs `totalDeposited += totalRewards`, but `totalRewards` is already accumulated in `BToken.distributeReward()`.

```
function distributeReward(uint256 assets) external {
    // [...]
    totalRewards += honeyAssets;
    totalDeposited += honeyAssets;

    emit FeesDistributed(honeyAssets, bgtAssets, assets, tvl());
}
```

This redundancy may lead to incorrect calculations or inconsistencies in the `totalDeposited` value.

Impact

Allowing anyone to call `initializeV2()` may lead to inconsistencies in the contract state.

Recommendations

Consider adding an appropriate access-control modifier to restrict the execution of `initializeV2()` to authorized entities. Alternatively, delete it if it is unnecessary.

Remediation

This issue has been acknowledged by Berachain.

3.18. Initialize functions are front-runnable

Target	BorrowingFees, BToken, Pairinfos, PairStorage, Referrals, TradingCallbacks, TradingStorage

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

Description

All `initialize()` functions are susceptible to front-running. The absence of an access-control modifier in these functions means that anyone can call them directly.

For example, in BorrowingFees,

```
function initialize(ITradingStorage _storageT, IPairInfos _pairInfos)
    external
    initializer
{
    require(
        address(_storageT) != address(0)
            && address(_pairInfos) != address(0),
        "WRONG_PARAMS"
    );

    storageT = _storageT;
    pairInfos = _pairInfos;
}
```

Impact

A malicious actor can front-run the initialization process.

Recommendations

Add the `onlyOwner` role to every `initialize()` function in all contracts to ensure that only authorized entities can invoke these functions.

Remediation

This issue has been acknowledged by Berachain.

3.19. Charging fees even upon cancel

Target	TradingStorage, TradingCallbacks		
Category	Protocol Risks	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

In cases where updating SL fails, the fee is still charged in `TradingStorage.handleDevGovFees()`. Following this, the contract emits `SlCanceled` after the `FeeDistributed` event.

```
| | |- emit FeesDistributed(assetsToHoney: 65800000000000 [6.58e13]
    , assetsToBGT: 28200000000000 [2.82e13], assetsTotal: 94000000000000
     [9.4e13], vaultTVL: 100000000000000048500000000000 [1e30])
| | |- ← ()
| |- ← 94000000000000 [9.4e13]
|- [2297] TradingStor-
    age::updateTrade((0x0000000000000000000000000000000011111, 0, 0,
    930600000000000000000000 [9.306e25], 9306000000000000 [9.306e15],
    100000000000000000000 [1e20], true, 100,
    101000000000000000000 [1.01e20], 99250000000000000000 [9.925e19]))
| |- ← ()
|- emit SlCanceled(trader: 0x0000000000000000000000000000000011111,
    pairIndex: 0, index: 0, cancelReason: 5)
```

Impact

This behavior may lead to confusion or unexpected outcomes as fees are charged even when the SL update fails.

Recommendations

Consider adding a check that will revert any TX that does not successfully update SL at the end of the `updateSlCallback`.

```
    require(
+           cancelReason == CancelReason.NONE,
+           TradeUtils.getCancelReasonMsg(cancelReason)
    );
```

Remediation

This issue has been acknowledged by Berachain.

3.20. The allowance of `WithdrawRequests` can be ignored

Target	BToken		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The functions `makeWithdrawRequest()` and `cancelWithdrawRequest()` both allow a spender to make and cancel requests on behalf of an owner. It checks to ensure that the allowance is greater than or equal to the amount of shares that are being requested to be withdrawn or canceled.

However, this check is ineffective because the spender can call the functions multiple times with whatever their allowance is to make or to cancel as many shares as they want for a withdraw request. Users only need a nonzero allowance to interact with these functions (and of course, not just one WEI of allowance but at least a minimum allowance that makes it feasible to call these functions multiple times).

Impact

A spender can call the functions multiple times with whatever their allowance is to withdraw or cancel and lock request of the owner.

Recommendations

Remove their allowance or restrict withdrawal requests to only come from the owner.

Remediation

Berachain would like to eliminate issues due to withdrawing spending for an owner, so they restricted withdrawal requests to only come from the owner. In the future, they would like to enable making requests via approval.

This issue has been acknowledged by Berachain, and a fix was implemented in commit 25a83e24 ↗.

3.21. User can do self-referrals

Target	Referrals		
Category	Business Logic	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

Traders can set a referral for their account. If they set a referral, `referrer` could gain a referral reward when trading is registered.

The `registerPotentialReferrals` does not check whether the `referrer` is `msg.sender`.

```
function registerPotentialReferrer(address referrer) external {
    ReferrerDetails storage r = _referrerDetails[referrer];
    if (
       referrerByTrader[msg.sender] != address(0) || referrer ==
           address(0)
       || !r.active
    ) {
        return;
    }
    }
```

Impact

Users can set themselves to `referrer` and gain the reward referral fee by just opening a trade.

Recommendations

Check if `msg.sender` is different from `referrer`.

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit `dbb113b9` ↗.

3.22. Nil pointer dereference on API server

Target	services/api		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

There is a nil pointer dereference at `HandleWSPriceFeedForSymbol()` in `/api/websocket/manager.go`. In case of an exception, an error is logged using `wp.Logger.Error()`. However, the `wp` does not have a logger, and this leads to application panic.

Impact

The application would panic.

Recommendations

Set an appropriate `Logger` upon initialization.

Remediation

This issue has been acknowledged by Berachain.

3.23. Trader contract can bypass max trades per pair

Target	Trading		
Category	Protocol Risks	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

There is a limit on the number of trades a trader can have open.

```
require(
    storageT.openTradesCount(sender, t.pairIndex)
        + storageT.openLimitOrdersCount(sender, t.pairIndex)
        < storageT.maxTradesPerPair(),
    "MAX_TRADES_PER_PAIR"
);
```

However, this limit can be bypassed by operating from multiple trading accounts or by using a contract that splits requested trades across multiple deployed proxies.

Impact

This limit can be bypassed for sophisticated traders.

Recommendations

We recommend removing this limit to equalize the playing field between traders using the front-end and sophisticated traders who deploy contracts to instantiate trades.

Remediation

This issue has been acknowledged by Berachain.

3.24. Unnecessary overflow check

Target	Trading		
Category	Protocol Risks	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The overflow check `require(t.openPrice * slippageP < type(uint256).max, “OVERFLOW”)` in the `openTrade()` function is redundant for Solidity versions 0.8.0 and above, as these versions automatically catch overflow.

```
function openTrade(
    ITradingStorage.Trade memory t,
    ITradingCallbacks.TradeType orderType,
    uint256 slippageP // for market orders only
) external notContract notDone {
    require(!isPaused, "PAUSED");
    require(t.openPrice * slippageP < type(uint256).max, "OVERFLOW");
```

Impact

It may compromise code readability and could consume more gas fees.

Recommendations

Consider removing this unnecessary code to streamline the contract, save gas, and align with the improved overflow handling introduced in Solidity 0.8.0.

Remediation

This issue has been acknowledged by Berachain.

3.25. The `pairMaxLeverage` could be set to a huge value

Target	PairStorage, TradingCallbacks		
Category	Protocol Risks	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

There is a potential risk that `pairStorage.pairMaxLeverage()` could be set to an excessively large value by the manager. While `maxLeverage` is initially initialized in PairStorage within the range of `maxLeverage <= 1000 (MAX_LEVERAGE)`, it can be overridden in the TradingCallbacks contract using `TradingCallbacks.setPairMaxLeverage()` without any restrictions.

Impact

The unrestricted ability to modify `maxLeverage` above its intended value of `MAX_LEVERAGE` poses a potential risk and should be addressed to prevent unintended consequences.

Recommendations

We recommend restricting the `maxLeverage` value in `TradingCallbacks.setPairMaxLeverage()` as well.

```
+   require(maxLeverage <= getPairsStorage().MAX_LEVERAGE(),
    "LEVERAGE_TOO_HIGH");
```

Remediation

This issue has been acknowledged by Berachain.

3.26. No modifier

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

Description

The `PairStorage.addPairs()` function is external, allowing anyone to call it. However, it internally calls `addPair()` with the `onlyGov` modifier, which restricts access to only those with the `Gov` role.

Impact

Calling `addPairs()` without the necessary role will result in a revert.

Recommendations

Consider adding the modifier `onlyGov` to `addPairs()`.

Remediation

This issue has been acknowledged by Berachain.

3.27. Transferring whole balance of vault tokens will brick their transfers

Target	PoLRewarder		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	High	**Impact**	Informational

Description

If a user initiates a transfer of vault tokens to themselves — for example, if `user1` calls `vault.transfer(user1, vault.balanceOf(user1))` — the accounting becomes unusable due to `debtBGT` being higher than the user's current `userShares`.

This results in a situation where subsequent operations such as `withdraw`, `transfer`, and others trigger `updateUserBGT()`, leading to a revert due to the now problematic accounting state.

```
// updates the BGT accrued and debt for receiver
function updateUserBGT(address receiver, uint256 sharesDelta, bool isMint)
    private
{
    User storage user = users[receiver];
    uint256 userShares = feeAsset.balanceOf(receiver);
    if (userShares > 0) {
        // set aside the receiver's accrued BGT
        user.accBGT +=
            ((userShares * accBGTPerShare) / PRECISION) - user.debtBGT;
    }
    user.debtBGT = (
        (isMint ? userShares + sharesDelta : userShares - sharesDelta)
            * accBGTPerShare
    ) / PRECISION;
}
```

Impact

If a user transfers their entire vault token balance to themselves, token vault transfers will be disrupted until either the `accBGTPerShare` increases or the user transfers out their `feeAsset` tokens and subsequently executes a vault transfer.

Recommendations

Prevent self-transfers between users and turn them into a no-op.

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit 10918ce2 ↗.

3.28. Function `pendingBGT` does not have return value

Target	BToken		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	High	**Impact**	Informational

Description

In BToken, the `pendingBGT` function returns the current amount of a user's BGT reward, which are received when depositing HONEY tokens. The `BToken.pendingBGT()` function only calls `rewarder.accRewards(owner)` but does not return it, so the `pendingBGT` view function does not work properly.

```
// gets the accrued BGT rewards to owner up until this point in
// time
function pendingBGT(address owner) external view returns (uint256) {
    rewarder.accRewards(owner);
}
```

```
function accRewards(address owner) external view returns (uint256) {
    // calculate the new BGT accrued to owner since the time of last update
    (, uint256 newAvailableBGT) = getAvailableBGT();
    uint256 newAccBGTPerShare = accBGTPerShare;
    uint256 supply = vault.totalSupply();
    if (supply > 0) {
        newAccBGTPerShare += ((newAvailableBGT * PRECISION) / supply);
    }

    User memory user = users[owner];
    return user.accBGT
        + ((vault.balanceOf(owner) * newAccBGTPerShare) / PRECISION)
        - user.debtBGT;
}
```

Impact

The view function `BToken.pendingBGT()` does not return `rewarder.accRewards(owner)`.

Recommendations

Ensure that `BToken.pendingBGT()` returns `rewarder.accRewards()` appropriately.

```
function pendingBGT(address owner) external view returns (uint256) {
    rewarder.accRewards(owner);
    return rewarder.accRewards(owner);
}
```

Remediation

This issue has been acknowledged by Berachain, and a fix was implemented in commit c0e25773 ↗.

3.29. Unused variable

Target	Trading		
Category	Code Maturity	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The Trading contract includes the variable `marketOrdersTimeout` and sets it in the `constructor()`. Additionally, there is a function, `setMarketOrdersTimeout()`, to set this variable, even though it is not utilized elsewhere in the contract.

Impact

It may compromise code readability and could consume more gas fees.

Recommendations

It is recommended to either remove the unused function or integrate the variable into the contract logic to ensure consistency and avoid potential confusion.

Remediation

This issue has been acknowledged by Berachain.

3.30. Unused module

Target	BToken		
Category	Code Maturity	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The BToken contract imports `ERC721MinterBurner`, but there is no utilization of `ERC721MinterBurner` within the contract.

Impact

It may compromise code readability and could consume more gas fees.

Recommendations

If it is not being used, consider removing the import statement to enhance code clarity and reduce unnecessary dependencies.

Remediation

This issue has been acknowledged by Berachain.

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. Potential SQL injection

For the `rpc` API, certain sections of SQL involve direct insertion of variables into the SQL statement. Upon investigating for SQL injection vulnerabilities, it was observed that the variables used in the SQL statement are converted into fixed values within a switch statement.

```
func constructCandleQueryAndArgs(pairIndex uint, resolution string, from,
    to, countback uint64) (string, []any) {
    ts := "ts"
    table := "candle_1m"

    if resolution != "1m" {
        ts = ts + "_" + resolution
        table = "candle_" + resolution // [1]
    }

    // LAG ensures the open price is the close price of the previous candle
    query := `
    SELECT
    ` + ts + ` as timestamp,
        COALESCE(LAG(close(candlestick)) OVER (PARTITION BY pair_index ORDER
    BY ` + ts + `), open(candlestick)) as open,
        high(candlestick),
        low(candlestick),
        close(candlestick)
    FROM ` + table + ` // [2]
    WHERE `
```

```
func NormalizeTradingViewResolution(tvFormat string) (ResolutionString,
    error) {
    switch tvFormat {
    case "1":
        return OneMinute, nil
    case "3":
        return ThreeMinutes, nil
    case "5":
        return FiveMinutes, nil
    case "15":
        return FifteenMinutes, nil
```

```
    case "60":
        return OneHour, nil
    case "240":
        return FourHours, nil
    case "1D":
        return OneDay, nil
    case "1W":
        return OneWeek, nil
    case "1M":
        return OneMonth, nil
    default:
        return "", errors.New("unsupported TradingView resolution format")
    }
}
```

This practice eliminates the ability to control the variables directly in a way that could lead to SQL injection. While the Berachain team assured that the code is secure based on these considerations, a precautionary recommendation is made to use prepared statements as an additional measure to prevent potential SQL injection issues in the future.

4.2. Setting maximum and minimum of `_maxNegativePnlOnOpenP`

The `_maxNegativePnlOnOpenP` parameter in the PairInfos contract enables a trader to initiate trading with initial losses. However, without proper bounds checks, if this value becomes excessively large or small, it may compromise the safety of trading. It is recommended to implement lower and upper bound checks for the `maxNegativePnlOnOpenP` parameter wherever it is set, ensuring that the value remains within a reasonable range.

4.3. Extreme TP/SL values may lead to an unintended EVM revert

Excessively large TP/SL values can lead to an overflow issue in the `currentPercentProfit()` function due to multiplication, resulting in the unexpected reverting of the `openTrade` function. For more complex protocols, it is advisable to implement checks to restrict TP and SL from having excessively large values.

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: BToken.sol

Function: `receiveAssets(uint256 assets, address user)`

This function is to receive assets for the user.

Inputs

- `assets`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Asset of mint.
- `user`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of `msg.sender` (just using for emit).

Branches and code coverage

Intended branches

- `assets` must be a previously calculated value.
  - ☐ Test coverage

Negative behavior

- Revert if sender is not the same with `pnlHandler`.
  - ☐ Negative test
- Revert if assets are larger than sender's amount.
  - ☐ Negative test

Function call analysis

- `SafeERC20Upgradeable.safeTransferFrom(this._assetIERC20(), sender, address(this), assets)`
  - **What is controllable?** `sender` and `assets`.

- **If the return value is controllable, how is it used and how can it go wrong?** None.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this.newEpochRequest()`
  `-> this.startNewEpoch()`
  `-> this.updateShareToAssetsPrice()`
  `-> this.storeAccBlockWeightedMarketCap()`
  `-> this.getPendingAccBlockWeightedMarketCap(block.number)`
  `-> MathUpgradeable.max(this.marketCap(), 1)`
  - **What is controllable?** None.
    - **If the return value is controllable, how is it used and how can it go wrong?** None.
    - **What happens if it reverts, reenters or does other unusual control flow?** It does not have any filters at receiver, so sender could be zero.

Function: `sendAssets(uint256 assets, address receiver)`

This function is to send assets to the receiver.

Inputs

- `assets`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Asset of mint.
- `receiver`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of receiver.

Branches and code coverage

Intended branches

- `_msgSender()` sends calculated assets value to receiver.
  - ☐ Test coverage

Negative behavior

- Revert if `sender` is not `pnlHandler`.
  - ☐ Negative test
- Revert if `accPnlPerToken` is larger than `int256(maxAccPnlPerToken())`.

- ☐ Negative test
- Revert if `dailyAccPnlDelta` is larger than `int256(maxDailyAccPnlDelta)`.
  - ☐ Negative test
- Revert if `assets` is larger than sender's amount.
  - ☐ Negative test

Function call analysis

- `this.newEpochRequest()`
  `-> this.startNewEpoch()`
  `-> this.updateShareToAssetsPrice()`
  `-> this.storeAccBlockWeightedMarketCap()`
  `-> this.getPendingAccBlockWeightedMarketCap(block.number)`
  `-> MathUpgradeable.max(this.marketCap(), 1)`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `SafeERC20Upgradeable.safeTransfer(this._assetIERC20(), receiver, assets)`
  - **What is controllable?** `receiver`, `assets`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** It does not have any filters at receiver. So, It could be set zero.

Function: `transferOwnership(address newOwner)`

This function is to transfer the `owner` role to the `newOwner` address.

Inputs

- `newOwner`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be zero.
  - **Impact**: Address to set new owner.

Branches and code coverage

Intended branches

- Transfer the `owner` to `newOwner` address.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not an owner.
  - ☐ Negative test
- Revert if `newOwner` is zero.
  - ☐ Negative test

Function: `updateLossesBurnP(uint256 newValue)`

This function is to update `LossesBurnP` to `newValue`.

Inputs

- `newValue`
  - **Control**: Arbitrary.
  - **Constraints**: Smaller than 25% or equal.
  - **Impact**: The value of `lossesBurnP`.

Branches and code coverage

Intended branches

- Set `lossesBurnP` to `newValue` and emit `NumberParamUpdated` event.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not `manager`.
  - ☐ Negative test
- Revert if `newValue` is larger than 25%.
  - ☐ Negative test

Function: `updateManager(address newValue)`

This function is to transfer the manager role to `newValue` address.

Inputs

- `newValue`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be zero.

- **Impact**: Address for manager.

Branches and code coverage

Intended branches

- Transfer `Manger` to `newValue` address and emit `AddressParamUpdated` event.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not an owner.
  - ☐ Negative test
- Revert if `newValue` is zero.
  - ☐ Negative test

Function: `updateMaxDailyAccPnlDelta(uint256 newValue)`

This function is to update `MaxDailyAccPnlDelta`.

Inputs

- `newValue`
  - **Control**: Arbitrary.
  - **Constraints**: Larger than `MIN_DAILY_ACC_PNL_DELTA`.
  - **Impact**: Max value of `DailyAccPnlDelta`.

Branches and code coverage

Intended branches

- Set `PnlHandler` to `newValue` value and emit `maxDailyAccPnlDelta` event.
  - ☐ Test coverage

Negative behavior

- Revert if `newValue` is smaller than `MIN_DAILY_ACC_PNL_DELTA`.
  - ☐ Negative test
- Revert if the caller is not the manager.
  - ☐ Negative test

Function: `updateMaxSupplyIncreaseDailyP(uint256 newValue)`

This function is to update `MaxSupplyIncreaseDailyP` to `newValue`.

Inputs

- `newValue`
  - **Control**: Arbitrary.
  - **Constraints**: Smaller than `MAX_SUPPLY_INCREASE_DAILY_P` or equal.
  - **Impact**: Max value of `SupplyIncreaseDailyP`.

Branches and code coverage

Intended branches

- Set `maxSupplyIncreaseDailyP` to `newValue` and emit `NumberParamUpdated` event.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not the manager.
  - ☐ Negative test
- Revert if newValue is larger than `MAX_SUPPLY_INCREASE_DAILY_P`.
  - ☐ Negative test

Function: `updatePnlHandler(address newValue)`

This function is to transfer `PnlHandler` to `newValue` value.

Inputs

- `newValue`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be zero.
  - **Impact**: Address of `PnlHandler`.

Branches and code coverage

Intended branches

- Set `PnlHandler` to `newValue` value and emit `AddressParamUpdated` event.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not an owner.
  - ☐ Negative test
- Revert if `newValue` is zero.
  - ☐ Negative test

Function: `updateWithdrawLockThresholdsP(uint256[Literal(value=2, unit=None)] newValue)`

This updates `WithdrawLockThresholdsP`.

Inputs

- `newValue`
  - **Control**: Arbitrary.
  - **Constraints**: Value of index 1 larger than the value of index 0.
  - **Impact**: Array of `withdrawLockThresholdsP`.

Branches and code coverage

Intended branches

- Set `withdrawLockThresholdsP` to `newValue` and emit `WithdrawLockThresholdsPUpdated` event.
  - ☐ Test coverage

Negative behavior

- Revert if `newValue` is not array.
  - ☐ Negative test
- Revert if the caller is not the manager.
  - ☐ Negative test
- Revert if `newValue[1]` is smaller than `newValue[0]` or same.
  - ☐ Negative test

5.2. Module: BorrowingFees.sol

Function: `getTradeBorrowingFee(BorrowingFeeInput input)`

This function is to return the fee of the trade.

Inputs

- `input`
  - **Control**: Arbitrary
  - **Constraints**: None.
  - **Impact**: The value of the initial accrue fees.

Branches and code coverage

Intended branches

- The value of the `input` has `trader`, `pairIndex`, and `index` correct.
  - ☐ Test coverage

Negative behavior

- Revert if `initialFees.block` is smaller than 1.
  - ☐ Negative test

Function: `handleTradeAction(address trader, uint256 pairIndex, uint256 index, uint256 positionSizeHoney, bool open, bool long)`

This function is to handle trade process such as setting `pair`, `group` and `fees` of the trade.

Inputs

- `trader`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of trader.
- `pairIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of pair.
- `index`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of `Initialfee`.
- `positionSizeHoney`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Position size of Honey.
- `open`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Status of trade.
- `long`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Boolean value of the position (LONG, SHORT).

Branches and code coverage

Intended branches

- The value of `GroupIndex` is set that `initialAccFees` has same value as index.
  - ☐ Test coverage
- The value of `pairIndex` is set that `initialAccFees` has same value as index.
  - ☐ Test coverage
- The value of `Index` is set that `initialAccFees` has same value as index.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not a `callback`.
  - ☐ Negative test
- Revert if `positionSizeHoney` is larger than `type(uint112).max`.
  - ☐ Negative test

Function call analysis

- `this._setPairPendingAccFees(pairIndex, block.number)`
  `-> this.getPairPendingAccFees(pairIndex, currentBlock)`
  `-> this.getPairWeightedVaultMarketCapSinceLastUpdate(pairIndex, currentBlock)`
  `-> this.getPendingAccBlockWeightedMarketCap(currentBlock)`
  `-> BToken.getPendingAccBlockWeightedMarketCap(currentBlock)`
  - **What is controllable?** `pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns `marketCap` value in current block.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._setPairPendingAccFees(pairIndex, block.number)`
  `-> this.getPairPendingAccFees(pairIndex, currentBlock)`
  `-> this.getPairWeightedVaultMarketCapSinceLastUpdate(pairIndex, currentBlock)`
  `-> this.getPendingAccBlockWeightedMarketCap(currentBlock)`
  `-> this.storageT.vault()`
  - **What is controllable?** `pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns `marketCap` value in current block.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._setPairPendingAccFees(pairIndex, block.number)`
  `-> this.getPairPendingAccFees(pairIndex, currentBlock)`
  `-> this.getPairOpenInterestHoney(pairIndex)`
  `-> this.storageT.openInterestHoney(pairIndex, 0)`

- **What is controllable?** `pairIndex`.
- **If the return value is controllable, how is it used and how can it go wrong?** Returns long of pair.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `initialize(ITradingStorage _storageT, IPairInfos _pairInfos)`

This sets `storageT` and `_pairInfos` as an initializer.

Inputs

- `_storageT`
  - **Control**: None.
  - **Constraints**: Must not be zero address.
  - **Impact**: Address to be `storageT`.
- `_pairInfos`
  - **Control**: None.
  - **Constraints**: Must not be zero address.
  - **Impact**: Address to be `_pairInfos`.

Branches and code coverage

Intended branches

- Checks the address of `_storageT`, then sets `_pairInfos` to initialize a contract.
  - ☐ Test coverage

Negative behavior

- Revert if the `initialize()` function is called in multiple times.
  - ☐ Negative test
- Revert if the address of `_storageT` is zero address.
  - ☐ Negative test
- Revert if the value of `_pairInfos` is zero address.
  - ☐ Negative test

Function: `setPairParamsArray(uint256[] indices, PairParams[] values)`

This function is to set the parameters for the pair with array.

Inputs

- `indices`
  - **Control**: Arbitrary.
  - **Constraints**: It must be the same with `values.length`.
  - **Impact**: Array of `pairIndex`.
- `values`
  - **Control**: Arbitrary.
  - **Constraints**: It must be the same with `indices.length`.
  - **Impact**: Array of pair parameters.

Branches and code coverage

Intended branches

- Set the given multiple pairs' parameter.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not a `manager`.
  - ☐ Negative test
- Revert if the length of the `indices` is different with the length of the `values`.
  - ☐ Negative test

Function call analysis

- `this._setPairParams(indices[i], values[i])`
  `-> this._setPairPendingAccFees(pairIndex, block.number)`
  `-> this.getPairPendingAccFees(pairIndex, currentBlock)`
  `-> this.getPairWeightedVaultMarketCapSinceLastUpdate(pairIndex, currentBlock)`
  `-> this.getPendingAccBlockWeightedMarketCap(currentBlock)`
  `-> BToken.getPendingAccBlockWeightedMarketCap(currentBlock)`
  - **What is controllable?** `indices`, `values`, and `pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._setPairParams(indices[i], values[i])`
  `-> this._setPairPendingAccFees(pairIndex, block.number)`
  `-> this.getPairPendingAccFees(pairIndex, currentBlock)`
  `-> this.getPairWeightedVaultMarketCapSinceLastUpdate(pairIndex, currentBlock)`
  `-> this.getPendingAccBlockWeightedMarketCap(currentBlock)`

-> `this.storageT.vault()`
    - **What is controllable?** `indices`, `values`, and `pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** None.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._setPairParams(indices[i], values[i])`
  -> `this._setPairPendingAccFees(pairIndex, block.number)`
  -> `this.getPairPendingAccFees(pairIndex, currentBlock)`
  -> `this.getPairOpenInterestHoney(pairIndex)`
  -> `this.storageT.openInterestHoney(pairIndex, 0)`
    - **What is controllable?** `indices`, `values`, and `pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** None.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `setPairParams(uint256 pairIndex, PairParams value)`

This function is to set the parameter for the pair.

Inputs

- `pairIndex`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: The index of `pair` to select.
- `value`
    - **Control**: Arbitrary.
    - **Constraints**: None.
    - **Impact**: Pair parameter to apply.

Branches and code coverage

Intended branches

- Save the value for the `pairIndex`.
    - ☐ Test coverage
- Set `AccFees` with long, short, max.
    - ☐ Test coverage

Negative behavior

- Revert if the caller is not a `manager`.

- ☐ Negative test

Function call analysis

- `this._setPairParams(pairIndex, value)`
  `-> this._setPairPendingAccFees(pairIndex, block.number)`
  `-> this.getPairPendingAccFees(pairIndex, currentBlock)`
  `-> this.getPairWeightedVaultMarketCapSinceLastUpdate(pairIndex, currentBlock)`
  `-> this.getPendingAccBlockWeightedMarketCap(currentBlock)`
  `-> BToken.getPendingAccBlockWeightedMarketCap(currentBlock)`
  - **What is controllable?** `pairIndex` and `value`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._setPairParams(pairIndex, value)`
  `-> this._setPairPendingAccFees(pairIndex, block.number)`
  `-> this.getPairPendingAccFees(pairIndex, currentBlock)`
  `-> this.getPairWeightedVaultMarketCapSinceLastUpdate(pairIndex, currentBlock)`
  `-> this.getPendingAccBlockWeightedMarketCap(currentBlock)`
  `-> this.storageT.vault()`
  - **What is controllable?** `pairIndex` and `value`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
  - `this._setPairParams(pairIndex, value)`
    `-> this._setPairPendingAccFees(pairIndex, block.number)`
    `-> this.getPairPendingAccFees(pairIndex, currentBlock)`
    `-> this.getPairOpenInterestHoney(pairIndex)`
    `-> this.storageT.openInterestHoney(pairIndex, 0)`
    - **What is controllable?** `pairIndex` and `value`.
    - **If the return value is controllable, how is it used and how can it go wrong?** None.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `withinMaxGroupOi(uint256 pairIndex, bool long, uint256 positionSizeHoney)`

This is function to check position value.

Inputs

- `pairIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of pair.
- `long`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Boolean value of the position (LONG, SHORT).
- `positionSizeHoney`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Position size of Honey.

5.3. Module: Delegatable.sol

Function: `delegatedAction(address trader, bytes call_data)`

This function calls `call_data` as delegate.

Inputs

- `trader`
  - **Control**: Arbitrary.
  - **Constraints**: `delegations[trader]` must be same with `msg.sender`.
  - **Impact**: The address of delegations.
- `call_data`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The data called as delegate.

Function: `removeDelegate()`

This sets `delegations[msg.sender]` to 0.

Function: `setDelegate(address delegate)`

This sets `delegations[msg.sender]` to delegate.

Inputs

- `delegate`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the delegate.

Function: `_msgSender()`

This function returns `senderOverride` or `msg.sender` when `senderOverride` is `address(0)`.

5.4. Module: PairInfos.sol

Function: `getTradeFundingFee(address trader, uint256 pairIndex, uint256 index, bool long, uint256 collateral, uint256 leverage)`

This function calculates funding fee. If the funding fee is positive, the user must pay the fee. If not, the user will get the reward.

Inputs

- `trader`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the trader.
- `pairIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of the pair.
- `index`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of the trade.
- `long`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Boolean value of the position (LONG, SHORT).
- `collateral`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The collateral.
- `leverage`

- **Control**: Arbitrary.
- **Constraints**: None.
- **Impact**: The leverage.

Function: `initialize(ITradingStorage _storageT, address _manager, uint256 _maxNegativePnlOnOpenP)`

This sets `storageT`, `manager`, and `maxNegativePnlOnOpenP` as an initializer.

Inputs

- `_storageT`
  - **Control**: None.
  - **Constraints**: Must not be zero address.
  - **Impact**: Address to be `storageT`.
- `_manager`
  - **Control**: None.
  - **Constraints**: Must not be zero address.
  - **Impact**: Address to be `manager`.
- `_maxNegativePnlOnOpenP`
  - **Control**: None.
  - **Constraints**: Must be bigger than zero.
  - **Impact**: A value to be `maxNegativePnlOnOpenP`.

Branches and code coverage

Intended branches

- After checking `_storageT`, `_manager`, and `_maxNegativePnlOnOpenP`, the `storageT`, `manager`, and `maxNegativePnlOnOpenP` must be set.
  - ☐ Test coverage

Negative behavior

- The `initialize` function must called only once.
  - ☐ Negative test
- The `_storageT` must not be zero address.
  - ☐ Negative test
- The `_manager` must not be zero address.
  - ☐ Negative test

Function: `setFundingFeePerBlockPArray(uint256[] indices, uint256[] values)`

This sets the multiple `fundingFeePerBlockP`.

Inputs

- `indices`
  - **Control**: Arbitrary.
  - **Constraints**: Must be same length with `values`.
  - **Impact**: Array of `pairIndex`.
- `values`
  - **Control**: Arbitrary.
  - **Constraints**: Must be same length with `indices`.
  - **Impact**: Array of `fundingFeePerBlockP`.

Branches and code coverage

Intended branches

- Set the each `pairIndex`'s `fundingFeePerBlockP`.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not a manager.
  - ☐ Negative test
- Revert if the length of `indices` and `values` are not the same.
  - ☐ Negative test

Function call analysis

- `this.setFundingFeePerBlockP(indices[i], values[i])`
  `-> this.storeAccFundingFees(pairIndex)`
  `-> this.getPendingAccFundingFees(pairIndex)`
  `-> this.storageT.openInterestHoney(pairIndex, 0)`
  - **What is controllable?** `indices` and `values` for the `setFundingFeePerBlockP()` — `pairIndex` for the other functions.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.setFundingFeePerBlockP(indices[i], values[i])`
  `-> this.storeAccFundingFees(pairIndex)`

```
-> this.getPendingAccFundingFees(pairIndex)
-> this.storageT.openInterestHoney(pairIndex, 1)
```

- **What is controllable?** `indices` and `values` for the `setFundingFeePerBlockP()` — `pairIndex` for the other functions.
- **If the return value is controllable, how is it used and how can it go wrong?** None.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `setFundingFeePerBlockP(uint256 pairIndex, uint256 value)`

This function sets the funding fee for the pair.

Inputs

- `pairIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of the pair.
- `value`
  - **Control**: Arbitrary.
  - **Constraints**: Must be less than 10,000,000.
  - **Impact**: Value for the `fundingFeePerBlockP` of the pair.

Branches and code coverage

Intended branches

- The pair's `fundingFeePerBlockP` must be set as given `value`.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not a manager.
  - ☐ Negative test
- Revert if the value is bigger than 10,000,000.
  - ☐ Negative test

Function call analysis

- `this.storeAccFundingFees(pairIndex)`
  `-> this.getPendingAccFundingFees(pairIndex)`
  `-> this.storageT.openInterestHoney(pairIndex, 0)`

- **What is controllable?** `pairIndex`.
- **If the return value is controllable, how is it used and how can it go wrong?** None.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this.storeAccFundingFees(pairIndex)`
  `-> this.getPendingAccFundingFees(pairIndex)`
  `-> this.storageT.openInterestHoney(pairIndex, 1)`
  - **What is controllable?** `pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `setManager(address _manager)`

This function sets the manager address and can be called by `Gov` only.

Inputs

- `_manager`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: An address to be `manager`.

Branches and code coverage

Intended branches

- Set manager address and emits the `ManagerUpdated` event.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is non-`Gov`.
  - ☐ Negative test

Function: `setMaxNegativePnlOnOpenP(uint256 value)`

This function sets `maxNegativePnlOnOpenP`, which is a max negative PNL on opening the trade.

Inputs

- `value`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Value to be `maxNegativePnlOnOpenP`.

Branches and code coverage

Intended branches

- `maxNegativePnlOnOpenP` must be set as value and emits event.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not manager.
  - ☐ Negative test

Function: `setOnePercentDepthArray(uint256[] indices, uint256[] valuesAbove, uint256[] valuesBelow)`

This is a function for setting the multiple `OnePercentDepth`.

Inputs

- `indices`
  - **Control**: Arbitrary.
  - **Constraints**: Must be the same length with `valuesAbove` and `valuesBelow`.
  - **Impact**: Array of the `pairIndex`.
- `valuesAbove`
  - **Control**: Arbitrary.
  - **Constraints**: Must be the same length with `indices`.
  - **Impact**: Array of the `onePercentDepthAbove`.
- `valuesBelow`
  - **Control**: Arbitrary.
  - **Constraints**: Must be the same length with `indices`.
  - **Impact**: Array of the `onePercentDepthBelow`.

Branches and code coverage

Intended branches

- Set each `pairIndex`'s `onePercentDepthAbove` and `onePercentDepthBelow`.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not a manager.
  - ☐ Negative test
- Revert if the length of given inputs are not the same.
  - ☐ Negative test

Function: `setOnePercentDepth(uint256 pairIndex, uint256 valueAbove, uint256 valueBelow)`

This function sets one percent depth for the pair.

Inputs

- `pairIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of the pair.
- `valueAbove`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The value of `onePercentDepthAbove` to be for the pair of `pairIndex`.
- `valueBelow`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The value of `onePercentDepthBelow` to be for the pair of `pairIndex`.

Branches and code coverage

Intended branches

- `onePercentDepth` and `onePercentDepthBelow` must be set following the given `valueAbove` and `valueBelow` on the `PairParams` of `pairIndex`.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not a manager.
  - ☐ Negative test

Function: `setPairParamsArray(uint256[] indices, PairParams[] values)`

This is a function for setting the multiple pair parameters.

Inputs

- `indices`
  - **Control**: Arbitrary.
  - **Constraints**: Must have the same length with `values`.
  - **Impact**: Array of `pairIndex`.
- `values`
  - **Control**: Arbirary.
  - **Constraints**: Must have the same length with `indices`.
  - **Impact**: Array of pair parameters.

Branches and code coverage

Intended branches

- Set the given multiple pairs' parameter.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not a manager.
  - ☐ Negative test
- Revert if the `indices`'s length is different with `values`'s length.
  - ☐ Negative test

Function call analysis

- `this.setPairParams(indices[i], values[i])`
  `-> this.storeAccFundingFees(pairIndex)`
  `-> this.getPendingAccFundingFees(pairIndex)`
  `-> this.storageT.openInterestHoney(pairIndex, 0)`
  - **What is controllable?** `indices` and `values` for `setPairParams()` — `pairIndex` for other functions.
  - **If the return value is controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.setPairParams(indices[i], values[i])`
  `-> this.storeAccFundingFees(pairIndex)`
  `-> this.getPendingAccFundingFees(pairIndex)`

-> `this.storageT.openInterestHoney(pairIndex, 1)`

- **What is controllable?** `indices` and `values` for `setPairParams()` — `pairIndex` for other functions.
- **If the return value is controllable, how is it used and how can it go wrong?** Not used.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `setPairParams(uint256 pairIndex, PairParams value)`

This function sets the parameters for the pair.

Inputs

- `pairIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: The index of the pair to select.
- `value`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Pair parameter to apply.

Branches and code coverage

Intended branches

- Store the acc rollover fees (store right before fee percent update).
  - ☐ Test coverage
- Store the acc funding fees (store right before trades opened/closed and fee percent update).
  - ☐ Test coverage
- Save the `value` for the `pairIndex`.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not a manager.
  - ☐ Negative test

Function call analysis

- `this.storeAccFundingFees(pairIndex)`

-> `this.getPendingAccFundingFees(pairIndex)`
-> `this.storageT.openInterestHoney(pairIndex, 0)`
    - **What is controllable?** `pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Not used.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storeAccFundingFees(pairIndex)`
  -> `this.getPendingAccFundingFees(pairIndex)`
  -> `this.storageT.openInterestHoney(pairIndex, 1)`
    - **What is controllable?** `pairIndex`.
    - **If the return value is controllable, how is it used and how can it go wrong?** Not used.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `setRolloverFeePerBlockPArray(uint256[] indices, uint256[] values)`

This is a function for setting the multiple `rolloverFeePerBlockP`.

Inputs

- `indices`
    - **Control**: Arbitrary.
    - **Constraints**: Must be the same length with `values`.
    - **Impact**: Array of `pairIndex`.
- `values`
    - **Control**: Arbitrary.
    - **Constraints**: Must be the same length with `indices`.
    - **Impact**: Array of `rolloverFeePerBlockP`.

Branches and code coverage

Intended branches

- Set the each `pairIndex`'s `rolloverFeePerBlockP`.
    - ☐ Test coverage

Negative behavior

- Revert if the caller is not a manager.
    - ☐ Negative test

- Revert if the length of `indices` and `values` are not the same.
  - ☐ Negative test

Function: `setRolloverFeePerBlockP(uint256 pairIndex, uint256 value)`

This function sets the rollover fee for the pair.

Inputs

- `pairIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of the pair.
- `value`
  - **Control**: Arbitrary.
  - **Constraints**: Must be less than 25,000,000.
  - **Impact**: Value for rollover fee of the pair.

Branches and code coverage

Intended branches

- The pair's `rolloverFeePerBlockP` must be set as given `value`.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not a manger.
  - ☐ Negative test
- Revert if the value is bigger than 25,000,000.
  - ☐ Negative test

Function: `storeAccFundingFees(uint256 pairIndex)`

This function sets `accPerOiLong` and `accPerOiShort` of the pair.

Inputs

- `pairIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of the pair.

Function: `storeAccRolloverFees(uint256 pairIndex)`

This function sets `accPerCollateral` pending acc rollover fee.

Inputs

- `pairIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of the pair.

Function: `storeTradeInitialAccFees(address trader, uint256 pairIndex, uint256 index, bool long)`

This function stores trade details and is called when the trade is opened.

Inputs

- `trader`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the trader.
- `pairIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of the pair.
- `index`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: Index of trade.
- `long`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Boolean value of the position (LONG, SHORT).

Branches and code coverage

Intended branches

- Set `rollover`, `funding`, and `openedAfterUpdate` with given inputs.
  - ☐ Test coverage.

Negative behavior

- Revert if the transaction is not from callback.
  - ☐ Negative test.

Function call analysis

- `this.storeAccFundingFees(pairIndex)`
  `-> this.getPendingAccFundingFees(pairIndex)`
  `-> this.storageT.openInterestHoney(pairIndex, 0)`
  - **What is controllable?** `pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The expected rollover will be returned, and if the function returns the wrong value, the trade can be broken.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storeAccFundingFees(pairIndex)`
  `-> this.getPendingAccFundingFees(pairIndex)`
  `-> this.storageT.openInterestHoney(pairIndex, 1)`
  - **What is controllable?** `pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** None.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.5. Module: PairsStorage.sol

Function: `addFee(Fee _fee)`

This function adds `fee` on `fees`. Note that this is not the function that increases the amount of fee.

Inputs

- `_fee`
  - **Control**: Arbitrary.
  - **Constraints**: Satisfy the `feeOk` condition.
  - **Impact**: `Fee` to add.

Function: `addGroup(Group _group)`

This function adds the group.

Inputs

- `_group`
  - **Control**: Arbitrary.
  - **Constraints**: Satisfy the `groupOk` condition.
  - **Impact**: Group to add.

Function: `addPairs(Pair[] _pairs)`

This function adds multiple pairs. Note that this function has no permission or right pair check but the `addPair` function has them.

Inputs

- `_pairs`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Array of the pairs.

Function call analysis

- `addPair(_pairs[i])`
  - **What is controllable?** `_pairs`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `addPair(Pair _pair)`

This function adds the pair.

Inputs

- `_pair`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be already listed.
  - **Impact**: Pair to add.

Branches and code coverage

Intended branches

- The pair must be listed on `pairs` and emit `PairAdded` event.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not `Gov`.
  - ☐ Negative test
- Revert if the feed of `_pair` has less than zero as `maxDeviationP` of `feed`.
  - ☐ Negative test
- Revert if the feed of `_pair` has `feed1` as zero address of `feed`.
  - ☐ Negative test
- Revert if the group of `_pair` is not listed.
  - ☐ Negative test
- Revert if the fee of `_pair` is not listed.
  - ☐ Negative test

Function: `initialize(ITradingStorage _storageT)`

This sets `storageT` as an initializer.

Inputs

- `_storageT`
  - **Control**: None.
  - **Constraints**: Must not be zero address.
  - **Impact**: Sets a trading storage for an initialization.

Branches and code coverage

Intended branches

- Set `storageT` if `_storageT` it not a zero address.
  - ☐ Test coverage

Negative behavior

- The `initialize` function must be called only once.
  - ☐ Negative test
- The `_storageT` must not be a zero address.
  - ☐ Negative test

Function: `updateFee(uint256 _id, Fee _fee)`

This function updates the `Fee`.

Inputs

- `_id`
  - **Control**: Arbitrary.
  - **Constraints**: Listed at fee.
  - **Impact**: ID of a fee to update.
- `_fee`
  - **Control**: Arbitrary.
  - **Constraints**: Satisfy the `feeOk` condition.
  - **Impact**: New fee.

Function: `updateGroupCollateral(uint256 _pairIndex, uint256 _amount, bool _long, bool _increase)`

This function increases or decreases collateral for the group.

Inputs

- `_pairIndex`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Index of the pair.
- `_amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount to increase/decrease.
- `_long`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Long or not.
- `_increase`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Increase or decrease.

Branches and code coverage

Intended branches

- The `collateralOpen` of a group must be updated following the given inputs.
  - ☐ Test coverage

Negative behavior

- Revert if the function caller is not a callback address.
  - ☐ Negative test

Function call analysis

- `this.storageT.callbacks()`
  - **What is controllable?** None.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the address of callbacks.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `updateGroup(uint256 _id, Group _group)`

This function updates the group of given `_id`.

Inputs

- `_id`
  - **Control**: Arbitrary.
  - **Constraints**: Listed at group.
  - **Impact**: ID of a group to update.
- `_group`
  - **Control**: Arbitrary.
  - **Constraints**: Satisfy the `groupOk` condition.
  - **Impact**: New group.

Function: `updatePair(uint256 _pairIndex, Pair _pair)`

This function updates `feed`, `spreadP`, and `feeIndex` of the pair.

Inputs

- `_pairIndex`

- **Control**: Arbitrary.
- **Constraints**: None.
- **Impact**: Index of the pair.

- `_pair`
  - **Control**: Arbitrary.
  - **Constraints**: Right feed for the pair.
  - **Impact**: A pair to be.

Branches and code coverage

Intended branches

- The destination pair's `feed`, `spreadP`, and `feeIndex` must be updated as a given pair.
  - ☐ Test coverage

Negative behavior

- Revert if the caller is not `Gov`.
  - ☐ Negative test
- Revert if the feed of `_pair` has less than zero as `maxDeviationP` of `feed`.
  - ☐ Negative test
- Revert if the feed of `_pair` has `feed1` as zero address of `feed`.
  - ☐ Negative test
- Revert if the fee of `_pair` is not listed.
  - ☐ Negative test

5.6. Module: PoLRewarder.sol

Function: `accRewards(address owner)`

This function is used to calculate the accrued BGT rewards for a user.

Inputs

- `owner`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the user to calculate rewards for.

Branches and code coverage

Intended branches

- Return the accrued BGT for the user.
  - ☐ Test coverage

Negative behavior

- Revert if the total supply of share token is equal to or lower than zero.
  - ☐ Negative Test

Function: `distributeFees(address sender, uint256 amount)`

This function is used to distribute fees to the distribution module.

Inputs

- `sender`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the sender.
- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of fees to distribute

Branches and code coverage

Intended branches

- Send the fees to the distribution module.
  - ☑ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☑ Negative test

Function: `getAvailableBGT()`

This function is used to get the available BGT rewards from rewards module.

Branches and code coverage

Intended branches

- Return the available BGT rewards.

- ☐ Test coverage

Negative behavior

- Revert if the total supply of share token is lower than zero
  - ☐ Negative Test

Function: `harvestRewards(uint256 amount, address claimer, address recipient)`

This function is used to harvest BGT rewards from the rewards module and update the global BGT and user BGT.

Inputs

- `amount`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of BGT to harvest.
- `claimer`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the claimer of the rewards.
- `recipient`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the recipient of the rewards.

Branches and code coverage

Intended branches

- Update the global bgt and claimer bgt.
  - ☐ Test coverage
- Withdraw the rewards from the rewards module.
  - ☐ Test coverage
- Update the global bgt after withdrawal.
  - ☐ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☑ Negative test

Function: `isRewardsAcc()`

This function is used to check if there are any BGT rewards available.

Branches and code coverage

Intended branches

- Check if there are any BGT rewards available.
  - ☐ Test coverage

Negative behavior

- N/A.

Function: `onBurn(address owner, uint256 shares)`

This function is used to hooking withdraw and burn functions in BToken, and update the glboal bgt, and user bgt.

Inputs

- `owner`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of BToken holder.
- `shares`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of shares bruned.

Branches and code coverage

Intended branches

- Update the global bgt.
  - ☐ Test coverage
- Update the user's bgt.
  - ☐ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☑ Negative test

Function: `onMint(address receiver, uint256 shares)`

This function is used to hooking deposit and mint functions in BToken, and update the glboal bgt, and user bgt.

Inputs

- `receiver`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of BToken receiver.
- `shares`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of shares minted.

Branches and code coverage

Intended branches

- Update the global bgt.
  - ☐ Test coverage
- Update the user's bgt.
  - ☐ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☑ Negative test

Function: `onTransfer(address from, address to, uint256 shares)`

This function is used to hooking transferring in BToken, and update the glboal bgt, and user bgt.

Inputs

- `from`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of BToken sender
- `to`
  - **Control**: Arbitrary.
  - **Constraints**: None.

- **Impact**: Address of BToken receiver
- `shares`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of shares transferred.

Branches and code coverage

Intended branches

- Update the global bgt.
  - ☑ Test coverage
- Update the sender's bgt.
  - ☑ Test coverage
- Update the receiver's bgt.
  - ☑ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☑ Negative test
- Return if from is equal to to.
  - ☑ Negative test

Function: `transferOwnership(address newOwner)`

This function is used to transfer the ownership of the contract to a new owner.

Inputs

- `newOwner`
  - **Control**: Arbitrary.
  - **Constraints**: Must not be the zero address.
  - **Impact**: Address of newOwner.

Branches and code coverage

Intended branches

- Update the owner to the newOwner.
  - ☑ Test coverage

Negative behavior

- Reverts if caller is not the owner.
  - ☑ Negative test
- Reverts if the newOwner is the zero address.
  - ☑ Negative test

Function: `updateGlobalBGT()`

This function is used to update the global BGT available balance and the accrued BGT per share.

Branches and code coverage

Intended branches

- Update the global BGT available.
  - ☐ Test coverage
- Update the accrued BGT per share.
  - ☐ Test coverage

Negative behavior

- Calculate accBGTPerShare when supply is greater than 0.
  - ☑ Negative test

Function: `updateUserBGT(address receiver, uint256 sharesDelta, bool isMint)`

This function is used to update the accrued and debt BGT for a user.

Inputs

- `receiver`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Address of the receiver.
- `sharesDelta`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Amount of shares to update.
- `isMint`
  - **Control**: Arbitrary.
  - **Constraints**: None.
  - **Impact**: Whether the shares are minted or burned.

Branches and code coverage

Intended branches

- Update the user's accrued BGT.
  - ☐ Test coverage
- Update the user's debt BGT.
  - ☐ Test coverage

Negative behavior

- Revert if the total supply of share token is equal to or lower than zero.
  - ☐ Negative Test

5.7. Module: Referrals.sol

Function: `distributePotentialReward(address trader, uint256 volumeHoney, uint256 pairOpenFeeP)`

This functions distributes the rewards.

Inputs

- `trader`
  - **Control**: Arbitrary.
  - **Constraints**: Must be true for `active`.
  - **Impact**: Address of the trader.
- `volumeHoney`
  - **Control**: None (calculated with trade's `positionSizeHoney` and `leverage`).
  - **Constraints**: None.
  - **Impact**: Volume of Honey.
- `pairOpenFeeP`
  - **Control**: None (the return value of `pairOpenFeeP()` from PairStorage contract).
  - **Constraints**: None.
  - **Impact**: Percent of open fee.

Branches and code coverage

Intended branches

- The referral gets the HONEY reward.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not callbacks address.
  - ☐ Negative test
- Revert if referrer is not active.
  - ☐ Negative test

Function call analysis

- `this.storageT.transferHoney(address(this.storageT), referrer, referrerRewardValueHoney)`
  - **What is controllable?** `referrer`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `initialize(ITradingStorage _storageT, uint256 _startReferrerFeeP, uint256 _openFeeP, uint256 _targetVolumeHoney)`

This sets `storageT`, `startReferrerFeeP`, `openFeeP`, and `targetVolumeHoney` as an initializer.

Inputs

- `_storageT`
  - **Control**: None.
  - **Constraints**: Not a zero address.
  - **Impact**: Address of `storageT`.
- `_startReferrerFeeP`
  - **Control**: None.
  - **Constraints**: Less than 100.
  - **Impact**: Percent of referrer fee when zero volume.
- `_openFeeP`
  - **Control**: None.
  - **Constraints**: Less than 50.
  - **Impact**: Percent of opening fee used for referral system.
- `_targetVolumeHoney`
  - **Control**: None.
  - **Constraints**: Larger than zero.
  - **Impact**: Amount of HONEY to reach maximum referral system.

Branches and code coverage

Intended branches

- Initialize the values of `storageT`, `startReferrerFeeP`, `openFeeP`, and `targetVolumeHoney`.
  - ☐ Test coverage

Negative behavior

- Revert if the function is called multiple times.
  - ☐ Negative test
- Revert if `_storageT` is a zero address.
  - ☐ Negative test
- Revert if `_startReferrerFeeP` is larger than 100.
  - ☐ Negative test
- Revert if `_openFeeP` is larger than 50.
  - ☐ Negative test
- Revert if `_targetVolumeHoney` is less than 0.
  - ☑ Negative test

Function: `registerPotentialReferrer(address referrer)`

This function registers the referrer.

Inputs

- `referrer`
  - **Control**: Arbitrary.
  - **Constraints**: Not a zero address.
  - **Impact**: Address of referrer.

Function: `updateOpenFeeP(uint256 value)`

This function updates `openFeeP`.

Inputs

- `value`
  - **Control**: Arbitrary.
  - **Constraints**: Less than 50.
  - **Impact**: Value to be used for update.

Function: `updateStartReferrerFeeP(uint256 value)`

This function updates `StartReferrerFeeP`.

Inputs

- `value`
  - **Control**: Arbitrary.
  - **Constraints**: Less than 100.
  - **Impact**: Value to be used for update.

Function: `updateTargetVolumeHoney(uint256 value)`

This function updates `targetVolumeHoney`.

Inputs

- `value`
  - **Control**: Arbitrary.
  - **Constraints**: Larger than zero.
  - **Impact**: Value to be used for update.

5.8. Module: TradingCallbacks.sol

Function: `closeTradeMarketCallback(AggregatorAnswer a, ITradingStorage.PendingMarketOrder o)`

This is a callback of `closeTrade` and unregisters and removes the trade from storage.

Inputs

- `a`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: None.
  - **Impact**: The oracle price and `spreadP` of the index.
- `o`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: None.
  - **Impact**: The struct of pended market order.

Branches and code coverage

Intended branches

- Close the trade.
  - ☐ Test coverage

Negative behavior

- Revert if the market order is already closed.
  - ☐ Negative test

Function call analysis

- `getOpenTrade(o.trade.trader, o.trade.pairIndex, o.trade.index)`
  - **What is controllable?** `o.trade.trader`, `o.trade.pairIndex`, and `o.trade.index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the trade.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `getOpenTradeInfo(t.trader, t.pairIndex, t.index)`
  - **What is controllable?** `t.trader`, `t.pairIndex`, and `t.index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the trade information.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `currentPercentProfit(t.openPrice, a.price, t.buy, t.leverage)`
  - **What is controllable?** `t.openPrice`, `t.buy`, and `t.leverage`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the current profit percentages.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `unregisterTrade(t,true,v.profitP,v.posHoney,i.openInterestHoney, (v.levPosHoney * getPairsStorage().pairCloseFeeP(t.pairIndex)) / 100 / PRECISION, (v.levPosHoney * getPairsStorage().pairLimitOrderFeeP(t.pairIndex)) / 100 / PRECISION)`
  - **What is controllable?** `t`, `v.profitP`, `v.posHoney`, `v.levPosHoney`, and `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The Honey sent to trader — incorrect values may lead to incorrect trade-information retrieval.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `currentPercentProfit(uint256 openPrice, uint256 currentPrice, bool buy, uint256 leverage)`

This calculates the current profit percentages.

Inputs

- `openPrice`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The price when the trade was opened.
- `currentPrice`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The current price.
- `buy`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The type of an order (`LONG`, `SHORT`).
- `leverage`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The size of a leverage.

Branches and code coverage

Intended branches

- Calculates the proper current profit percentage with `leverage` and returns it.
  - ☐ Test coverage

Negative behavior

- Revert if the `p` is bigger than 900%.
  - ☐ Negative test
- Revert if `currentPrice` or `openPrice` has changed their symbols due to the `int256` type casting.
  - ☐ Negative test

Function: `executeLimitCloseOrderCallback(AggregatorAnswer a, ITradingStorage.PendingLimitOrder o)`

This executes limit-close order and unregisters the trade.

Inputs

- `a`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: None.
  - **Impact**: The oracle price and `spreadP` of the index.
- `o`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: None.
  - **Impact**: The struct of pended market order.

Branches and code coverage

Intended branches

- Close the trade.
  - ☐ Test coverage

Negative behavior

- Revert if the market order is already closed.
  - ☐ Negative test

Function call analysis

- `getOpenTrade(o.trade.trader, o.trade.pairIndex, o.trade.index)`
  - **What is controllable?** `o.trade.trader`, `o.trade.pairIndex`, and `o.trade.index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the trade.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `getOpenTradeInfo(t.trader, t.pairIndex, t.index)`
  - **What is controllable?** `t.trader`, `t.pairIndex`, and `t.index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the trade information.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `pairsStored.guaranteedSlEnabled(t.pairIndex)`
  - **What is controllable?** `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Checks the `groupIndex` of the trade is not zero.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `borrowingFees.getTradeLiquidationPrice(LiqPriceInput(t.trader, t.pairIndex, t.index, t.openPrice, t.buy, v.posHoney, t.leverage))`

- **What is controllable?** `t.trader`, `t.pairIndex`, `t.index`, `t.openPrice`, `t.buy`, and `t.leverage`.
- **If the return value is controllable, how is it used and how can it go wrong?** Retrieves trade-liquidation price, including `borrowing fee` and `funding fee`.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `currentPercentProfit(t.openPrice, a.price, t.buy, t.leverage)`
  - **What is controllable?** `t.openPrice`, `t.buy`, and `t.leverage`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the current profit percentages.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `unregisterTrade(t, false, v.profitP, v.posHoney, i.openInterestHoney, o.orderType == ITradingStorage.LimitOrder.LIQ ? v.reward1 : (v.levPosHoney * pairsStored.pairCloseFeeP(t.pairIndex)) / 100 / PRECISION, v.reward1)`
  - **What is controllable?** `t`, `v.profitP`, `v.posHoney`, `i.openInterestHoney`, `v.levPosHoney`, and `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The Honey sent to trader — incorrect values may lead to incorrect trade-information retrieval.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `executeLimitOpenOrderCallback(AggregatorAnswer a, ITradingStorage.PendingLimitOrder n)`

This executes limit-open order, registers the trade, and removes open limit order of storage.

Inputs

- `a`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: None.
  - **Impact**: The oracle price and `spreadP` of the index.
- `o`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: None.
  - **Impact**: The struct of pended market order.

Branches and code coverage

Intended branches

- Calculates price impact for the trade to be opened.
  - ☐ Test coverage
- To executes limit order, registers the trade and then removes open limit order of the storage.
  - ☐ Test coverage

Negative behavior

- Revert if the contract is paused.
  - ☐ Negative test
- Revert if the oracle price is zero.
  - ☐ Negative test
- Revert if the limit order is already closed.
  - ☐ Negative test
- Revert if the trade is within the maximum open interest limit.
  - ☐ Negative test
- Revert if the `priceImpactP` multiplied by leverage is bigger than `maxNegativePnlOnOpenP`.
  - ☐ Negative test
- Revert if the leverage is in the incorrect range.
  - ☐ Negative test

Function call analysis

- `pairInfos.getTradePriceImpact(marketExecutionPrice(a.price, a.spreadP, o.spreadReductionP, o.buy),o.pairIndex,o.buy,o.positionSizeHoney * o.leverage)`
  - **What is controllable?** `o.spreadReductionP`, `o.buy`, `o.pairIndex`, and `o.positionSizeHoney * o.leverage`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the price impact for the trade.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `withinExposureLimits(o.pairIndex, o.buy, o.positionSizeHoney, o.leverage)`
  - **What is controllable?** `o.pairIndex`, `o.buy`, `o.positionSizeHoney`, and `o.leverage`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Checks the trade is within maximum open interest limit.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `withinMaxLeverage(o.pairIndex, o.leverage)`
  - **What is controllable?** `t.pairIndex` and `t.leverage`.

- **If the return value is controllable, how is it used and how can it go wrong?** Checks the leverage in the correct range.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `registerTrade(Trade( o.trader, o.pairIndex, 0, 0, o.positionSize, a.price, o.buy, o.leverage, o.tp, o.sl))`
  - **What is controllable?** `o.trader`, `o.pairIndex`, `o.positionSize`, `o.buy`, `o.leverage`, `o.tp`, and `o.sl`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Registers the trade — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `storageT.unregisterOpenLimitOrder(o.trader, o.pairIndex, o.index)`
  - **What is controllable?** `o.trader`, `o.pairIndex`, and `o.index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Unregisters opened limit order — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `openTradeMarketCallback(AggregatorAnswer a, ITradingStorage.PendingMarketOrder o)`

This is a callback of `openTrade` and registers and stores the trade.

Inputs

- `a`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: None.
  - **Impact**: The oracle price and `spreadP` of the index.
- `o`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: None.
  - **Impact**: The struct of pended market order.

Branches and code coverage

Intended branches

- Calculates price impact for the trade to be opened.
  - ☐ Test coverage
- Opens new trade.

- ☐ Test coverage

Negative behavior

- Revert if the contract is paused.
  - ☐ Negative test
- Revert if the oracle price is zero.
  - ☐ Negative test
- Revert if the difference of the wanted price with slippage is bigger than the maximum slippage.
  - ☐ Negative test
- Revert if the trade is within maximum open interest limit.
  - ☐ Negative test
- Revert if the `priceImpactP` multiplied by leverage is bigger than `maxNegativePnlOnOpenP`.
  - ☐ Negative test
- Revert if the leverage is in the incorrect range.
  - ☐ Negative test

Function call analysis

- `pairInfos.getTradePriceImpact(marketExecutionPrice(a.price, a.spreadP, o.spreadReductionP, t.buy),t.pairIndex,t.buy,t.positionSizeHoney * t.leverage)`
  - **What is controllable?** `o.spreadReductionP`, `t.buy`, `t.pairIndex`, and `t.positionSizeHoney * t.leverage`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the price impact for the trade.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `withinExposureLimits(t.pairIndex, t.buy, t.positionSizeHoney, t.leverage)`
  - **What is controllable?** `t.pairIndex`, `t.buy`, `t.positionSizeHoney`, and `t.leverage`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Checks the trade is within maximum open interest limit.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `withinMaxLeverage(t.pairIndex, t.leverage)`
  - **What is controllable?** `t.pairIndex` and `t.leverage`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Checks the leverage is in the correct range.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `registerTrade(t)`
  - **What is controllable?** `t` stands for trade.

- **If the return value is controllable, how is it used and how can it go wrong?** Registers the trade — no return value.
- **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

- `transferFromStorageToAddress(t.trader, t.positionSizeHoney)`
  - **What is controllable?** `t.trader` and `t.positionSizeHoney`.
  - **If the return value is controllable, how is it used and how can it go wrong?** If an order is canceled, Honey will be returned to the trader — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

5.9. Module: Trading.sol

Function: `cancelOpenLimitOrder(uint256 pairIndex, uint256 index)`

This cancels an open limit order.

Inputs

- `pairIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the trading pair.
- `index`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the order.

Branches and code coverage

Intended branches

- Unregisters the open limit order.
  - ☐ Test coverage
- Transfers HONEY to the trader.
  - ☐ Test coverage

Negative behavior

- Revert if the `sender` does not have the open limit order.
  - ☐ Negative test

Function call analysis

- `this.storageT.hasOpenLimitOrder(sender, pairIndex, index)`
  - **What is controllable?** `sender`, `pairIndex`, and `index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns boolean according to whether the `sender` does have the open limit order.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.getOpenLimitOrder(sender, order.pairIndex, order.index)`
  - **What is controllable?** `sender`, `order.pairIndex`, and `order.index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the open limit order; this limit order is updated and later stored in storage.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.unregisterOpenLimitOrder(sender, order.pairIndex, order.index)`
  - **What is controllable?** `sender`, `order.pairIndex`, and `order.index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.transferHoney(address(storageT), sender, o.positionSize)`
  - **What is controllable?** `sender` and `o.positionSize`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Transfers the HONEY balance associated with the canceled order back to the caller — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `closeTradeMarket(uint256 pairIndex, uint256 index)`

This closes trade for `MARKET` order of `msg.sender`.

Inputs

- `pairIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the trading pair for the open trade.
- `index`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the open trade.

Branches and code coverage

Intended branches

- The `getPrice()` function fetches the price from the oracle then calls the callback function in `TradingCallbacks`.
  - ☐ Test coverage
- The callback function unregisters the trade and unregisters the pending market order.
  - ☐ Test coverage

Negative behavior

- Revert if the market order is already closed.
  - ☐ Negative test
- Revert if the leverage of the trade is zero.
  - ☐ Negative test

Function call analysis

- `this.storageT.openTrades(sender, pairIndex, index)`
  - **What is controllable?** `sender`, `pairIndex`, and `index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Checks the existence of the open trade; incorrect values may lead to incorrect trade-information retrieval.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.openTradesInfo(sender, pairIndex, index)`
  - **What is controllable?** `sender`, `pairIndex`, and `index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves additional information about the open trade; incorrect values may lead to incorrect information retrieval.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert; no reentrancy scenarios.
- `this.getPrice(pairIndex)`
  - **What is controllable?** `pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return `price` and `spreadP` of the index.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.TradingCallbacks.closeTradeMarketCallback(getPrice(pairIndex),PendingMar pairIndex, index, 0, 0, 0, false, 0, 0, 0),0,0,0))`
  - **What is controllable?** `pairIndex`, `sender`, `pairIndex`, and `index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Stores the pending market order — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `openTrade(ITradingStorage.Trade t, ITradingCallbacks.TradeType orderType, uint256 slippageP)`

This opens a new market/limit trade.

Inputs

- `t`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The details of the trade to open.
- `orderType`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Market or limit or stop-limit type of trade.
- `slippageP`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The slippage percentage.

Branches and code coverage

Intended branches

- If the order type is `MARKET`, store the pending market order and call open trade market callback to register the order and unregister the pending order.
  - ☐ Test coverage
- If the order type is `LIMIT`, store the open limit order.
  - ☐ Test coverage
- If TP and SL are provided, check if they are in correct range.
  - ☐ Test coverage

Negative behavior

- Revert if the open trades count plus the open limit-orders count is greater than or equal to the max trades per pair.
  - ☐ Negative test
- Revert if leverage is not in the correct range.
  - ☐ Negative test
- Revert if the position size multiplied by leverage is less than the minimum leverage position.
  - ☐ Negative test
- Revert if the trade price impact multiplied by leverage is higher than the max negative PNL percent on trade opening.

- ☐ Negative test

Function call analysis

- `this.storageT.pairsStorage()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returned value is the TradingStorage contract, to which calls will be made.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.openTradesCount(sender, t.pairIndex)`
  - **What is controllable?** `msg.sender` and `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the count of pending market orders for the user and trading pair.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.openLimitOrdersCount(sender, t.pairIndex)`
  - **What is controllable?** `msg.sender` and `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the count of open limit orders for the user and trading pair.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.maxTradesPerPair()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the max amount of trades per pair.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.pairsStorage.groupMaxCollateral(t.pairIndex)`
  - **What is controllable?** `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the maximum collateral of group for the trading pair.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.pairsStorage.pairMinLevPosHoney(t.pairIndex)`
  - **What is controllable?** `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the minimum leverage position HONEY for the trading pair.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.pairsStorage.pairMinLeverage(t.pairIndex)`
  - **What is controllable?** `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the minimum leverage for the trading pair.

  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.pairMaxLeverage(pairsStorage, t.pairIndex)`
  - **What is controllable?** `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves the maximum leverage for the trading pair.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.pairInfos.getTradePriceImpact(0, t.pairIndex, t.buy, t.positionSizeHoney * t.leverage)`
  - **What is controllable?** `t.pairIndex`, `t.buy`, and `t.positionSizeHoney * t.leverage`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves a dynamic price-impact value on trade opening.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.transferHoney(sender, address(storageT), t.positionSizeHoney)`
  - **What is controllable?** `sender` and `t.positionSizeHoney`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Transfers Honey from the caller to the storage contract.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.firstEmptyOpenLimitIndex(sender, t.pairIndex)`
  - **What is controllable?** `sender` and `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Finds the first empty open limit index for the user and trading pair.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.storeOpenLimitOrder(ITradingStorage.OpenLimitOrder(sender, t.pairIndex, index, t.positionSizeHoney, 0, t.buy, t.leverage, t.tp, t.sl, t.openPrice, t.openPrice, block.timestamp))`
  - **What is controllable?** `sender`, `t.pairIndex`, `index`, `t.positionSizeHoney`, `t.buy`, `t.leverage`, `t.tp`, `t.sl`, and `t.openPrice`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Stores an open limit order — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.TradingCallbacks.setOpenOrderType(sender, t.pairIndex, index, orderType)`
  - **What is controllable?** `sender`, `t.pairIndex`, `index`, and `orderType`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Set open order type — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?**

N/A.

- `this.TradingCallbacks.setTradeLastUpdated(simplifiedTradeId, lastUpdated)`
  - **What is controllable?** `simplifiedTradeId` and `lastUpdated`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Set trade last updated — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.getPrice(t.pairIndex)`
  - **What is controllable?** `t.pairIndex`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Return `price` and `spreadP` of the index.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.TradingCallbacks.openTradeMarketCallback(getPrice(t.pairIndex), ITradingStorage.PendingMarketOrder(ITradingStorage.Trade(sender,t.pairIndex,C`
  - **What is controllable?** `sender`, `t.pairIndex`, `t.positionSizeHoney`, `t.buy`, `t.leverage`, `t.tp`, `t.sl`, `t.openPrice`, and `slippageP`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Stores the pending market order — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `updateOpenLimitOrder(uint256 pairIndex, uint256 index, uint256 price, uint256 tp, uint256 sl)`

This updates an open limit order.

Inputs

- `pairIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the trading pair.
- `index`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the order.
- `price`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The price level to set (`_PRECISION`).

- `tp`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The take-profit price.
- `sl`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The stop-loss price.

Branches and code coverage

Intended branches

- If the new TP and SL are in the correct range, update the open limit order.
  - ☐ Test coverage

Negative behavior

- Revert if the `sender` does not have the open limit order.
  - ☐ Negative test
- Revert if `tp` is set and not valid according to order type.
  - ☐ Negative test
- Revert if `sl` is set and not valid according to order type.
  - ☐ Negative test

Function call analysis

- `this.storageT.hasOpenLimitOrder(sender, pairIndex, index)`
  - **What is controllable?** `sender`, `pairIndex`, and `index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns boolean according to whether the `sender` does have the open limit order.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.updateOpenLimitOrder(storageT.getOpenLimitOrder(sender, pairIndex, index))`
  - **What is controllable?** `sender`, `pairIndex`, and `index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Updates the open limit order based on the provided information — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.

Function: `updateSl(uint256 pairIndex, uint256 index, uint256 newSl)`

This updates the stop loss for an open trade.

Inputs

- `pairIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the trading pair.
- `index`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the order.
- `newSl`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The new stop-loss price.

Branches and code coverage

Intended branches

- Updates new stop-loss price.
  - ☐ Test coverage

Negative behavior

- Revert if the `sender` does not have the open limit order.
  - ☐ Negative test
- Revert if `newSl` is in the correct range.
  - ☐ Negative test
- Revert if `newSl` deviates more than `maxSlDist`.
  - ☐ Negative test

Function call analysis

- `this.storageT.openTrades(sender, pairIndex, index)`
  - **What is controllable?** `sender`, `pairIndex`, and `index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Checks the existence of the open trade; incorrect values may lead to incorrect trade-information retrieval.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.storageT.pairsStorage().guaranteedSlEnabled(pairIndex)`

- **What is controllable?** `pairIndex`.
- **If the return value is controllable, how is it used and how can it go wrong?** Checks the `groupIndex` of the trade is not zero.
- **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `this.storageT.updateSl(sender, pairIndex, index, newSl)`
  - **What is controllable?** `sender`, `pairIndex`, `index`, and `newSl`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Updates the SL value — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.openTradesInfo(sender, pairIndex, index)`
  - **What is controllable?** `sender`, `pairIndex`, and `index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves additional information about the open trade; incorrect values may lead to incorrect information retrieval.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.TradingCallbacks.updateSlCallback(getPrice(pairIndex),PendingSl(sender, pairIndex, index, t.openPrice, t.buy, newSl))`
  - **What is controllable?** `pairIndex`, `sender`, `pairIndex`, `index`, `t.openPrice`, `t.buy`, and `newSl`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `updateTp(uint256 pairIndex, uint256 index, uint256 newTp)`

This updates the take profit for an open trade.

Inputs

- `pairIndex`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the trading pair.
- `index`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The index of the order.
- `newTp`

- **Control**: Fully controlled by the caller.
- **Constraints**: None.
- **Impact**: The new take-profit price.

Branches and code coverage

Intended branches

- Updates new take-profit price.
  - ☐ Test coverage

Negative behavior

- Revert if the `sender` does not have the open limit order.
  - ☐ Negative test
- Revert if new TP is in the correct range.
  - ☐ Negative test

Function call analysis

- `this.storageT.openTrades(sender, pairIndex, index)`
  - **What is controllable?** `sender`, `pairIndex`, and `index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Checks the existence of the open trade; incorrect values may lead to incorrect trade-information retrieval.
  - **What happens if it reverts, reenters or does other unusual control flow?** If it reverts, the entire call will revert — no reentrancy scenarios.
- `this.c.correctTp(t.openPrice, t.leverage, newTp, t.buy)`
  - **What is controllable?** `t.openPrice`, `t.leverage`, `newTp`, and `t.buy`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Checks `newTp` price is in the correct range — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.updateTp(sender, pairIndex, index, newTp)`
  - **What is controllable?** `sender`, `pairIndex`, `index`, and `newTp`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Updates the TP value — no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.storageT.openTradesInfo(sender, pairIndex, index)`
  - **What is controllable?** `sender`, `pairIndex`, and `index`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Retrieves additional information about the open trade; incorrect values may lead to incorrect information retrieval.
  - **What happens if it reverts, reenters or does other unusual control flow?**

If it reverts, the entire call will revert — no reentrancy scenarios.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the mainnet but deployed to their public testnet ↗.

During our assessment on the scoped Berachain BTS contracts, we discovered 30 findings. Eight critical issues were found. Four were of high impact, seven were of medium impact, seven were of low impact, and the remaining findings were informational in nature. Berachain acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the 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.