// HALBORN

Siren - GammaProtocol

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **December 11th, 2023 - January 26th, 2024**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	01/29/2024
0.2	Draft Review	01/29/2024
1.0	Remediation Plan	02/16/2024
1.1	Remediation Plan Review	02/19/2024

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Siren engaged Halborn to conduct a security assessment on their smart contract beginning on December 11th, 2023 and ending on January 26th, 2024. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 ASSESSMENT SUMMARY

Halborn was provided about seven weeks for the engagement and assigned one full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified several vulnerabilities of varying severity in the smart contract code, which were mostly addressed by the Siren team.

During the detailed assessment of the GammaProtocol and core-v4 repositories, including key contracts such as TradeExecutor, Gmx2Hedger, and others, a comprehensive analysis was conducted across various components and functionalities. The examination focused on ensuring operational efficiency, security, and optimal gas usage. The findings are categorized into several core areas:

Contract Initialization and Configuration
- Rigorous validation of parameters during contract initialization.
- Secure setting and updating of vital contract addresses and configurations.

Error Handling and Gas Optimization
- Evaluation of the use of string-based versus custom errors for improved gas efficiency.
- Assessment of error message handling in try-catch blocks for robust error management.

Margin Vault and Order Management
- Procedures for the creation and validation of margin vaults.
- Detailed review of order submission, execution, and cancellation processes.

Fee Calculation and Management
- Analysis of execution fee computation methods, focusing on dynamic vs. static approaches.
- Investigation of fee handling during order submissions and completions in varied transaction scenarios.

Position Management in Trading Operations
- Optimization strategies for managing long positions during oToken minting.
- Efficiency in burning long offsets compared to user transfers.

Role-Based Access Control and Security Measures
- Assessment of access control mechanisms and permissions for sensitive functions.
- Security review of critical functionalities like order execution and vault management.

Smart Contract Libraries and Interoperability
- Usage and reliability of libraries for managing orders, vaults, and hedging data.
- Inspection of library functions to handle edge cases and maintain contract integrity.

Upgrade Patterns and Contract Interactions
- Review of upgradeability patterns and their implications on contract functionality.
- Analysis of interactions among various contracts within the reposito-

ries.

Order and Trade Execution Processes
- Mechanisms and validations involved in order creation and execution.
- Trade execution procedures and their effects on contract states and user positions.

Summary of Findings
1. **Error Handling**: Identified potential for gas savings by transitioning from string-based to custom errors.
2. **Fee Calculations**: Noted inconsistency in fee calculations between order submission and completion.
3. **Position Management**: Observed opportunities for optimization in handling long positions during oToken minting.
4. **Vault Management**: Suggested improvements in vault selection logic to align with multiple vault scenarios.

Questions and Scenarios for Consideration
- Impact of processing orders with invalid or non-existent trader vault IDs.
- Effectiveness of error handling strategies in different error scenarios.
- Efficiency of long position management in minting processes.
- Appropriateness of fee calculation methods in dynamic transaction environments.
- Strategies for managing multiple vaults for the same underlying asset.
- Contract functionality and integrity under diverse edge cases and user actions.

This comprehensive analysis aimed to ensure the contracts' operational efficiency, security, and optimal gas utilization while maintaining their intended functionalities.

1.3 SCOPE

The assessment was scoped into the following smart contracts:

- contracts/core/Controller.sol
- contracts/core/MarginCalculator.sol
- contracts/core/calculators/NakedMarginCalculator.sol
- contracts/libs/Actions.sol
- contracts/libs/MarginVault.sol

Commit ID: 7d3b05ffdedf2abf2ab3906043011c04a0e724ec (order_flow_redo branch)

Repository URL: https://github.com/sirenmarkets/GammaProtocol/tree/order_flow_redo

- contracts/core/HedgedPool.sol
- contracts/core/LpManager.sol
- contracts/core/TradeExecutor.sol
- contracts/core/hedgers/Gmx2Hedger.sol

Commit ID: 1a77b36821bec584991efb35032ebe5529835df5 (order-flow-redo branch)

Repository URL: https://github.com/sirenmarkets/core-v4/tree/order-flow-redo

1.4 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.

- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions (solgraph).
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Testnet deployment (Foundry).

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker’s control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A)	1
	Specific (AO:S)	0.2
Attack Cost (AC)	Low (AC:L)	1
	Medium (AC:M)	0.67
	High (AC:H)	0.33
Attack Complexity (AX)	Low (AX:L)	1
	Medium (AX:M)	0.67
	High (AX:H)	0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
4	5	3	4	12

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) UNHANDLED ZERO PRICE IN MARGINCALCULATOR LEADING TO INVALID CALCULATIONS	High (7.5)	RISK ACCEPTED
(HAL-02) UNINITIALIZED SPOTSHOCKVALUE LEADING TO DIVISION BY ZERO	High (7.5)	RISK ACCEPTED
(HAL-03) UNINITIALIZED IMPLEMENTATION IN UPGRADEABLE CONTRACT	Medium (6.2)	SOLVED - 02/15/2024
(HAL-04) UNNECESSARY MEMORY STORE UPDATES IN RUNACTIONS FUNCTION	Informational (0.0)	FUTURE RELEASE
(HAL-05) REDUNDANT PARAMETER IN ONLYAUTHORIZED MODIFIER	Informational (0.0)	FUTURE RELEASE
(HAL-01) UNDERFLOW IN RESET PENDING ORDER	Critical (10)	SOLVED - 02/15/2024
(HAL-02) MISMATCHED RETURN VALUES	Critical (10)	NOT APPLICABLE
(HAL-03) UNSET SWAP PATH	Critical (10)	NOT APPLICABLE
(HAL-04) INEFFECTIVE ERROR HANDLING IN EXECUTE ORDER FUNCTION	Critical (10)	SOLVED - 02/15/2024
(HAL-05) MISSING VALIDATION OF VAULT ID IN SUBMIT FUNCTION	High (8.8)	SOLVED - 02/15/2024
(HAL-06) MISMATCHED WITHDRAWAL AND DEPOSIT ROUND IDS	High (7.5)	NOT APPLICABLE
(HAL-07) UNVERIFIED ORACLE PRICE	High (7.5)	RISK ACCEPTED
(HAL-08) REDUNDANT CHECK ON PRICE PER SHARE	Medium (5.0)	SOLVED - 02/15/2024
(HAL-09) MISSING DISABLEINITIALIZERS CALL IN CONTRACT CONSTRUCTORS	Medium (6.2)	SOLVED - 02/15/2024
(HAL-10) INCONSISTENT FEE CALCULATION	Low (3.1)	NOT APPLICABLE

(HAL-11) POTENTIAL MISALIGNMENT IN VAULT SELECTION	Low (3.1)	RISK ACCEPTED
(HAL-12) LACK OF VALIDATION FOR UNDERLYING ASSET	Low (2.5)	FUTURE RELEASE
(HAL-13) INCORRECT COLLATERALDIFF CALCULATION IN SYNC FUNCTION	Low (2.4)	SOLVED - 02/15/2024
(HAL-14) UNVERIFIED RETURN VALUES IN REFRESHCONFIGINTERNAL FUNCTION	Informational (1.9)	ACKNOWLEDGED
(HAL-15) INSUFFICIENT VALIDATION OF SHOCK PERCENTAGE	Informational (1.5)	ACKNOWLEDGED
(HAL-16) MISSING VALIDATION OF PARAMETERS	Informational (1.5)	FUTURE RELEASE
(HAL-17) SUBOPTIMAL HANDLING OF LONG POSITIONS	Informational (1.0)	NOT APPLICABLE
(HAL-18) UNUSED FUNCTION	Informational (0.5)	SOLVED - 02/15/2024
(HAL-19) UNUSED FUNCTION IN LPMANAGER CONTRACT	Informational (0.5)	SOLVED - 02/15/2024
(HAL-20) OVERESTIMATION OF EXECUTION FEE	Informational (0.5)	ACKNOWLEDGED
(HAL-21) INEFFECTIVE AFTER ORDER FROZEN FUNCTION	Informational (0.5)	SOLVED - 02/15/2024
(HAL-22) REDUNDANT ORACLE CALLS IN HEDGE FUNCTION	Informational (0.5)	SOLVED - 02/15/2024
(HAL-23) GAS INEFFICIENCIES IN ERROR HANDLING	Informational (0.5)	FUTURE RELEASE

FINDINGS & TECH DETAILS GAMMAPROTOCOL

4.1 (HAL-01) UNHANDLED ZERO PRICE IN MARGINCALCULATOR LEADING TO INVALID CALCULATIONS - HIGH (7.5)

Description:

In the MarginCalculator contract, the functions _convertAmountOnLivePrice and _convertAmountOnExpiryPrice exhibit a critical vulnerability due to the lack of handling for zero price values returned by the oracle. These functions are responsible for converting amounts between two assets based on their respective prices. The conversion is achieved by multiplying the amount of asset A by its price and dividing by the price of asset B.

The vulnerability arises when either priceA or priceB is zero. This situation could occur if the oracle fails to return a valid price for an asset. Notably, the _convertAmountOnExpiryPrice function attempts to handle cases where the expiry price is not set by using the current price, but it does not account for the possibility of the current price also being zero. A zero price leads to division by zero in the calculation, resulting in undefined behavior or transaction reversion. This can impact the integrity of the contract, as it may lead to invalid underlying and strike calculations, potentially causing a vault to be incorrectly deemed valid.

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (7.5)

Recommendation:

To mitigate this vulnerability, the following steps should be taken:

1. **Zero Price Validation**: Implement validation checks in both functions to ensure that neither priceA nor priceB is zero after retrieving values from the oracle. If a zero price is encountered, the function

should revert with a clear error message indicating the invalid price.

2. **Oracle Contract Review**: Although it's outside the scope of this audit, it's recommended to review the oracle contract to ensure it has appropriate mechanisms to prevent zero prices. This could include validation checks, reliable data sources, or fallback mechanisms in case of data unavailability.

3. **Fallback Mechanism**: Consider implementing a fallback mechanism in the MarginCalculator contract itself. For example, if a valid price cannot be obtained, the contract could use a default or historical price, or it could revert the transaction to prevent erroneous calculations.

4. **Comprehensive Testing**: Extensive testing should be conducted to ensure the contract behaves as expected in scenarios where the oracle returns zero prices. This includes unit tests and integration tests with the oracle contract.

5. **Documentation and Communication**: Update the contract documentation to clearly state the dependency on the oracle's price feed and the risks associated with zero prices. Communicate to users and stakeholders about these dependencies and potential risks.

Implementing these recommendations will enhance the robustness of the MarginCalculator contract and safeguard against the risks posed by zero price values.

Remediation Plan:

RISK ACCEPTED: Siren stated that "Code relies on oracle implementation to not return zero prices. ChainlinkPricer which is where the price comes from". This contract was out of scope.

4.2 (HAL-02) UNINITIALIZED SPOTSHOCKVALUE LEADING TO DIVISION BY ZERO - HIGH (7.5)

Description:

In the NakedMarginCalculator contract, specifically within the getNakedMarginRequired2 function, there exists a critical vulnerability due to the potential uninitialized state of spotShockValue. The function calculates various financial metrics based on the type of option (NMCI.OptionType) provided as an input. A key component in these calculations is spotShockValue, which is derived from spotShock[_productHash].

The vulnerability arises in scenarios where spotShockValue might not be initialized for a given _productHash. Particularly in the case of a NAKED_CALL option type, the code performs a division operation with spotShockValue. If spotShockValue is uninitialized and defaults to zero, this leads to a division by zero scenario. This could either cause transaction reversion or undefined behavior, depending on the Ethereum Virtual Machine's (EVM) handling of such cases. The impact is primarily on the integrity of the contract, as it can lead to incorrect financial calculations or failed transactions.

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (7.5)

Recommendation:

To mitigate this vulnerability, the contract should incorporate checks to ensure spotShockValue is properly initialized before proceeding with calculations. This can be achieved through the following steps:

1. **Validation Check**: Implement a check to validate that spotShock[_productHash] returns a non-zero value before it's used in calcu-

lations. If the value is zero, the function should revert with a clear error message indicating the uninitialized state.

2. **Initialization Routine**: Establish a routine to initialize spotShockValue for all relevant _productHash values. This could be part of the contract deployment process or a separate initialization function that must be called before the contract is used.

3. **Documentation and Error Messages**: Enhance documentation and error messages to guide users and developers about the necessity of initializing spotShockValue. Clear documentation can prevent misconfiguration and misuse of the contract.

4. **Testing**: Implement comprehensive unit and integration tests to simulate scenarios where spotShockValue might be uninitialized and ensure the contract behaves as expected.

By implementing these recommendations, the NakedMarginCalculator contract can be safeguarded against division by zero errors and ensure more robust and reliable financial computations.

Remediation Plan:

RISK ACCEPTED: Siren stated that “Initializing a spotShock value if part of the admin routine. Even if we add a non-zero check, there is still a possibility of an incorrect value being set. It is the admins responsibility to correctly configure all risk parameters”.

4.3 (HAL-03) UNINITIALIZED IMPLEMENTATION IN UPGRADEABLE CONTRACT - MEDIUM (6.2)

Description:

The Controller contract, which employs the Initializable contract, is designed to support upgradeability and initialization functions. In an upgradeable contract architecture, it's vital that the implementation contracts (like Controller) are never initialized on their own, to prevent direct interaction which could lead to security vulnerabilities. However, in the given scenario, the Controller contract's implementation does not mark itself as initialized during its deployment. This oversight means that the implementation contract could potentially be initialized independently. Such a situation opens up risks including phishing attacks or, in cases of other vulnerabilities, even contract destruction. The root cause of this issue is the absence of the _disableInitializers function in the used version of the Initializable contract, which would have automatically set the initialized flag to true, preventing further initializations.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:N/Y:N/R:N/S:U (6.2)

Recommendation:

To mitigate this vulnerability, it is crucial to ensure that the implementation contract (Controller) cannot be initialized after deployment. Since the latest version of Initializable that includes the _disableInitializers function is not used, a manual approach is needed. This involves explicitly setting the initialized flag to true in the implementation contract's constructor or initialization function. By doing this, any attempt to re-initialize the implementation contract will be rejected, closing the vulnerability. Additionally, it's recommended to upgrade to a newer version of the Initializable contract if possible, to benefit from improved security features and best practices. This preventive measure secures the contract against unintended initializations, thereby safeguarding against potential phishing or destructive actions.

Remediation Plan:

SOLVED: The issue was solved in commit 566f8b8

4.4 (HAL-04) UNNECESSARY MEMORY STORE UPDATES IN RUNACTIONS FUNCTION - INFORMATIONAL (0.0)

Description:

The runActions internal function in the smart contract contains a segment of code that updates the vaultUpdated, vaultId, and vaultOwner variables in each iteration of its loop. This code segment is designed to handle various actions on a vault. However, there's a redundancy in the way these variables are updated. The vaultId and vaultOwner values are being reassigned in every loop iteration, even when they haven't changed from the previous iteration. Since these values are expected to remain constant for all actions in a single operate call, repeatedly writing the same values into memory is unnecessary and leads to inefficiency in terms of gas usage.

In Solidity, and most programming languages, writing to memory/storage is more expensive in terms of computational resources (gas, in the case of Ethereum) than reading from it. Therefore, optimizing memory writes can lead to more efficient code execution, especially important in a blockchain context where every operation costs gas.

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

To optimize the runActions function, consider updating the vaultId and vaultOwner variables only once per function call or when they truly need to change. This can be achieved by checking if vaultUpdated is already true before updating these variables. If vaultUpdated is true, it implies that these variables have already been set for the current operation, and subsequent updates in the same function call are unnecessary.

Remediation Plan:

PENDING: Siren stated that “This has minimal gas impact. Will be fixed in future versions”.

4.5 (HAL-05) REDUNDANT PARAMETER IN ONLYAUTHORIZED MODIFIER - INFORMATIONAL (0.0)

Description:

The onlyAuthorized modifier in the provided smart contract is designed to restrict access to certain functions, ensuring they can be executed only by authorized users. Currently, the modifier is defined to accept two parameters: _sender (representing the address attempting to execute the function) and _accountOwner (representing the account owner's address). However, within the context of the smart contract, _sender is consistently passed as msg.sender. This pattern introduces a redundancy since msg.sender is globally accessible within the contract and does not need to be passed as a parameter.

The redundancy of the _sender parameter can lead to unnecessary complexity and potential misunderstandings about how the modifier should be used. Simplifying the modifier by removing the _sender parameter and directly using msg.sender in the _isAuthorized function call would make the code cleaner, less prone to errors, and more gas-efficient.

BVSS:

AO:S/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:U (0.0)

Recommendation:

To enhance code clarity and efficiency, it is recommended to refactor the onlyAuthorized modifier and the _isAuthorized function as follows:

1. **Modify onlyAuthorized Modifier**:
   Remove the _sender parameter from the onlyAuthorized modifier. Update the modifier to use msg.sender directly when calling _isAuthorized.

Listing 1

```
modifier onlyAuthorized(address _accountOwner) {
    _isAuthorized(msg.sender, _accountOwner);
    _;
}
```

2. **Refactor _isAuthorized Function**:
   The _isAuthorized function should remain unchanged, as it correctly takes two addresses to perform its check. The modification in the onlyAuthorized modifier ensures that msg.sender is always used as the first parameter.

3. **Update Function Calls**:
   Update all instances where the onlyAuthorized modifier is used in the contract. Remove the msg.sender argument from these calls, as it is now implicitly used within the modifier.

   Before:

   Listing 2

```
function someFunction(...) external onlyAuthorized(msg.
↳ sender, someAccountOwner) {
    // Function logic here
}
```

   After:

   Listing 3

```
function someFunction(...) external onlyAuthorized(
↳ someAccountOwner) {
    // Function logic here
}
```

These changes will make the contract's code more concise and reduce the chance of errors related to the misuse of the _sender parameter. Additionally, it could slightly reduce the gas cost associated with these function calls due to the decreased computational overhead.

Remediation Plan:

PENDING: Siren stated that “This has minimal gas impact. Will be fixed in future versions”.

FINDINGS & TECH DETAILS CORE-V4

5.1 (HAL-01) UNDERFLOW IN RESET PENDING ORDER - CRITICAL(10)

Description:

In the Gmx2Hedger contract, the resetPendingOrder function directly sets pendingOrdersCount to 0. This action can create a discrepancy between the actual number of pending orders and the pendingOrdersCount state variable. Subsequent order execution or cancellation, which decrement pendingOrdersCount, can lead to an underflow error, as the EVM's safety checks will catch pendingOrdersCount = pendingOrdersCount - 1; when pendingOrdersCount is already 0, but there are still pending orders.

This misalignment can result in:

1. **Failure of Order Processing**: Any unprocessed orders will fail to execute or cancel properly due to the underflow error, disrupting normal contract operations.
2. **Operational Risk**: The function might be misused or misunderstood as a way to pause operations or cancel transactions, leading to operational inconsistencies.
3. **Contract Robustness**: The direct manipulation of pendingOrdersCount without proper checks and balances undermines the contract's robustness and reliability.

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:C/D:N/Y:N/R:N/S:C (10)

Recommendation:

To address this issue:

1. **Revise Function Logic**: Consider revising the logic of resetPendingOrder. Instead of directly setting pendingOrdersCount to 0, ensure that it reflects the actual number of pending orders. One approach could be to decrement pendingOrdersCount only after successfully processing each pending order.

2. **Implement Safety Checks**: Include safety checks to prevent underflow and ensure that the state variable accurately represents the real count of pending orders.

3. **Reevaluate Function Necessity**: Assess if resetPendingOrder is necessary and used appropriately. If it's meant to serve as a pause or cancel functionality, implement it more explicitly with proper tracking and revert conditions.

4. **Document Usage and Limitations**: Clearly document the purpose and limitations of resetPendingOrder to avoid misuse or misinterpretation.

By implementing these recommendations, the contract can maintain accurate tracking of pending orders and enhance its overall reliability and integrity.

Remediation Plan:

SOLVED: The client did address the issue in commit 8b83e69.

5.2 (HAL-02) MISMATCHED RETURN VALUES - CRITICAL(10)

Description:

In the Gmx2Hedger contract, the _calculateAdjustedExecutionPrice function incorrectly expects four return values from the IGmxUtils .getExecutionPrice call, while the actual GMX function returns a ExecutionPriceResult struct with only three values. This mismatch results in the contract trying to access an undefined fourth return value, leading to a revert.

Recommendation:

Modify _calculateAdjustedExecutionPrice to correctly handle the ExecutionPriceResult struct returned by IGmxUtils.getExecutionPrice, ensuring that it properly extracts the executionPrice from the struct. This correction will align the function with the GMX interface and prevent unintended reverts.

Remediation Plan:

NOT APPLICABLE: The original contracts were used for the integrity check https://github.com/gmx-io/gmx-synthetics/blob/main/contracts/reader/ReaderPricingUtils.sol#L144. However, Siren is using a custom implementation that was out of scope and provided afterward which do return 4 values: https://arbiscan.io/address/0xb80E321fA8eCF53E354E72A254438eC6caB837eF#code

5.3 (HAL-03) UNSET SWAP PATH - CRITICAL(10)

Description:

In the Gmx2Hedger contract, the functions _createOrderParamAddresses and _orderParamAddresses define a swapPath variable but do not assign any value to it. This oversight can cause issues when these functions are used to create orders, especially if the GMX integration expects a properly set swapPath for order execution.

Not setting swapPath could lead to:

1. **Failed Order Execution**: Orders might fail or not execute as expected if the GMX system requires a valid swapPath for processing orders.
2. **Integration Inconsistencies**: Incomplete or incorrect implementation of the GMX protocol specifications can lead to inconsistencies and unexpected behavior in the contract's interaction with GMX.
3. **Testing and Reliability Concerns**: The lack of appropriate testing for external contract integrations, especially for critical functionalities like order creation, can undermine the contract's reliability and robustness.

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:C/D:N/Y:N/R:N/S:C (10)

Recommendation:

To resolve this issue:

1. **Define swapPath Appropriately**: Modify _createOrderParamAddresses and _orderParamAddresses to correctly initialize and set swapPath as required by the GMX protocol or the contract's design.

2. **Comprehensive Testing**: Implement thorough testing to validate the functionality of order creation and execution, particularly focusing on integration with the GMX system. Ensure that all aspects of the order, including swapPath, function as intended.

By addressing these recommendations, the Gmx2Hedger contract can ensure proper integration with the GMX system, enhance its operational effectiveness, and maintain the reliability and integrity of its functionalities.

Remediation Plan:

NOT APPLICABLE: Siren stated that “We use USDC as collateral, so no conversion is required and empty needed to be used as a parameter”.

5.4 (HAL-04) INEFFECTIVE ERROR HANDLING IN EXECUTE ORDER FUNCTION - CRITICAL(10)

Description:

In the TradeExecutor contract, the executeOrder function's try-catch block is designed to handle errors from the executeActions call. However, it only catches errors with string messages (catch Error(string memory reason)), not custom errors. This design oversight becomes problematic when executeActions uses custom errors, like CustomErrors.NotEnoughLiquidity. The EVM fails to decode the custom error's bytes4 signature into a string, causing the catch block to fail and not execute the intended completeOrderAndIssueRefund.

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:N/D:N/Y:N/R:N/S:U (10)

Recommendation:

To address this issue and improve error handling:

1. **Expand Catch Block**: Include an additional catch block to handle custom errors as bytes memory. This ensures that both string-based errors and custom errors are effectively caught and handled.

Listing 4

```
try this.executeActions(vars, poolAddress) {
    // Success logic...
} catch Error(string memory reason) {
    // Handle string error...
} catch (bytes memory _err) {
    // Handle custom error...
}

```

2. **Consistency in Error Usage**: Standardize the use of error handling throughout the contract. Decide on a consistent approach between custom errors and string-based errors. This consistency will simplify error handling and make the contract more maintainable.
3. **Thorough Testing**: Test the contract extensively to ensure that both types of errors are correctly caught and handled, especially in critical functions like executeOrder.

By implementing these recommendations, the contract can robustly handle errors in executeOrder, ensuring that it responds appropriately to all error types and maintains its intended functionality even in error scenarios.

Remediation Plan:

SOLVED: The issue was solved in commit 961e4b3

5.5 (HAL-05) MISSING VALIDATION OF VAULT ID IN SUBMIT FUNCTION - HIGH (8.8)

Description:

In the TradeExecutor contract, the submitOrder function lacks a critical validation check for the traderVaultId. This parameter is crucial for identifying the specific vault associated with a trader's order. Without validating traderVaultId against the vault counter from the controller (controller.getAccountVaultCounter(_owner)), there's a risk of accepting orders linked to invalid or non-existent vault IDs.

Potential consequences of this oversight include:

1. **Non-Executable Orders**: Orders associated with invalid traderVaultIds might be recorded in the system but would be unexecutable. This could clutter the system with unusable orders, impacting its efficiency and usability.
2. **Operational Disruption**: The inability to execute orders due to invalid vault IDs could disrupt trading operations, leading to trader dissatisfaction and potential loss of trust in the platform.
3. **Dependency on Controller's Security**: Relying solely on the controller for vault ID verification later in the process puts undue pressure on the controller's security mechanisms. It's more effective and safer to catch such issues early in the submitOrder function itself.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:H/D:N/Y:N/R:N/S:U (8.8)

Recommendation:

To mitigate these risks:

- Implement an early validation check in submitOrder to ensure that the provided traderVaultId is valid. This should involve comparing it against controller.getAccountVaultCounter(_owner) to confirm its existence and correct association with the owner.
- By adding this validation step, the system can promptly reject orders with invalid vault IDs, preventing the accumulation of non-executable orders and maintaining operational efficiency and reliability.

Remediation Plan:

SOLVED: The issue was solved in commit b9ffe4e

5.6 (HAL-06) MISMATCHED WITHDRAWAL AND DEPOSIT ROUND IDS - HIGH (7.5)

Description:

The getCashLocked function in the LpManager contract calculates the total amount of cash locked in a pool by summing values from the current withdrawal and deposit rounds. However, the function does not verify that the withdrawal round ID and deposit round ID correspond to the same round. This oversight could lead to an inaccurate calculation of locked cash, as it might combine values from different rounds.

Combining values from mismatched rounds can result in:

1. **Inaccurate Liquidity Calculations**: If the withdrawal and deposit rounds are not synchronized, the liquidity calculation may either overstate or understate the actual locked cash, affecting operational decisions and risk assessments.
2. **Potential Financial Implications**: Miscalculating locked cash can impact the pool's financial health, affecting decisions related to withdrawals, deposits, and collateral requirements.

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (7.5)

Recommendation:

To ensure accurate calculations, the getCashLocked function should be updated to include a validation check confirming that the withdrawal round ID and deposit round ID are for the same round. If a mismatch is detected, the function should either revert or handle the discrepancy appropriately to prevent incorrect calculations. This validation will enhance the accuracy and reliability of the liquidity management in the LpManager contract.

Remediation Plan:

NOT APPLICABLE: Siren stated that “By design. The duration of a deposit round can be different from the duration of a withdrawal round, so we do not have to track round ids”

5.7 (HAL-07) UNVERIFIED ORACLE PRICE - HIGH (7.5)

Description:

The getCashBuffer function in the HedgedPool contract calculates the cash buffer required for the pool based on the notional exposures and the underlying asset prices obtained from an oracle. However, this function does not verify if the oracle price is non-zero before using it in calculations. Relying on potentially zero oracle prices can lead to incorrect calculation of the cash buffer, which is critical for maintaining adequate collateralization in the pool.

A zero price from the oracle can result in:

1. **Underestimation of Cash Buffer**: If the oracle returns a zero price, the calculated cash buffer for the respective exposures (calls or puts) might be inaccurately low or zero. This can cause the contract to underestimate the amount of collateral required, leading to under-collateralization.
2. **Operational and Financial Risk**: Inaccurate cash buffer calculation poses significant operational and financial risks. It can affect the contract's ability to meet its financial obligations, potentially leading to solvency issues.
3. **Contract Integrity**: Reliable and accurate collateral management is crucial for the contract's integrity and trustworthiness. Inaccuracies in collateral calculation can undermine user confidence and the contract's overall reliability.

BVSS:

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (7.5)

Recommendation:

To mitigate these risks, consider the following actions:

1. **Implement Price Validation Checks**: Modify the getCashBuffer function to include validation checks for non-zero oracle prices. If a zero price is returned, the function should handle this case appropriately, potentially by reverting the transaction with a clear error message.
2. **Fallback Mechanism for Oracle Prices**: Implement a fallback mechanism for cases where the oracle fails to provide a valid price. This could involve using a historical price or an average of recent prices as a temporary measure until the oracle issue is resolved.
3. **Enhanced Oracle Reliability**: Review the oracle implementation to ensure its reliability and accuracy in providing asset prices. Consider using multiple oracles or a decentralized oracle network to reduce the risk of incorrect price feeds.

By addressing these recommendations, the HedgedPool contract can enhance its collateral management accuracy and reliability, safeguarding against the risks posed by potential oracle inaccuracies.

Remediation Plan:

RISK ACCEPTED: Siren stated that “Code relies on oracle implementation to not return zero prices. ChainlinkPricer which is where the price comes from”. This contract was out of scope.

5.8 (HAL-08) REDUNDANT CHECK ON PRICE PER SHARE - MEDIUM (5.0)

Description:

In the LpManager contract, the function getDepositStatus includes a conditional check for pricePerShare > 0 when calculating sharesRedeemable for a user's deposit. This check is redundant, considering the contract's design should not allow a round to advance without a non-zero pricePerShare. However, this additional validation introduces a risk:

- If any part of the contract fails to validate pricePerShare adequately (allowing a round to advance with a zero pricePerShare), deposits from that round may not be correctly processed in redeemShares .
- Specifically, if pricePerShare is zero, cashPending would be 0, leading to a reset of the deposit's roundId and amount in redeemShares . This reset could erase the record of previous deposits, resulting in a potential loss of funds for the users.

This situation could occur if a user's last deposit round ID does not match the current depositRoundId, and the pricePerShare for that previous round was improperly set to zero.

POC:

Listing 5

```
    function test_invalid_redeem_deposit() public {

        stdstore.target(address(lpManager))
            .sig("depositRoundId(address)")
            .with_key(address(hedgedPool))
            .checked_write(1);


        stdstore.target(address(lpManager))
```

```
                .sig("deposits(address,address)")
                .with_key(address(hedgedPool))
                .with_key(address(ADMIN))
                .depth(1)  // amount (roundId is 0)
                .checked_write(100);

            stdstore.target(address(lpManager))
                .sig("deposits(address,address)")
                .with_key(address(hedgedPool))
                .with_key(address(ADMIN))
                .depth(2)  // unredeemedShares (roundId is 0)
                .checked_write(100);

            // To make things easier, we will be calling lpmanager
↳ directly as
            // if the pool did call it.
            vm.prank(address(hedgedPool));
            lpManager.redeemShares(address(ADMIN));

            // This will set the previous round pricePerShare to 2e8
            //
            stdstore.target(address(lpManager))
                .sig("depositRounds(address,uint256)")
                .with_key(address(hedgedPool))
                .with_key(uint256(0))
                .depth(1) // pricePerShare
                .checked_write(2e8);

            (uint256 cashPending, uint256 sharesRedeemable) =
↳ lpManager.getDepositStatus(address(hedgedPool), ADMIN);

            console.log("====== INVALID REDEEM DEPOSIT ======");
            console.log("cashPending", cashPending);
            console.log("sharesRedeemable", sharesRedeemable);
        }
```

POC output:

Listing 6

```
====== INVALID REDEEM DEPOSIT ======
redeemed 100 // This should revert, as no pricePerShare was set
↳ for that round
```

```
cashPending 0
sharesRedeemable 0
```

Expected output if price for that round is not zero (2e8):

Listing 7

```
====== INVALID REDEEM DEPOSIT ======
redeemed 150
cashPending 0
sharesRedeemable 0
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (5.0)

Recommendation:

To address this issue, the following steps should be considered:

1. **Remove Redundant Check**: Consider removing the pricePerShare > 0 check in getDepositStatus if it's guaranteed by the contract's design that pricePerShare cannot be zero when a new round starts.
2. **Consistent Validation Across the Contract**: Ensure that all sections of the contract that handle pricePerShare do so consistently and prevent rounds from advancing with a zero pricePerShare.
3. **Error Handling and Fallback**: Implement appropriate error handling for cases where pricePerShare might be zero. This can prevent the unintended reset of deposit records.

By implementing these recommendations, the risk of deposit loss in the LpManager contract due to the redundant pricePerShare check can be significantly mitigated, thereby enhancing the contract's overall security and reliability.

Remediation Plan:

SOLVED: The issue was solved in commit d2659c7

5.9 (HAL-09) MISSING DISABLEINITIALIZERS CALL IN CONTRACT CONSTRUCTORS - MEDIUM (6.2)

Description:

The absence of _disableInitializers in the constructors of the LpManager, HedgedPool, Gmx2Hedger, Gmx1Hedger, TradeExecutor, and Perennial1Hedger contracts presents significant security risks. This function is crucial in Solidity contracts, especially those following the upgradeable pattern, to prevent the initialization of the implementation contract directly.

Without calling _disableInitializers, the implementation contracts remain vulnerable to malicious initialization. This can include setting adverse states or manipulating contract logic. The risks associated with this vulnerability are multifold:

1. **Phishing Attacks**: Attackers could trick contract administrators or users into interacting with the implementation contract directly instead of the proxy. This could lead to transactions that have unintended consequences, such as transferring funds or altering critical contract states.
2. **Self-Destruction Risks**: If any of the contracts contain a selfdestruct function or similar, and it’s callable after initialization, the contract could be destroyed by an attacker. This would result in the loss of code and state, potentially affecting all dependent systems or contracts.

BVSS:

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:N/Y:N/R:N/S:U (6.2)

Recommendation:

To mitigate these risks, it’s imperative to include _disableInitializers in the constructors of all mentioned contracts. This will prevent direct initialization of the implementation contracts, safeguarding them against the aforementioned risks.

Remediation Plan:

SOLVED: The issue was solved in commit cf7a6d4

5.10 (HAL-10) INCONSISTENT FEE CALCULATION - LOW (3.1)

Description:

In the TradeExecutor contract, there's an inconsistency in how execution fees are calculated and handled between the submitOrder and completeOrderAndIssueRefund functions. submitOrder uses a static value minExecutionFee to determine the required fees, while completeOrderAndIssueRefund calculates the transaction cost dynamically based on the number of order.legs and other factors. This discrepancy can lead to scenarios where the actual transaction costs exceed the statically set minExecutionFee, potentially causing the system to incur excess costs, especially for orders with a large number of legs.

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:L/R:N/S:U (3.1)

Recommendation:

To address this inconsistency and potential financial risk:

- Revise the fee calculation in submitOrder to dynamically estimate the execution fee based on factors such as the number of order legs and other relevant parameters. This approach should align with the dynamic calculation used in completeOrderAndIssueRefund.
- Ensure that the dynamically calculated fee in submitOrder includes a buffer to account for gas price fluctuations and other variables, safeguarding against underestimation.
- Regularly review and adjust the fee calculation logic as necessary to reflect changes in network conditions and contract usage patterns.

By implementing dynamic and more accurate fee calculations in submitOrder

, the contract can better manage its financial risks and ensure that execution fees cover the actual costs incurred during order processing.

Remediation Plan:

NOT APPLICABLE: Siren stated that “This works how it was designed. in submitOrder user sends some predefined amount which is much larger than the potential tx fee. And executeOrder calculates how much tx actually costs and returns the difference.”

5.11 (HAL-11) POTENTIAL MISALIGNMENT IN VAULT SELECTION - LOW (3.1)

Description:

In the TradeExecutor contract, the executeOrder function currently selects the margin vault for a pool using the first vault ID (index 0) associated with an underlying asset (marginVaults[poolAddress][vars.order.underlying][0]). However, since openMarginVault allows for the creation of multiple vaults for the same _underlyingAsset, this approach might not always reference the most relevant or recent vault, potentially leading to operational inconsistencies or inefficiencies.

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:N/Y:N/R:N/S:U (3.1)

Recommendation:

To better align the vault selection with the potential multiple vaults scenario:

1. **Use the Most Recent Vault**: Modify executeOrder to select the latest vault created for a given underlying asset. This can be achieved by accessing the last element in the marginVaults[poolAddress][vars.order.underlying] array, which represents the most recently opened vault.
2. **Review Vault Management Logic**: Assess the overall logic for managing multiple vaults per underlying asset. Ensure that the contract's design and functions are consistent with the intended use and management of these vaults.
3. **Update Documentation**: Reflect these changes and the rationale behind

them in the contract’s documentation, so that maintainers and users understand the vault selection process.

4. **Test for Edge Cases**: Thoroughly test the updated functionality to ensure it handles scenarios involving multiple vaults per underlying asset correctly.

By selecting the most recent vault for order execution, the TradeExecutor contract can better accommodate the dynamics of multiple vault scenarios, enhancing its operational effectiveness and reliability.

Remediation Plan:

RISK ACCEPTED: Siren stated that “Right now it is by design that there is only one for the pool”

5.12 (HAL-12) LACK OF VALIDATION FOR UNDERLYING ASSET - LOW (2.5)

Description:

In the TradeExecutor contract, the openMarginVault function does not validate the _underlyingAsset parameter. It neither checks if _underlyingAsset is a valid/whitelisted address nor if it is a non-zero value. This oversight could lead to operational issues or vulnerabilities. Although the check is latter performed on validateExecuteOrderArguments it would be a good idea to do it on early stages.

BVSS:

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (2.5)

Recommendation:

Implement validation checks in openMarginVault to ensure _underlyingAsset is neither zero nor an invalid address. This may involve checking against a list of allowed underlying assets or ensuring that _underlyingAsset is a non-zero address. This validation will enhance the function's robustness and prevent potential misuse or errors.

Remediation Plan:

PENDING: Siren stated that “This is low priority and will be fixed in future versions”

5.13 (HAL-13) INCORRECT COLLATERALDIFF CALCULATION IN SYNC FUNCTION - LOW (2.4)

Description:

In the Gmx2Hedger contract, the _sync function calculates collateralDiff , the difference in collateral required after hedging operations. This function calls _depositCollateral and _withdrawCollateral, adjusting the total collateralDiff based on the amounts processed. However, there's a discrepancy in how _depositCollateral handles the deposit amount.

The _depositCollateral function caps the deposit amount to the pool's balance if the requested amount exceeds the available balance. But the collateralDiff in _sync still uses the original, uncapped amount for its calculations, leading to an inaccurate representation of the actual collateral difference.

This discrepancy can result in:

1. **Inaccurate Collateral Tracking**: The contract might track more or less collateral than what is actually deposited or withdrawn, leading to potential financial discrepancies.
2. **Operational Risk**: Misrepresenting the actual collateral could impact the contract's operational decisions, such as future hedging actions or collateral requirements.

BVSS:

AO:S/AC:L/AX:L/C:N/I:C/A:H/D:N/Y:N/R:N/S:U (2.4)

Recommendation:

To resolve this issue, _depositCollateral should be modified to return the actual deposited amount. The _sync function should then use this returned value to update collateralDiff accurately. This change ensures that collateralDiff reflects the true state of the contract's collateral balance after each hedging operation, maintaining financial accuracy and integrity.

Remediation Plan:

SOLVED: The issue was solved in commit 443f180.

5.14 (HAL-14) UNVERIFIED RETURN VALUES IN REFRESHCONFIGINTERNAL FUNCTION - INFORMATIONAL (1.9)

Description:

In the HedgedPool contract, the function _refreshConfigInternal is responsible for updating and initializing key addresses used by the contract, such as the controller, calculator, oracle, and others. This function retrieves these addresses from the addressBook contract. However, the function does not currently verify whether the returned addresses are non-zero. Using zero addresses in these key functionalities could lead to unexpected behaviors and vulnerabilities in the contract.

The absence of validation checks for zero addresses can lead to several critical issues:

1. **Function Calls to Zero Addresses**: If any of these addresses are zero, function calls to these addresses will fail, potentially leading to transaction reversion or unexpected contract behavior.
2. **Security Risks**: Assigning a zero address to key functionalities like the controller or oracle can expose the contract to various security vulnerabilities. It might allow attackers to manipulate contract state or execute functions that shouldn't be accessible.
3. **Operational Failure**: Essential operations relying on these addresses would fail, rendering the contract non-functional. This could affect critical processes like margin calculations, trade executions, or fee collections.
4. **Loss of Funds**: In the worst-case scenario, such as if the margin pool address is zero, any funds sent to this address would be irretrievably lost.

BVSS:

AO:S/AC:L/AX:L/C:N/I:H/A:H/D:N/Y:N/R:N/S:U (1.9)

Recommendation:

To mitigate these risks, the following steps should be recommended:

1. **Validation Checks**: Implement validation checks in the _refreshConfigInternal function to ensure that all addresses retrieved from the addressBook are non-zero. If a zero address is detected, the function should revert the transaction with a clear error message.
2. **Robust Address Management**: Enhance the addressBook contract, if possible, to include safeguards that prevent the registration of zero addresses for key functionalities.

By implementing these recommendations, the HedgedPool contract can significantly enhance its operational reliability and security, protecting itself from risks associated with zero addresses.

Remediation Plan:

ACKNOWLEDGED: Siren stated that “This is done in deployment scripts”.

5.15 (HAL-15) INSUFFICIENT VALIDATION OF SHOCK PERCENTAGE - INFORMATIONAL (1.5)

Description:

In the HedgedPool contract's configUnderlying function, there is a lack of validation for the input parameters _spotShockPercentCalls and _spotShockPercentPuts. These parameters are crucial as they determine the percentage increase in the cash buffer calculated in the getCashBuffer function based on market volatility and potential price movements of the underlying assets. However, without validation ensuring that these values are greater than or equal to 100 and less than or equal to 100, respectively, the cash buffer might be under-calculated, leading to insufficient collateralization.

If _spotShockPercentCalls or _spotShockPercentPuts are set to invalid values, it implies an underestimation of potential price volatility. This could result in a cash buffer that is too small to cover the actual risk, posing a significant risk to the contract's financial stability and the security of its users' assets.

BVSS:

AO:S/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (1.5)

Recommendation:

To mitigate this risk and ensure the integrity of the cash buffer calculation, consider modifying the configUnderlying function to include validation checks for _spotShockPercentCalls and _spotShockPercentPuts, ensuring that the former is greater than or equal to 100 and less than or equal for the latter. If the values are not valid, the function should revert the transaction with an appropriate error message.

Remediation Plan:

ACKNOWLEDGED: Siren stated that “This is done in deployment scripts”.

5.16 (HAL-16) MISSING VALIDATION OF PARAMETERS - INFORMATIONAL (1.5)

Description:

In the HedgedPool contract, the processOToken function adds oTokens to the pool but does not validate whether the provided oToken actually corresponds to the specified underlying asset and expiry timestamp. While the oToken is created with these parameters by the trade executor, verifying these details in processOToken is crucial for ensuring the integrity and consistency of the contract's operations.

Lack of this validation poses risks:

1. **Incorrect oToken Association**: Without validation, there's a risk of associating an oToken with an incorrect underlying asset or expiry, leading to potential mismatches and operational issues.
2. **Contract Integrity and Reliability**: Ensuring that each oToken correctly represents its underlying asset and expiry is vital for the contract's reliability and the trust of its users.

BVSS:

AO:S/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (1.5)

Recommendation:

To enhance security and consistency, processOToken should be updated to include validation checks confirming that the provided oToken matches the specified underlying asset and expiry timestamp. This can be done by querying the oToken's properties and comparing them with the function's parameters. If a mismatch is detected, the function should revert to prevent incorrect oToken processing. This validation will strengthen the contract's integrity and ensure accurate tracking and handling of oTokens.

Remediation Plan:

PENDING: Siren stated that “This is low priority and will be fixed in future versions”.

5.17 (HAL-17) SUBOPTIMAL HANDLING OF LONG POSITIONS - INFORMATIONAL (1.0)

Description:

In the Controller contract's mintOtoken function, there is a potentially suboptimal handling of long positions when creating new short positions. Currently, the function transfers long oTokens to the user and then creates a new short position. However, this approach might not be the most efficient, especially when there are existing long positions that could offset the newly minted short positions.

BVSS:

AO:S/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:U (1.0)

Recommendation:

To optimize the process:

1. **Burn Long Offsets**: Instead of transferring long oTokens to the user, consider directly burning the long positions that can offset the new shorts. This would streamline the process, reducing unnecessary token transfers and simplifying position management.
2. **Reassess Position Management**: Review the overall logic for handling long and short positions in the mintOtoken function. Ensure that the process is efficient, transparent, and aligns with the contract's intended functionality.
3. **Implement Conditional Logic**: Introduce conditional logic to handle different scenarios, such as when there are sufficient long positions to offset the shorts versus when additional short positions need to be created.
4. **Test for Impact**: Thoroughly test the revised implementation to assess its impact on gas costs, transaction efficiency, and overall

contract behavior.

By adopting a more streamlined approach for managing long and short positions in the mintOtoken function, the Controller contract can enhance efficiency, reduce transaction costs, and simplify the user experience.

Remediation Plan:

NOT APPLICABLE: Siren stated that “The result of mintOToken is a short position added to MarginVault and a corresponding long oToken balance transferred to the user. Because it is possible that the vault already contains a long position, we first transfer it to the user and then mint and transfer the remainder. This is done to avoid simultaneously having long and short position for the same oToken in the vault”.

5.18 (HAL-18) UNUSED FUNCTION - INFORMATIONAL (0.5)

Description:

In the HedgedPool contract, the function setQuoteProvider is present but appears not to be used in any integration with other contracts. This function allows the contract owner to set or unset permissions for quote provider addresses. The presence of unused functions in a smart contract, especially those related to permissions or roles, could potentially lead to confusion, increase the surface for potential bugs, and complicate contract maintenance and audits.

Having an unused function doesn't directly pose a security risk, but it does have implications:

1. **Maintenance Complexity**: Unused functions can make the contract code more complex than necessary, increasing the difficulty of maintenance and understanding of the contract's logic.
2. **Potential for Future Errors**: While currently unused, future modifications or extensions of the contract could inadvertently interact with this function, leading to unintended consequences.
3. **Audit Clarity**: During security audits, the presence of unused functions can distract or confuse, potentially leading to oversight of more critical areas of the contract.

BVSS:

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.5)

Recommendation:

To address this issue, consider the following actions:

1. **Remove Unused Function**: If setQuoteProvider is indeed not required for the contract's current and foreseeable functionality, removing it could simplify the contract and reduce potential confusion.
2. **Assess Future Necessity**: Before removal, assess if the function might be needed in future expansions or updates to the contract. If there's a potential use case on the horizon, consider keeping the function, but clearly document its intended use.

Remediation Plan:

SOLVED: The issue was solved in commit 9a6bd4b.

5.19 (HAL-19) UNUSED FUNCTION IN LPMANAGER CONTRACT - INFORMATIONAL (0.5)

Description:

The addPricedCash function in the LpManager contract is present, but not utilized in any contract integrations. While this doesn't pose a direct security risk, it adds unnecessary complexity to the contract, potentially leading to confusion and challenges in maintenance and auditing.

BVSS:

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.5)

Recommendation:

To address this, it's advisable to remove the addPricedCash function if it's confirmed to be redundant for the contract's current and future operations. This simplification will enhance the contract's clarity and maintainability. Additionally, ensure that this change is documented and communicated clearly to all stakeholders. Regular code reviews and refactoring should be a part of ongoing contract maintenance to keep the codebase efficient and relevant.

Remediation Plan:

SOLVED: The issue was solved in commit 57c4a92.

5.20 (HAL-20) OVERESTIMATION OF EXECUTION FEE - INFORMATIONAL (0.5)

Description:

In the Gmx2Hedger contract, the _calculateExecutionFee function uses a fixed ORDER_GAS_LIMIT of 7e6 for estimating the execution fee. This approach may lead to a significant overestimation of the required fee. As per GMX documentation, the execution fee should be calculated using tx.gasprice * GasUtils.adjustGasLimitForEstimate(datastore, estimatedGasLimit), where estimatedGasLimit varies based on the operation (deposits, orders, withdrawals).

Overestimating the execution fee can result in:

1. **Inefficient Gas Usage**: Users may end up paying more in transaction fees than necessary, leading to inefficiency and higher operational costs.
2. **Contract Reliability**: Relying on an overestimated gas limit could affect the contract's reliability and user trust, especially if users consistently notice higher than expected transaction costs.

BVSS:

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.5)

Recommendation:

To align with GMX documentation and optimize gas usage, the _calculateExecutionFee function should be updated to dynamically calculate the execution fee based on the specific operation being executed. This involves using GasUtils.estimateExecuteDepositGasLimit, GasUtils.estimateExecuteOrderGasLimit, or GasUtils.estimateExecuteWithdrawalGasLimit as appropriate for the given context. Implementing this change ensures more accurate and efficient execution fee calculations, enhancing the contract’s overall efficiency and user experience.

Remediation Plan:

ACKNOWLEDGED: Siren stated that “This function is called only by the bot and users never interact with it. Also, GMX does a refund: . So sending a predefined amount instead of estimating will also save gas”

5.21 (HAL-21) INEFFECTIVE AFTER ORDER FROZEN FUNCTION - INFORMATIONAL (0.5)

Description:

The afterOrderFrozen function in Gmx2Hedger lacks implementation. Given the contract's design to use only market orders, this function is not expected to be triggered. However, its current form without a revert statement could lead to confusion or unintended execution.

BVSS:

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.5)

Recommendation:

Introduce a revert statement in afterOrderFrozen to prevent any potential misuse or unexpected behavior, ensuring the function aligns with the intended design of using only market orders. This change will clarify the function's purpose and safeguard the contract against unforeseen changes in the GMX system.

Remediation Plan:

SOLVED: The issue was solved in commit 71d6030.

5.22 (HAL-22) REDUNDANT ORACLE CALLS IN HEDGE FUNCTION - INFORMATIONAL (0.5)

Description:

In the hedge function of the Gmx2Hedger contract, the IGmxOracle(gmxOralce).getStablePrice is called twice for the same value. This redundancy might lead to unnecessary gas usage, as the Solidity compiler may not optimize by reusing the result of the first call.

BVSS:

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.5)

Recommendation:

Optimize the function by storing the result of the IGmxOracle(gmxOralce).getStablePrice call in a local variable and reusing it. This change will reduce gas costs and enhance the contract's efficiency.

Remediation Plan:

SOLVED: The issue was solved in commit c40f131.

5.23 (HAL-23) GAS INEFFICIENCIES IN ERROR HANDLING - INFORMATIONAL (0.5)

Description:

In both the core-v4 and GammaProtocol repositories, there is a notable inefficiency in error handling due to the use of string-based errors. Examples of this issue can be seen in the core-v4 repository's Gmx2Hedger , TradeExecutor, and LpManager contracts, as well as across various GammaProtocol contracts. String-based errors, while clear, are less gas-efficient than custom errors. In Solidity, a string error, regardless of its length, occupies a full 32-byte memory slot, leading to higher gas consumption.

BVSS:

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.5)

Recommendation:

To address this inefficiency:

1. **Adopt Custom Errors**: Transition from string-based errors to custom errors in both core-v4 and GammaProtocol repositories. Custom errors utilize a bytes4 error signature and are significantly more gas-efficient.
2. **Maintain Error Clarity**: Ensure that the new custom errors are adequately descriptive, maintaining the same level of clarity as the original string messages.

Implementing custom errors in the core-v4 and GammaProtocol repositories will result in substantial gas savings, improving the overall efficiency and cost-effectiveness of these contracts without compromising the clarity of error messages.

Remediation Plan:

”PENDING: Siren stated that “This is low priority and will be fixed in future versions”

REVIEW NOTES

6.1 GammaProtocol

Controller.sol:

Function	Visibility	Mutability	Modifiers
_isNotPartiallyPaused	Internal	N	-
_isNotFullyPaused	Internal	N	-
_isAuthorized	Internal	N	-
initialize	External	Y	initializer
donate	External	Y	NO
setSystemPartiallyPaused	External	Y	onlyPartialPauser
setSystemFullyPaused	External	Y	onlyFullPauser
setFullPauser	External	Y	onlyOwner
setPartialPauser	External	Y	onlyOwner
setCallRestriction	External	Y	onlyOwner
setOperator	External	Y	NO
refreshConfiguration	External	Y	onlyOwner
cleanTmpVault	External	Y	onlyOwner
_cleanTmpVault	Internal	Y	-
operate	External	Y	nonReentrant notFullyPaused onlyAccessKey
sync	External	Y	nonReentrant notFullyPaused
isOperator	External	N	NO
getConfiguration	External	N	NO
getProceed	External	N	NO
isLiquidatable	External	N	NO
getPayout	Public	N	NO

isSettlementAllowed	External	N	NO
canSettleAssets	External	N	NO
getAccountVaultCounter	External	N	NO
hasExpired	External	N	NO
getVault	External	N	NO
getVaultExtended	External	N	NO
getVaultExposure	External	N	NO
getVaultWithDetails	Public	N	NO
getOtokenIndex	Public	N	NO
_runActions	Internal	Y	-
_verifyFinalState	Internal	N	-
_openVault	Internal	Y	notPartiallyPaused onlyAuthorized
_depositLong	Internal	Y	notPartiallyPaused onlyAuthorized
_withdrawLong	Internal	Y	notPartiallyPaused onlyAuthorized
_depositCollateral	Internal	Y	notPartiallyPaused onlyAuthorized
_- withdrawCollateral	Internal	Y	notPartiallyPaused onlyAuthorized
_mintOtoken	Internal	Y	notPartiallyPaused onlyAuthorized
_burnOtoken	Internal	Y	notPartiallyPaused onlyAuthorized
_redeem	Internal	Y	-
_settleVault	Internal	Y	onlyAuthorized
_liquidate	Internal	Y	notPartiallyPaused
_call	Internal	Y	notPartiallyPaused onlyWhitelistedCallee
_checkVaultId	Internal	N	-
_- isCalleeWhitelisted	Internal	N	-

_-refreshConfigInternal	Internal	Y	-
_getTmpVault	Internal	N	-
setLock	External	Y	onlyOwner

General Overview of Controller Contract

- **Purpose**: Manages interactions in the Gamma Protocol, controlling how users, through their vaults, can mint, redeem, and settle options, among other actions.
- **Inheritance**: Inherits from OwnableUpgradeSafe, ReentrancyGuardUpgradeSafe , and Initializable, indicating upgradeable contract functionality with ownership control and reentrancy protection.

Key Functions and Features

- **Contract State Management**:
  - Ability to pause/unpause the system either partially or fully, affecting the availability of functions.
  - Control over whether only whitelisted callees can be used for specific contract calls.
- **Vault Management**:
  - Opening new vaults, depositing and withdrawing collateral, and managing positions in vaults.
  - Tracking of individual vaults per account, including a count of vaults per owner.
- **Action Execution**:
  - Function operate to execute multiple actions in a single transaction.
  - Supported actions include opening vaults, depositing/withdrawing collateral, minting/burning options, and more.

- **Access and Authorization**:
  - Functions to set and manage account operators.
  - Permissions for full and partial pausers to pause certain aspects of the system.
- **Configurations and Address Management**:
  - Functions to refresh and retrieve configurations, like updating the address book, oracle, calculator, and margin pool addresses.
  - Ability to update pauser roles and restrict callable functions.
- **Margin and Liquidation**:
  - Functions to check vault collateralization and perform liquidations.
  - Calculations for payouts, settlements, and determining if vaults are undercollateralized.
- **Utility Functions**:
  - Functions to donate assets to the pool, sync vault state, and check if an oToken has expired.
  - Retrieval of detailed vault information, including exposures and extended vault data.
- **Lock Mechanism**:
  - Feature to lock the controller, potentially restricting actions to holders of a specific NFT.

Error Codes

- Uses a series of error codes (C1 to C42) for different failure states, improving readability and debugging.

Considerations and Potential Risks

- **Complex Interactions**: The contract handles a wide range of actions and states, increasing the complexity of interactions and potential for bugs or unintended consequences.
- **Upgradeable Contract**: As an upgradeable contract, it's essential to manage upgrades carefully to ensure compatibility and avoid introducing vulnerabilities.
- **Dependency on External Contracts**: Relies on external contracts (e.g., Oracle, Calculator, Pool) for key functionalities, making it crucial to ensure these dependencies are secure and reliable.
- **System Pause Controls**: The ability to pause the system, either partially or fully, is powerful and must be managed judiciously to avoid misuse or unintended impact on users.
- **Access Control**: The use of pauser roles and operator permissions adds layers of access control, requiring careful management to maintain system integrity.
- **Liquidation Logic**: The liquidation functions need to be robust against market an

In-Function details

- The onlyWhitelistedCallee modifier does only trigger the whitelist check (_isCalleeWhitelisted) if callRestricted is set. Used internally on the _call function to prevent un-whitelisted callee addresses.
- The onlyAuthorized modifier does call the _isAuthorized function, which verifies if the _accountOwner is the sender or the operator was authorised by the sender via the setOperator function.
- The notPartiallyPaused modifier will verify if the systemPartiallyPaused is not set. This modifier is being used on internal functions to prevent partial actions. All external functions should first check for the full pause to be consistent. **As the comments, only redeem and settleVault is allowed.**
- The notFullyPaused modifier is used only on external functions such as operate and sync.

- The onlyAccessKey modifier will check when the isLocked flag is set for balanceOf on an IERC1155 token. This means that the operate function will only be callable if the sender holds a valid token.
- The initialize function is correctly implementing the initializer modifier, preventing re-calls.
- The donate function does transfer the desired token/amount to from the sender to the MarginPool increasing the assetBalance.
- setSystemPartiallyPaused and setSystemFullyPaused are limited to respective callers to set the respective state on the system. Preventing functions calls. The owner can change those addresses using setFullPauser and setPartialPauser.
- The setOperator allows the caller to set an alternative address as allowed to act on behalf of him.
- The _runActions function does accept a list of actions formatted using the following struct:

Listing 8

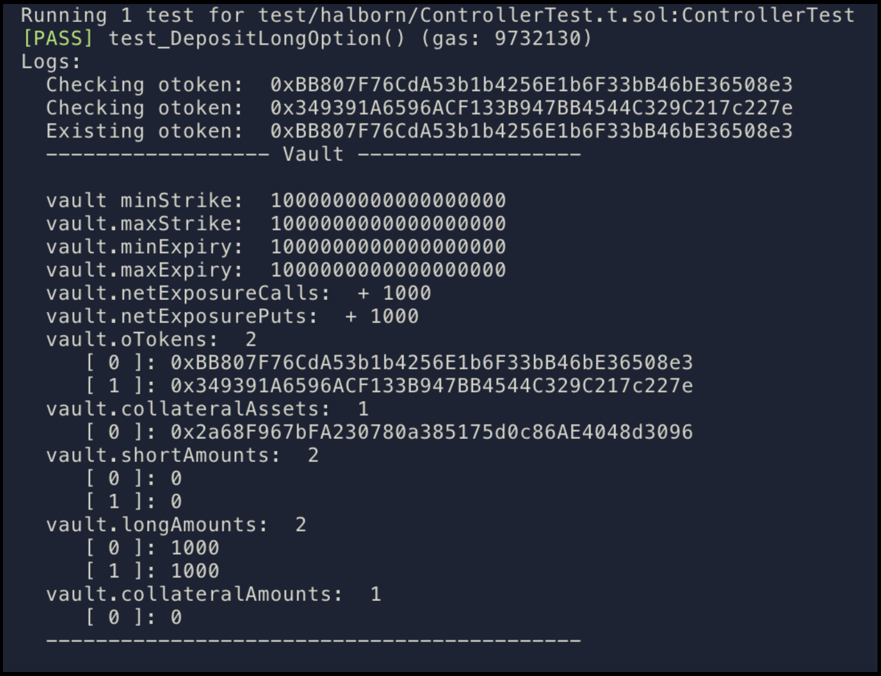
```
enum ActionType {
    OpenVault,
    MintShortOption,
    BurnShortOption,
    DepositLongOption,
    WithdrawLongOption,
    DepositCollateral,
    WithdrawCollateral,
    SettleVault,
    Redeem,
    Call,
    Liquidate,
    MintShortOptionLazy,
    WithdrawLongOptionLazy
}

struct ActionArgs {
    // type of action that is being performed on the system
    ActionType actionType;
    // address of the account owner
    address owner;
    // address which we move assets from or to (depending on the
↳ action type)
    address secondAddress;
    // asset that is to be transferred
    address asset;
    // index of the vault that is to be modified (if any)
    uint256 vaultId;
    // amount of asset that is to be transferred
    uint256 amount;
    // each vault can hold multiple short / long / collateral
↳ assets but we are restricting the scope to only 1 of each in this
↳ version
    // in future versions this would be the index of the short /
↳ long / collateral asset that needs to be modified
    uint256 index;
    // any other data that needs to be passed in for arbitrary
↳ function calls
    bytes data;
}
```

- The _runActions functionality is described:
  - Depending on the type, the _runActions function will execute a different internal function. Some operations may require the outer function (operate) to perform some additional checks.
  - The _getTmpVault is used to retrieve a vault for the address 0 and ID of 0.
  - The getVaultWithDetails will then get the vault for the action owner and vault ID of the action. This function does also return the last update timestamp of that vault. This is performed on the operate function if vaultUpdated is set.
  - The function will then copy over to the tmpVault all the action owner collateralAssets and set its collateralAmounts to 0.
  - **Nothing until this point is preventing the parameters to be the 0 address and id of 0. This means that both tmpVault and returned vault of getVaultWithDetails could be the same.** However, _checkVaultId will later prevent vault ID of 0. Also, some actions will check that the address is not zero.
  - The _verifyFinalState does calculate the excess collateral and verifies that the vault is valid via the getExcessCollateral call.
  - All check on the expiry value for past dates are using < expiry,

for greater >= expiry, which means actions cannot collide in expire timestamps.

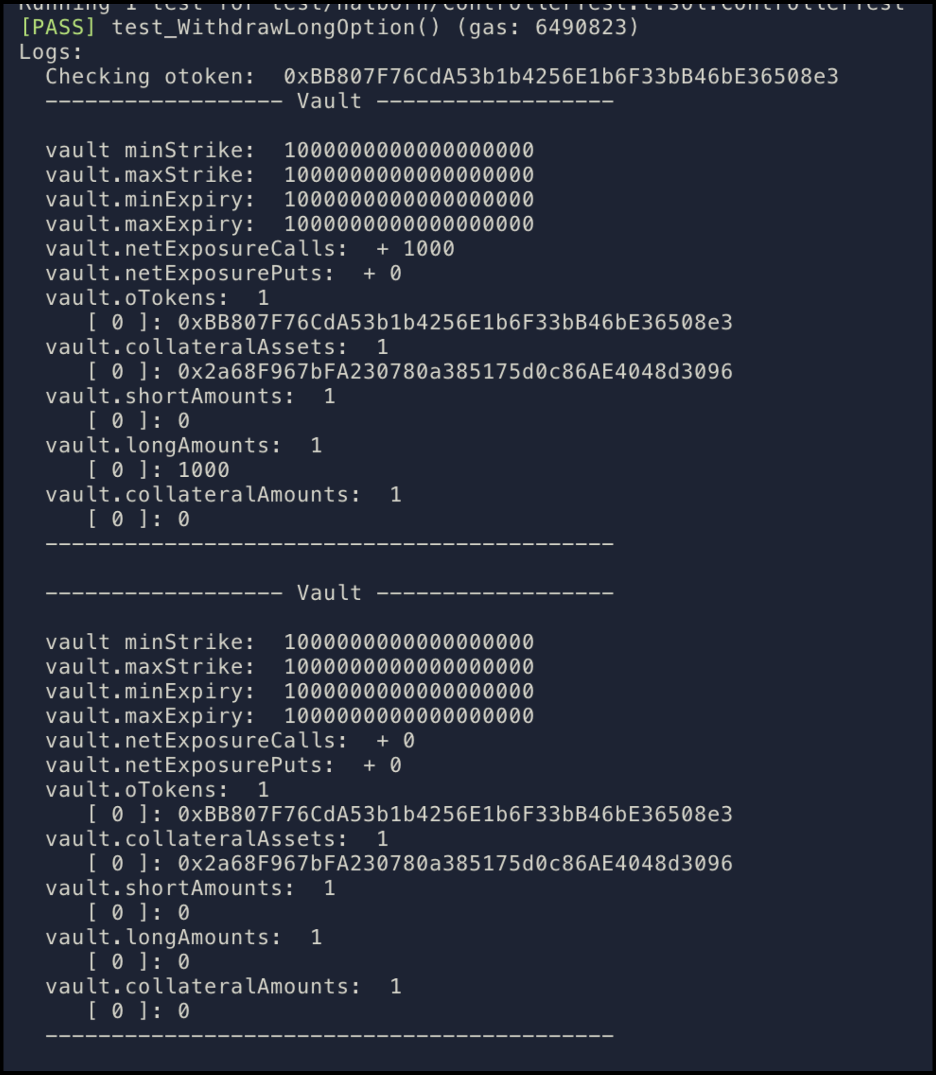
- Actions explained:
  - OpenVault: Will be using _parseOpenVaultArgs to verify if the extra data decoded into an uint256 is on the range of allowed vault types. It will then check the accountVaultCounter counter being the argument and increase it.
  - DepositLongOption: Will be using _parseDepositArgs to check that the owner is not zero address. Then getOtokenIndex to fetch the last index on the otoken array. It then calls the depositLong on the ControllerLib to settle the position on the vault. The indexes are then stored both on the user's vault and on the tmp vault using their corresponding index.
    - A critical check is performed, which consist of verifying that the _args.from parameter is either the vault owner or the sender (which previously the onlyAuthorized did check for permissions from the owner to operate).
    - Also the asset is verified for whitelisting isWhitelistedOtoken and the amount must be greater than 0.
    - If there exists any shortAmounts on the argument index it will get burned. Only a long position will be added with the difference between the requested amount and the burned amount. Example of two deposits, one call and one put:



```
Running 1 test for test/halborn/ControllerTest.t.sol:ControllerTest
[PASS] test_DepositLongOption() (gas: 9732130)
Logs:
  Checking otoken:  0xBB807F76CdA53b1b4256E1b6F33bB46bE36508e3
  Checking otoken:  0x349391A6596ACF133B947BB4544C329C217c227e
  Existing otoken:  0xBB807F76CdA53b1b4256E1b6F33bB46bE36508e3
  ------------------ Vault ------------------

  vault minStrike:  1000000000000000000
  vault.maxStrike:  1000000000000000000
  vault.minExpiry:  1000000000000000000
  vault.maxExpiry:  1000000000000000000
  vault.netExposureCalls:  + 1000
  vault.netExposurePuts:  + 1000
  vault.oTokens:  2
     [ 0 ]: 0xBB807F76CdA53b1b4256E1b6F33bB46bE36508e3
     [ 1 ]: 0x349391A6596ACF133B947BB4544C329C217c227e
  vault.collateralAssets:  1
     [ 0 ]: 0x2a68F967bFA230780a385175d0c86AE4048d3096
  vault.shortAmounts:  2
     [ 0 ]: 0
     [ 1 ]: 0
  vault.longAmounts:  2
     [ 0 ]: 1000
     [ 1 ]: 1000
  vault.collateralAmounts:  1
     [ 0 ]: 0
  -------------------------------------------
```

- WithdrawLongOption, WithdrawLongOptionLazy: Uses _parseWithdrawArgs to set the second address to the destination of withdrawal. Will get the index of the array and verify expiration time. The internal vault removeLong will make sure that no underflow occurs on the subtraction on the index longAmounts. Both netExposurePuts or netExposureCalls will be updated. Example of deposit and withdraw:



```
[PASS] test_WithdrawLongOption() (gas: 6490823)
Logs:
  Checking otoken:  0xBB807F76CdA53b1b4256E1b6F33bB46bE36508e3
  ------------------ Vault ------------------

  vault minStrike:  1000000000000000000
  vault.maxStrike:  1000000000000000000
  vault.minExpiry:  1000000000000000000
  vault.maxExpiry:  1000000000000000000
  vault.netExposureCalls:  + 1000
  vault.netExposurePuts:  + 0
  vault.oTokens:  1
     [ 0 ]: 0xBB807F76CdA53b1b4256E1b6F33bB46bE36508e3
  vault.collateralAssets:  1
     [ 0 ]: 0x2a68F967bFA230780a385175d0c86AE4048d3096
  vault.shortAmounts:  1
     [ 0 ]: 0
  vault.longAmounts:  1
     [ 0 ]: 1000
  vault.collateralAmounts:  1
     [ 0 ]: 0
  -------------------------------------------

  ------------------ Vault ------------------

  vault minStrike:  1000000000000000000
  vault.maxStrike:  1000000000000000000
  vault.minExpiry:  1000000000000000000
  vault.maxExpiry:  1000000000000000000
  vault.netExposureCalls:  + 0
  vault.netExposurePuts:  + 0
  vault.oTokens:  1
     [ 0 ]: 0xBB807F76CdA53b1b4256E1b6F33bB46bE36508e3
  vault.collateralAssets:  1
     [ 0 ]: 0x2a68F967bFA230780a385175d0c86AE4048d3096
  vault.shortAmounts:  1
     [ 0 ]: 0
  vault.longAmounts:  1
     [ 0 ]: 0
  vault.collateralAmounts:  1
     [ 0 ]: 0
  -------------------------------------------
```

For malicious otoken input, the removeLong will verify that the given index does contain the same otoken address

- DepositCollateral: Will parse deposit arguments and verify that the collateral is valid by using isWhitelistedCollateral. It will then call the vault addCollateral function and either append or add into existing index, the amount. However, user can control index, so this probably allows untrusted array manipulation. After analysing the flow by unit testing, only one collateral asset can exist at the time. The MarginCalculator will verify that the vault does not exist in this limit.
- MintShortOption and MintShortOptionLazy: Does use _parseMintArgs verifying that the owner is not the zero address. The to address corresponds to the secondAddress.

Will use the getOtokenIndex system to fetch previous Otoken index if any. It will verify the token being whitelisted and use _checkOtoken to make sure that the assets are being used on the valid vault and verify expiration. If there is any long position with that index Otoken it will offset it with the short by removeLong.

- BurnShortOption: Will use _parseBurnArgs for sanity checks. Only the owner or authorized can burn tokens. Burning expired tokens is not allowed as per “C26” error. It will then add a long position to the tmp vault to offset any other position. **For malicious otoken input, the removeShort will verify that the given index does contain the same otoken address**.
- Redeem: This action can be called by anyone and there is no onlyAuthorized. This means that the redeem will only happen to the sender when calling burnOtoken. Only expired tokens whose isDisputePeriodOver period is over can be redeemed. **The internal getPayout function will call _convertAmountOnExpiryPrice and make sure that prices are finalized, otherwise the real price past the expiration will be used, resulting in incremented losses or undesired profits.**
- SettleVault: It is checking for onlyAuthorized based on the arguments owner. Will add or remove collateral from the vault based on the expired positions and the amount from getExpiredPayout. For each otoken it will iterate over and verify with canSettleAssets for settlement with the expiry date. It will then remove the long/short position.
- Liquidate: It does use the _parseLiquidateArgs which will set the receiver as the second address. Based on the vault state, it will determine if either longs or shorts should be liquidated. In case of short liquidation, It will burn otokens, remove collateral and remove the short position. It will then transfer the liquidated collateral amount to the receiver. For the long position, it will transfer collateral from the sender to the pool and add those as the collateral for the liquidated vault, removing the long on

the process. Finally, it will transfer the otoken position to the receiver address. **For malicious inputs, the otoken is verified to be the same as the being liquidated vault one.**

  - Call: This does allow executing arbitrary calls. It will be using the secondAddress as the contract that needs to be called. The onlyWhitelistedCallee will make sure that only whitelisted contracts can be called.

- The operate function is using a list of ActionArgs as parameters. It will The struct has the following details:

MarginCalculator.sol:

Function	Visibility	Mutability	Modifiers
setCollateralDust	External	Y	onlyOwner
setStrikeIncrement	External	Y	onlyOwner
getCollateralDust	External	N	-
getMarginRequired	External	N	-
getMarginRequiredWithDiff	External	N	-
getExcessCollateral	External	N	-
getExpiredPayout	Public	N	-
_getExpiredCashValue	Internal	N	-
_getMarginRequired	Internal	N	-
min	Internal	N	-
_convertAmountOnLivePrice	Internal	N	-
_convertAmountOnExpiryPrice	Internal	N	-
_convertAmountOnStrikePrice	Internal	N	-
_getVaultDetails	Internal	N	-
_isNotEmpty	Internal	N	-
_checkIsValidVault	Internal	N	-
_getProductHash	Internal	N	-
_getCashValue	Internal	N	-
isLiquidatable	External	N	-
getShortLiquidationPrice	External	N	-
getLongLiquidationPrice	External	N	-
getExpiredPayoutRate	External	N	-
getNakedMarginRequired	External	N	-

General Overview of MarginCalculator Contract

- Purpose: This contract calculates margin requirements and checks the validity of vaults in options trading.
- Inheritance: Inherits from Ownable, suggesting certain functions are restricted to the contract owner.
- Dependencies: Uses SafeMath for safe arithmetic operations and FixedPointInt256 for fixed-point arithmetic.

Key Functions and Features

- **Vault Details Structure**: A comprehensive struct VaultDetails is used to store detailed information about a vault, including positions, expiration times, asset details, and more.
- **Oracle Interface**: Utilizes an oracle interface for fetching asset prices, crucial for calculating margin requirements and payouts.
- **Naked Margin Calculator**: Integrates with an external NakedMarginCalculator contract for specific margin calculations.

Margin Calculation and Validation Functions

- getMarginRequired: Calculates the required margin for a given vault.
- getMarginRequiredWithDiff: Similar to getMarginRequired, but considers position differences in the calculation.
- getExcessCollateral: Determines the excess or deficit collateral in a vault.
- getExpiredPayout: Calculates gains or losses from expired options in a vault.
- isLiquidatable: Checks if a vault is undercollateralized and hence liquidatable.

Setters and Configurations

- setCollateralDust: Sets a minimum collateral amount (dust) for a given asset.
- setStrikeIncrement: Configures the strike price increment for options of specific asset pairs.

Price Conversion and Calculation Helpers

- _convertAmountOnLivePrice: Converts an amount from one asset to another based on current prices.
- _convertAmountOnExpiryPrice: Converts an amount between assets based on prices at a specific expiry timestamp.
- _convertAmountOnStrikePrice: Converts an amount between the strike asset and collateral asset, considering the strike price.

Vault Handling Helpers

- _getVaultDetails: Extracts and structures details from a given vault.
- _checkIsValidVault: Validates the structure and contents of a vault.
- _getProductHash: Generates a hash to uniquely identify an option product.

Liquidation and Payout Calculations

- getShortLiquidationPrice and getLongLiquidationPrice: Calculate the liquidation price for short and long positions.
- getExpiredPayoutRate: Determines the payout rate for an expired option.

Miscellaneous

- min: A helper function to find the minimum of two values.
- _isNotEmpty: Checks if an array of addresses is not empty.

Potential Considerations and Risks

- Complex Calculations: The contract's calculations for margin, liquidation prices, and payouts are multifaceted and depend on accurate inputs, particularly from the oracle and the NakedMarginCalculator.
- External Dependencies: Relies heavily on external contracts (oracle, NakedMarginCalculator) for critical data, introducing dependencies that need to be robust and reliable.
- Oracle Price Accuracy: The oracle's price data is crucial for many calculations. Any inaccuracies or delays in updating prices could affect the contract's functions.
- Vault Validation: The _checkIsValidVault function ensures vault integrity, but the complexity of criteria means any oversight in validation logic could lead to issues in vault handling.
- Dust Configuration: The concept of 'dust' (minimal collateral amount) introduces an additional parameter that needs careful management to prevent dust amounts from being either too restrictive or too permissive.
  #### In-Function details
- getExcessCollateral function does take a valid vault. It will verify that the collateralAssets is no longer than 1. Which means only one collateral asset can exist.
  - The _getVaultDetails will return just the collateral information if oTokens is empty.
  - If there exists oTokens the underlyingAsset, strikeAsset and collateralAsset is defined by the oTokens[0] information.
  - underlyingPrice is obtained using oracle getPrice. The value is factored by 1e8.
  - minExpiry is bumped to block.timestamp if exceeded.
  - numExpiries: Calculates the number of expiry events for a vault. It first checks if there's at least one future expiry, and then adds additional weekly expires if the maximum expiry date is more than a day ahead.

- It will iterate over all OToken long and short position, skipping offsets of 0 amount.
- _convertAmountOnExpiryPrice does have a new argument _requireFinalized that will make sure that the oracle dispute period is over.
- The _getExpiredCashValue is multiplied by -position as a positive value (meaning that there are more shorts than longs) means an obligation in value.
- The _getMarginRequired function does perform an invert for loop, starting from numExpiries and decrementing its value underflowing. When the values are greater than the original numExpiries the loop is completed. This allows iterating from the end of the list to the beginning.
  - numStrikes will be more than 1 always, which means that the second loop based on _vaultDetails.numStrikes will start from either 0 or a positive value.
    - If vars.isPut is true, the range is from _vaultDetails.numStrikes - 1 down to 0 (inclusive).
    - If vars.isPut is false, the range is from 0 up to _vaultDetails.numStrikes - 1 (inclusive).
  - strikePrice is getting calculated based on strikeIncrement and the numStrikes index.
  - The nakedPortion is calculated based on the further expiring remaining long liquidity offset and the current position.
  - To calculate the nakedMargin and nakedPremium, the getNakedMarginRequired2 function is being used. Only on the former, the spotShock is being applied, allowing higher margin on rapid market volatility changes. If the spotShock value is not set, margin will not be incremented by it and standard margin returned.
  - The nakedMargin and coveredNakedMargin are set to the max value of its corresponding Margin and Premium Calls/Puts.

NakedMarginCalculator:

Function	Visibility	Mutability	Modifiers
setUpperBoundValues	External	Y	onlyOwner
updateUpperBoundValue	External	Y	onlyOwner
setSpotShock	External	Y	onlyOwner
getTimesToExpiry	External	N	-
getMaxPrice	External	N	-
getSpotShock	External	N	-
getShockedPrice	External	N	-
_findUpperBoundValue	Internal	N	-
findUpperBoundValue	External	N	-
getNakedMarginRequired	External	N	-
getNakedMarginRequired2	Public	N	-
_getProductHash	Internal	N	-
getOptionType	Public	N	-
makeShortScaledDetails	Internal	N	-

- The getNakedMarginRequired2 function does calculate the margin requirements for a given position by using what is described on https://medium.com/opyn/partially-collateralized-options-now-in-defi-b9d223eb3f4d.
- **Purpose**: This contract is designed to validate vaults, calculate margin requirements, and compute settlement proceeds for options trading.
- **Ownable**: Inherits from the Ownable contract, implying only the owner can execute certain functions.
- **Use of SafeMath and FixedPointInt256**: Ensures safe arithmetic operations and fixed-point arithmetic.

Key Functions and Features

- setUpperBoundValues and updateUpperBoundValue:
  - Sets and updates the upper bound values for options trading.
  - **Risk/Issue**: If not properly updated or managed, could lead to incorrect margin calculations.
- setSpotShock:
  - Sets the spot shock value, which is critical in calculating the required margin.
  - **Risk/Issue**: Inaccurate shock values can lead to either over-collateralization or under-collateralization.
- getNakedMarginRequired and getNakedMarginRequired2:
  - Calculates the collateral required for naked margin vaults.
  - **Risk/Issue**: Complex calculations with multiple inputs; errors in inputs or logic could significantly impact margin requirements.
- Time to Expiry Management:
  - Functions like getTimesToExpiry manage the array of expiry times.
  - **Risk/Issue**: Mismanagement or inaccuracies in expiry times can affect option pricing and risk assessments.
- Product Hash Management:
  - Uses product hashes to identify option contracts uniquely.
  - **Risk/Issue**: Collisions or incorrect hash calculations could lead to misidentification of contracts.
- Option Type Identification (getOptionType):
  - Determines the type of option (PUT, COVERED_CALL, etc.) based on the collateral and underlying assets.
  - **Risk/Issue**: Misclassification can lead to incorrect margin requirements or contract execution.
- Shock Price Calculation (getShockedPrice):
  - Adjusts the underlying price based on the shock value.

    - **Risk/Issue**: Critical for margin calculations; errors can lead to significant financial implications.

- findUpperBoundValue and _findUpperBoundValue:
    - Finds the upper bound value for product by expiry timestamp.
    - **Risk/Issue**: Essential for correct option valuation; discrepancies can lead to mispricing.

Discrepancies and Potential Risks

- **Vault Validation**: The contract appears to focus on naked margin vaults, which might differ from other protocols where vaults can be fully collateralized. **This might lead to a different risk profile compared to protocols mentioned in the Medium article.**
- **Complexity in Margin Calculations**: The contract's margin calculations are complex and depend on several variables. **Any miscalculation or misestimation (like spot shock or upper bound values) could lead to substantial financial risks.**
- **Dependence on External Inputs**: Functions like getNakedMarginRequired depend heavily on external inputs (e.g., underlying price), which introduces a dependency on the accuracy and timeliness of external data.
- **Collateral Asset Limitation**: The contract seems to be designed with a specific focus on certain types of collateral and options. **This may limit its flexibility compared to other protocols that might offer a broader range of collateral types.**
- **Time Sensitivity**: Functions involving expiry times are sensitive to timing and ordering. **Mishandling these aspects could result in incorrect valuations or contract behaviors.**
- **Owner Privileges**: The owner has significant control over critical parameters like upper bound values and spot shock. **This centralization poses a risk of manipulation or error.**

6.2 Core

HedgedPool.sol:

Function	Visibility	Mutability	Modifiers
onlyKeeper	Internal	N	-
onlyAccessKey	Internal	N	-
__HedgedPool_init	Public	Y	initializer
getTotalPoolValue	Public	N	-
getTotalPoolValueCached	Public	N	-
settleAll	Public	Y	-
closeRound	External	Y	onlyKeeper
closeRoundAdmin	External	Y	onlyOwner
_closeRound	Internal	Y	-
processWithdrawals	Internal	Y	-
redeemShares	External	Y	nonReentrant
_redeemShares	Private	Y	-
requestWithdrawal	External	Y	nonReentrant
withdrawCash	External	Y	nonReentrant
requestDeposit	External	Y	nonReentrant, requestDeposit
cancelPendingDeposit	External	Y	nonReentrant
getCollateralBalance	Public	N	-
getCashBuffer	Public	N	-
decimals	Public	N	-
balanceOf	Public	N	-
_transfer	Internal	Y	-
processOrder	External	Y	nonReentrant

getActiveOToken	Public	N	-
getActiveOTokens	Public	N	-
updateSeriesPerExpirationLimit	Public	Y	onlyOwner
processOToken	Internal	Y	-
setHedger	External	Y	onlyOwner
syncMargin	External	Y	onlyKeeper
_syncVaultMargin	Internal	Y	-
configUnderlying	Public	Y	onlyOwner
getAllUnderlyings	External	N	-
setAllowedExpirations	External	Y	onlyOwner
setKeeper	External	Y	onlyOwner
setQuoteProvider	External	Y	onlyOwner
refreshConfiguration	External	Y	onlyOwner
_- refreshConfigInternal	Internal	Y	-
hasUnderlying	External	N	-
setLock	External	Y	onlyOwner

Overview of the HedgedPool Contract The HedgedPool contract, based on the provided code, appears to be a financial instrument in the decentralized finance (DeFi) space. It integrates with the Opyn protocol (Gamma protocol) for options trading and risk management. The contract is upgradeable and includes mechanisms for liquidity provision, options trading, fee collection, and hedging.

Key Functions and Features

1. **Liquidity Management**: Allows liquidity providers (LPs) to deposit and withdraw collateral, with mechanisms to handle these requests.

2. **Round Management**: Implements rounds for deposits and withdrawals, with LPs able to request participation in these rounds.
3. **Options Trading**: Processes orders for options trading, adjusting the pool's exposure and ensuring compliance with predefined strike price ranges and expiration dates.
4. **Expiry Settlements**: Settles expired options and updates vault margins accordingly.
5. **ERC1155 Compliance**: Inherits ERC1155Holder, enabling the contract to manage ERC1155 tokens.
6. **Hedging Mechanism**: Includes functions to set up hedging strategies for different underlying, adjusting the margin in vaults.
7. **Fee Collection**: Integrates with a fee collector for fee management and distribution.
8. **Governance and Access Control**: Implements owner-specific functions for configuration and access control for keepers and other roles.

Considerations and Risks

1. **Complexity and Integration Risks**: High complexity due to interactions with multiple external contracts and protocols (e.g., Gamma protocol, Opyn). This complexity increases the risk of integration errors and unexpected behaviors.
2. **Upgradeability Risks**: Being an upgradeable contract, there's a risk of bugs in future upgrades. Proper governance and secure upgrade paths are essential.
3. **Reentrancy Risks**: Functions interacting with external contracts need safeguarding against reentrancy attacks.
4. **Liquidity and Slippage Concerns**: The mechanisms for managing liquidity rounds and options trading need to ensure fairness and efficiency, minimizing slippage and adverse impacts on LPs.
5. **Smart Contract Dependencies**: Relies heavily on external contracts (e.g., Oracles, Margin Calculators). Any vulnerabilities in these dependencies pose a risk.
6. **Operational Security**: Access control for administrative functions and keeper roles requires stringent management to prevent unauthorized access and centralization risks.

7. **Gas Efficiency**: Complex functions might be gas-inefficient, affecting transaction costs for users.
8. **Regulatory Compliance**: As a financial instrument in the DeFi space, it should adhere to evolving regulatory standards and considerations.
9. **User Experience**: The contract's complexity might pose challenges in user understanding and interaction, necessitating clear documentation and user interfaces.
10. **Audit Necessity**: Given the contract's complexity and financial nature, regular and thorough audits are recommended to identify and mitigate potential vulnerabilities.

In-Function details

- The keepers system with its corresponding functions to set them (setKeeper, only owner), allows function to be limited by some whitelisted entities. **Only closeRound and syncMargin does use this functionality**.
- The onlyAccessKey modifier does restrict functions via the ownership of an ERC1155 token. **Only requestDeposit does use this functionality**.
- redeemShares is expected to be called by anyone. The caller is transferred over into the internal/private function via lpAddress to the redeemShares on the LpManager contract. It will transfer to the caller the sharesAmount.
- get8amAligned can align on Friday because the unix timestamp of 0 corresponds to a Thursday 00:00 GMT. By adding 1 day and 8 hours, you get Friday 8AM GMT.
  - lastSettledExpiry will be set to the next Friday if initialised between Thursday 00:00 GMT and Friday 08:00 GMT. Otherwise, previous Friday is used.
  - withdrawalRoundEnd is lastSettledExpiry minus 1 week. Which means that if initialised before Thursday 00:00 GMT it will correspond to 2 weeks before.
  - depositRoundEnd will always return the current day at 8AM GMT, unless the timestamp for the initialisation is between 00:00-8:00, in that case it will return previous day 8AM GMT time.

> **This means that there is no guarantee that depositRoundEnd will be before or after lastSettledExpiry during initialisation.**

- pricePerShare is a value starting at 1e8, corresponding to a 1. There are two different functions that allow setting the value closeRound and closeRoundAdmin. The former does perform validation on the new value being on the 90-110% range. **The latter does not perform any sort of value validation and only the admin can call it.**
- The round is kept per pool under LpManager on depositRounds and synced with the pool timestamps.
- requestDeposit is only allowed to NFT holders. It will transfer the collateral amount to the pool and requestDeposit on the LpManager.
- closeRound and closeRoundAdmin, will verify that lastSettledExpiry was reached and if in the withdrawalRoundEnd period it will call closeWithdrawalRound on the LpManager and extend the withdrawalRoundEnd by 1 week. It will also store the new price per share specified on the arguments under pricePerShareCached.
    - If the depositRoundEnd is in the past, after 8AM of that day the closeDepositRound on the LpManager is called and the depositRoundEnd incremented by 1 day.
    - If the withdrawalRoundEnd is in the past, after 8AM of Friday, the closeWithdrawalRound on the LpManager is called and withdrawalRoundEnd incremented by 1 week.
- cancelPendingDeposit: will call cancelPendingDeposit on the ILpManager and transfer the collateral back to the caller.
- requestWithdrawal: Will first redeem any unredeemed shares first. Then perform the request on the LpManager and finally burn the shares amount from the caller and mint it on the pool to track.
- withdrawCash does call the LpManager.withdrawCash function and does transfer the cash amount to the user, if any.

- processOrder is being called by the executor. It will verify if the pool has enough collateral to cover the trade. For each order legs, it will process the token.
    - processOToken : If the otoken is not present as active it will validate the expiry timestamp, verify count limits and that strike ranges are set for it. It will then track the new otoken as active and store it based on the expiry under oTokensByExpiry.
- settleAll can be called by anyone.
- syncMargin, only callable by keepers. There is no check on whether hedgers[underlying[i]] is none zero or set. Any collateral excess is moved out of vaults. For the underlaying, the corresponding hedger sync is called.
- setHedger, only allowed to the owner. It will register a hedger for a given underlying token and approve the collateral to spend from the pool. It will revoke previous hedge if exists.
- configUnderlying allows setting for a given _underlying the allowedStrikeRanges and spotShockPercent. It will also register it on the underlyingTokens list and create a new vault for the caller via openMarginVault. This function is owner protected. If the user does remove the underlying and adds it again, the open vault function will not fail, as the new vault is appended to the marginVaults mapping array.
- setAllowedExpirations does set the underlying numMonths, numQuarters and allowDailyExpiration. If those values are not set, they are not causing any issue on the isValidExpiry function and a false is returned.
- processWithdrawals will get the cash buffer, based on puts/calls positions and the shock percentage. Based on the unfilled shares and the cached price per share, for the last closed round and an extra 10% it does calculate the required amount: uint256 requiredAmount = (unfilledShares * pricePerShareCached * 11)/1e9. (The operation keep in mind that pricePerShareCached is 1e8 based and the 11 corresponds to a 110 increment which gets divided by an extra 1e1). If there is amount to withdraw, the addPendingCash function is called.

LpManager.sol:

In-Function details

- **redeemShares can be called by anyone, there is no restriction on the caller being a valid pool**. Internally, the getDepositStatus is called.
  - It will always reset the roundId to 0 if no cashPending.
- requestDeposit does store based on the caller (anyone can call the function, no modifier restriction) the cash amount per this deposit round. unredeemedShares will contain any previous amount for past pending rounds.
  - It will always bump the deposit round to latest.
- closeWithdrawalRound based on the caller, treated as the pool. Will compute the cash needed based on the already filled shares and the total shares times the price per share (the result will be divided by 1e8 as per the pricePerShare precision). The pending shares, will be added to the sharesFilled. The cashPending, added to the total cash.
- getWithdrawalStatus will verify if shares a present for withdrawal and return 0 if not.
- getDepositStatus will return the deposit amount if the current round is the same as the one stored on the depositor as pending cash. Otherwise, it will get the round id for the last deposit and compute the sharesRedeemable based on the pricePerShare of that round. This function is only used on redeemShares.
  - During deposit, if a previous round was not redeemed, they will be storing the amount to unredeemedShares already.
- cancelPendingDeposit will act only on the current round id and decrease both the DepositRound and Deposit amounts.
- closeDepositRound: Will store the previous round pricePerShare and advance on the next round, returning the added shares based on the new pricePerShare.
- addPendingCash which is called during processWithdrawals will increment the cashPending for the current round with the parameter.

Gmx2Hedger.sol:

Function	Visibility	Mutability	Modifiers
hedgerType	External	N	-
updateConfig	External	Y	onlyOwner
<Receive Ether>	External	N	-
_updateConfig	Private	Y	-
afterOrderExecution	External	Y	-
afterOrderCancellation	External	Y	-
afterOrderFrozen	External	Y	-
hedge	External	N	onlyAuthorized
sync	External	Y	onlyAuthorized
getDelta	External	N	-
getCollateralValue	External	N	-
getRequiredCollateral	External	N	-
_sync	Internal	Y	-
_syncDelta	Internal	Y	noPendingOrders
_getPositionInformation	Internal	N	-
_getPositinPnlCollateral	Internal	N	-
_createOrderParamAddresses	Internal	N	-
_orderParamAddresses	Internal	N	-
_createOrderParamNumbers	Internal	N	-
_orderParamNumbers	Internal	N	-
_createMarketOrder	Internal	Y	-
_sendCollateral	Internal	Y	-
_changePosition	Internal	Y	-

_depositCollateral	Internal	Y	-
_-withdrawCollateral	Internal	Y	-
_-gmxPositionIncrease	Internal	Y	-
_-gmxPositionDecrease	Internal	Y	-
_-gmxDepositCollateral	Internal	Y	-
_-gmxWithdrawCollateral	Internal	Y	-
_-calculateAdjustedExecutionPrice	Internal	N	-
_getPositionKey	Internal	N	-
_getMarginRequired	Internal	N	-
_-getAcceptablePrice	Internal	N	-
_getMarketPrices	Internal	N	-
_-calculateExecutionFee	Internal	N	-
transfer	Public	Y	onlyOwner
resetPendingOrder	Public	Y	onlyOwner

Overview The Gmx2Hedger contract is designed for the GMX v2 protocol, a decentralized platform for trading perpetual contracts. It primarily focuses on managing and hedging pool delta by engaging in perpetual contract trading on GMX v2. The contract incorporates elements from OpenZeppelin's upgradeable contracts and uses SafeERC20 for safer ERC20 interactions.

Key Functions and Features

1. **Hedging Delta Exposure**: Executes strategies to hedge the delta exposure of a pool through perpetual contracts on GMX v2.

2. **Position Management**: Manages the opening, adjustment, and closing of positions on GMX v2.
3. **Collateral Handling**: Manages the collateral associated with perpetual contracts, including deposits and withdrawals.
4. **Order Execution**: Handles the creation, execution, and tracking of orders on GMX v2.
5. **Price and Market Data**: Utilizes GMX v2 oracles for fetching market and price data required for decision-making.
6. **Configurability**: Allows configuration of key parameters such as leverage, exchange router, and market details.
7. **Access Control**: Leverages OpenZeppelin's OwnableUpgradeable for owner-specific functions and provides mechanisms for authorized operations.
8. **Callback Handling**: Implements callbacks for post-order execution activities.
9. **Gas and Fee Management**: Manages the computation and payment of gas and other transaction-related fees.

Considerations and Risks

1. **Complex Financial Operations**: Managing perpetual contracts involves complex financial mechanisms that carry inherent risks and require precise execution.
2. **Dependence on External Protocols**: The contract's functionality is closely tied to the GMX v2 protocol and its associated contracts. Changes in these external systems could impact the contract's operations.
3. **Upgradeable Contract Risks**: Being upgradeable introduces additional layers of complexity and potential points of failure.
4. **Gas Efficiency**: Operations, especially those interacting with external contracts, might be gas-intensive, affecting the cost-effectiveness of transactions.

5. **Security and Access Control**: Functions with restricted access need stringent security measures to prevent unauthorized or malicious actions.

6. **Market Risks and Slippage**: Market volatility and slippage can impact the contract's hedging efficiency and overall performance.

7. **Smart Contract Security**: Vulnerabilities or bugs in the contract or in the integrated GMX v2 components could lead to financial losses.

8. **Regulatory Compliance**: Involvement in derivative trading and hedging might require adherence to specific regulations, depending on the jurisdiction.

In-Function details

- The afterOrderExecution similarly to the gmxPositionCallback on version 1 does verify that the caller is indeed the orderHandler.
- The internal _getPositionInformation function will use the none inclusive range 0-2 on the getAccountPositions reader's function. Returning the long and short positions respectively.
- _syncDelta does use _getPositionInformation to get the long and short positions. It will then remove the totalSizeInTokens decimals by dividing it by the collateral decimals. The value is scaled to 1e8 (SIREN_DECIMALS).
- sync will call the _syncDelta function and _getPositionInformation to fetch current positions and _getMarketPrices to obtain last prices.
    - The _getPositionKey does mimic the PositionUtils.getPositionKey .
    - The _getPositinPnlCollateral does use the reader getPositionPnlUsd to fetch the current position PLN in GMX decimals. It is then converted to collateral decimals precision.
    - It the required margin is less than the current, it will perform a _depositCollateral with the amount of difference and an acceptable price threshold. Otherwise, a _withdrawCollateral.

- _depositCollateral does cap the amount to the pool max amount and calls _gmxDepositCollateral. **However, the collateralDiff is using the full amount, and not the capped amount.**
  - The execution fee is set to the current tx.gasprice times 7000000 .
- hedge will be based on a target, increase or decrease the GMX position via _changePosition internal function.

TradeExecutor:

Function	Visibility	Mutability	Modifiers
setAddressBook	External	Y	onlyOwner
onlyOrderKeeper	Internal	N	-
setOrderKeeper	External	Y	onlyOwner
setMinExecutionFee	External	Y	onlyOwner
onlyAccessKey	Internal	N	-
validateExpirations	Internal	N	-
submitOrder	External	N	-
getNextKey	Internal	Y	-
validateExecuteOrderArguments	Internal	N	-
deleteOrder	Internal	Y	-
sendRefund	Internal	Y	-
completeOrderAndIssueRefund	Internal	Y	-
executeActions	Public	Y	-
executeOrder	External	Y	nonReentrant
cancelOrder	External	Y	nonReentrant
setLock	External	Y	onlyOwner
openMarginVault	External	Y	-
_openMarginVault	Internal	Y	-
getMarginVaultId	External	N	-
setAuthorizedPool	External	Y	onlyOwner
getOrderKeys	Public	N	-
getOrderKeysByAccount	Public	N	-

Overview The TradeExecutor contract serves as a comprehensive platform for executing and managing options trades on the Opyn Gamma protocol.

It provides a structured environment for users to submit, execute, and manage orders, alongside handling collateral and vault interactions.

Key Features

1. **Order Management**: The contract allows users to submit and cancel orders, ensuring that each order is uniquely identified and stored on-chain.
2. **Trade Execution**: It includes mechanisms to execute trades based on submitted orders, handling various aspects like option leg validation, expiration checks, and fee calculations.
3. **Collateral and Vault Management**: The contract integrates with the Opyn Gamma protocol for options trading, managing collateral deposits, withdrawals, and vault operations.
4. **Fee Handling**: It manages the execution fees, including setting minimum fees, refunding on order cancellation, and compensating keepers for executing trades.
5. **Access Control**: Functions are in place to restrict certain operations to authorized users, such as order keepers or the contract owner.
6. **Order Validation**: Validates orders and their components (legs) for expiration and integrity, ensuring that only executable orders are processed.

Considerations and Risks:

1. **Smart Contract Risk**: As with any complex smart contract, there is a risk of bugs or vulnerabilities, which could impact the execution and management of orders.
2. **Dependency on External Protocols**: The contract's functionality is closely tied to the Opyn Gamma protocol. Any changes or issues in the Opyn Gamma protocol could directly affect the TradeExecutor.

3. **Execution Fee Fluctuations**: The cost of executing orders could vary with network conditions. Although the contract allows adjusting minimum fees, rapid changes in network conditions could impact users' costs.

4. **Order Cancellation and Refunds**: The process of order cancellation and refunding execution fees adds complexity and potential points of failure, especially in cases where refunds fail.

5. **Order Expiration Handling**: The contract's strict checks on order and leg expirations, while ensuring timeliness, could lead to increased transaction rejections, especially in fast-moving market conditions.

6. **Access Key Requirement**: The contract has a mechanism to lock or unlock order submission based on access key token ownership, which could limit accessibility for some users.

7. **Keeper Dependency**: The reliance on external keepers for order execution introduces a human element, which could lead to delays or failures in order execution.

The TradeExecutor contract provides a structured and feature-rich environment for options trading but comes with inherent complexities and dependencies typical of advanced DeFi protocols. Users and developers should consider these aspects and conduct thorough due diligence before interaction.

In-Function details

- validateExpirations will iterate each leg and make sure that none is expired. This function, instead of reverting itself it will return true/false which always poses a risk if the returned value is not verified. In this case, it is being used with a revert under submitOrder and with a terminating return on executeOrder. **The latter is not being called from another function that would assume that a return meant a valid execution.**

- openMarginVault allows creating a new vault by any caller. In case of deposit, the _collateralAsset will be verified on the controller, but not the _underlyingAsset (which is latter verified using validateExecuteOrderArguments)
- submitOrder can be called only with onlyAccessKey owners.
  - It will validate expiration on all legs.
  - Make sure that if a vault id is not specified, it will open a new vault for the underlying. **However, it does not validate the traderVaultId is valid for this sender.**
  - It will then store the order via a unique nonce key to the storage.
- executeOrder will use validateExecuteOrderArguments to validate all order parameters. This function does perform underlaying checks and premium amount limits.
  - It will then validate all order expirations and in case of invalid completeOrderAndIssueRefund will be called. This function does perform gas costs checks to refund the keeper with the execution costs
- executeActions do correctly iterate over all legs and add it to the corresponding actions1, actions2 and actions3.

AUTOMATED TESTING

7.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance the coverage of certain areas of the scoped contracts. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their ABI and binary formats, Slither was run on the all-scoped contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Slither results:

GammaProtocol

Slither results for GammaProtocol	
Finding	**Impact**
MarginPool.transferToPool(address,address,uint256) (contracts/core/MarginPool.sol#74-85) uses arbitrary from in transferFrom: ERC20Interface(_asset).safeTransferFrom(_user,address (this),_amount) (contracts/core/MarginPool.sol#83)	High
Otoken._getNameAndSymbol() (contracts/core/Otoken.sol#119-172) calls abi.encodePacked() with multiple dynamic arguments: - tokenName = string(abi.encodePacked(underlying,strike, ,_uintTo2Chars(day),-,monthFull,-,Strings.toString(year), ,displayStrikePrice,typeFull, ,collateral, Collateral)) (contracts/core/Otoken.sol#135-152)	High
PayableProxyController.operate(Actions.ActionArgs[],address) (contracts/external/proxies/PayableProxyController.sol#51-85) use msg.value in a loop: ERC20Interface(address(weth)).safeIncreaseAllowance(action.secondAddress,msg.value) (contracts/external/proxies/PayableProxyController.sol#71)	High

Finding	Impact
NakedMarginCalculator.timesToExpiryForProduct (contracts/core/calculators/NakedMarginCalculator.sol#40) is never initialized. It is used in: - NakedMarginCalculator.setUpperBoundValues(address,address,address,bool,uint256[],uint256[]) (contracts/core/calculators/NakedMarginCalculator.sol#78-118)- NakedMarginCalculator.getTimesToExpiry(address,address,address,bool) (contracts/core/calculators/NakedMarginCalculator.sol#184-192)- NakedMarginCalculator._findUpperBoundValue(bytes32,uint256) (contracts/core/calculators/NakedMarginCalculator.sol#256-286)	High
MarginCalculator._getMarginRequired(MarginCalculator.VaultDetails) (contracts/core/MarginCalculator.sol#470-801) performs a multiplication on the result of a division: - coveredPortion = FixedPointInt256.fromScaledUint(longOffset,BASE).div(fullPosition) (contracts/core/MarginCalculator.sol#584-586) - vars.coveredNakedMarginCalls = vars.coveredNakedMarginCalls.add(nakedMargin).mul(coveredPortion) (contracts/core/MarginCalculator.sol#641-643)	Medium
MarginCalculator._getMarginRequired(MarginCalculator.VaultDetails) (contracts/core/MarginCalculator.sol#470-801) performs a multiplication on the result of a division: - coveredPortion = FixedPointInt256.fromScaledUint(longOffset,BASE).div(fullPosition) (contracts/core/MarginCalculator.sol#584-586) - vars.coveredNakedMarginPuts = vars.coveredNakedMarginPuts.add(nakedMargin).mul(coveredPortion) (contracts/core/MarginCalculator.sol#649-651)	Medium
MarginCalculator._getMarginRequired(MarginCalculator.VaultDetails) (contracts/core/MarginCalculator.sol#470-801) performs a multiplication on the result of a division: - coveredPortion = FixedPointInt256.fromScaledUint(longOffset,BASE).div(fullPosition) (contracts/core/MarginCalculator.sol#584-586) - vars.coveredNakedPremiumPuts = vars.coveredNakedPremiumPuts.add(nakedPremium).mul(coveredPortion) (contracts/core/MarginCalculator.sol#652-654)	Medium

Finding	Impact
MarginCalculator._getMarginRequired(MarginCalculator.VaultDetails) (contracts/core/MarginCalculator.sol#470-801) performs a multiplication on the result of a division: - coveredPortion = FixedPointInt256.fromScaledUint(longOffset,BASE) .div(fullPosition) (contracts/core/MarginCalculator.sol#584-586) - vars.coveredNakedPremiumCalls = vars.coveredNakedPremiumCalls.add (nakedPremium).mul(coveredPortion) (contracts/core/MarginCalculator.sol#644-646)	Medium
MarginCalculator._getMarginRequired(MarginCalculator.VaultDetails) (contracts/core/MarginCalculator.sol#470-801) uses a dangerous strict equality: - vars.optionType == NakedMarginCalculatorInterface.OptionType.PUT || vars.optionType == NakedMarginCalculatorInterface.OptionType.NAKED_CALL (contracts/core/MarginCalculator.sol#725)	Medium
MarginCalculator._convertAmountOnExpiryPrice(FixedPointInt256.FixedPointInt,address,address,uint256,bool) (contracts/core/MarginCalculator.sol#839-864) uses a dangerous strict equality: - priceA == 0 (contracts/core/MarginCalculator.sol#857)	Medium
MarginCalculator._getVaultDetails(MarginVault.Vault) (contracts/core/MarginCalculator.sol#898-1033) uses a dangerous strict equality: - require(bool,string)(vaultDetails.expirations[expirationId] == 0 || vaultDetails.expirations[expirationId] == expiryTimestamp,duplicate expirationId) (contracts/core/MarginCalculator.sol#1001-1005)	Medium
MarginCalculator._getMarginRequired(MarginCalculator.VaultDetails) (contracts/core/MarginCalculator.sol#470-801) uses a dangerous strict equality: - vars.isPut && _vaultDetails.putSumShort[expId] == 0 && _vaultDetails.putSumLong[expId] == 0 (contracts/core/MarginCalculator.sol#510)	Medium
MarginCalculator._convertAmountOnExpiryPrice(FixedPointInt256.FixedPointInt,address,address,uint256,bool) (contracts/core/MarginCalculator.sol#839-864) uses a dangerous strict equality: - priceB == 0 (contracts/core/MarginCalculator.sol#858)	Medium

Finding	Impact
MarginCalculator._getVaultDetails(MarginVault.Vault) (contracts/core/MarginCalculator.sol#898-1033) uses a dangerous strict equality: - vaultDetails.expirations[expirationId] == 0 (contracts/core/MarginCalculator.sol#1006)	Medium
MarginCalculator._getMarginRequired(MarginCalculator.VaultDetails) (contracts/core/MarginCalculator.sol#470-801) uses a dangerous strict equality: - vars.optionType == NakedMarginCalculatorInterface.OptionType.NAKED_PUT || vars.optionType == NakedMarginCalculatorInterface.OptionType.COVERED_CALL (contracts/core/MarginCalculator.sol#738)	Medium
MarginCalculator._getMarginRequired(MarginCalculator.VaultDetails) (contracts/core/MarginCalculator.sol#470-801) uses a dangerous strict equality: - ! vars.isPut && _vaultDetails.callSumShort[expId] == 0 && _vaultDetails.callSumLong[expId] == 0 (contracts/core/MarginCalculator.sol#506)	Medium

Finding	Impact
Reentrancy in Controller.operate(Actions.ActionArgs[]) (contracts/core/Controller.sol#360-390): External calls: - (vaultUpdated,needsFullMarginCheck,vaultOwner,vaultId) = _runActions(_actions) (contracts/core/Controller.sol#367) - CalleeInterface(_args.callee).callFunction(msg.sender,_args.data) (contracts/core/Controller.sol#970) - ControllerLib.redeem(_args,pool,oracle,calculator,whitelist) (contracts/core/Controller.sol#937) - ControllerLib.settleVault(_args,pool,oracle,calculator,vaults[_args.owner][_args.vaultId]) (contracts/core/Controller.sol#948) - ControllerLib.depositCollateral(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),pool,whitelist) (contracts/core/Controller.sol#859) - ControllerLib.withdrawCollateral(_args,vaults[_args.owner][_args.vaultId],pool) (contracts/core/Controller.sol#874) - pool.transferToUser(_args.asset,_args.to,_args.amount) (contracts/libs/ControllerLib.sol#224) - ControllerLib.liquidate(_args,pool,calculator,vaults[_args.owner][_args.vaultId]) (contracts/core/Controller.sol#961) - ControllerLib.burnOtoken(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),tmpVaultOtokenIndex) (contracts/core/Controller.sol#925) - ControllerLib.mintOtoken(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),tmpVaultOtokenIndex,pool,whitelist) (contracts/core/Controller.sol#894-901) - ControllerLib.withdrawLong(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),tmpVaultOtokenIndex,pool) (contracts/core/Controller.sol#836-842) - ControllerLib.depositLong(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),tmpVaultOtokenIndex,pool,whitelist) (contracts/core/Controller.sol#805-812) - pool.transferToUser(_args.asset,_args.to,_args.amount) (contracts/libs/ControllerLib.sol#176) - pool.transferToPool(_args.asset,_args.from,_args.amount) (contracts/libs/ControllerLib.sol#207) - otoken.burnOtoken(_args.from,_args.amount) (contracts/libs/ControllerLib.sol#304) - otoken.burnOtoken(msg.sender,_args.amount) (contracts/libs/ControllerLib.sol#338) - pool.transferToUser(collateral,_args.receiver,payout) (contracts/libs/ControllerLib.sol#340) - OtokenInterface(_args.asset).burnOtoken(_args.from,burnAmount) (contracts/libs/ControllerLib.sol#138) - pool.transferToUser(_args.otoken,_args.to,longOffset)	Medium

Finding	Impact
Reentrancy in Controller._depositLong(Actions.DepositArgs) (contracts/core/Controller.sol#793-817): External calls: - ControllerLib.depositLong(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),tmpVaultOtokenIndex,pool,whitelist) (contracts/core/Controller.sol#805-812) State variables written after the call(s): - oTokenIndexes[_args.owner][_args.vaultId][_args.asset] = _args.index (contracts/core/Controller.sol#814)Controller.oTokenIndexes (contracts/core/Controller.sol#110) can be used in cross function reentrancies: - Controller._cleanTmpVault() (contracts/core/Controller.sol#345-352) - Controller.getOtokenIndex(address,uint256,address) (contracts/core/Controller.sol#626-642) - Controller.oTokenIndexes (contracts/core/Controller.sol#110) - oTokenIndexes[address(0)][0][_args.asset] = tmpVaultOtokenIndex (contracts/core/Controller.sol#816)Controller.oTokenIndexes (contracts/core/Controller.sol#110) can be used in cross function reentrancies: - Controller._cleanTmpVault() (contracts/core/Controller.sol#345-352) - Controller.getOtokenIndex(address,uint256,address) (contracts/core/Controller.sol#626-642) - Controller.oTokenIndexes (contracts/core/Controller.sol#110)	Medium
Reentrancy in Controller._withdrawLong(Actions.WithdrawArgs) (contracts/core/Controller.sol#824-845): External calls: - ControllerLib.withdrawLong(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),tmpVaultOtokenIndex,pool) (contracts/core/Controller.sol#836-842) State variables written after the call(s): - oTokenIndexes[address(0)][0][_args.asset] = tmpVaultOtokenIndex (contracts/core/Controller.sol#844)Controller.oTokenIndexes (contracts/core/Controller.sol#110) can be used in cross function reentrancies: - Controller._cleanTmpVault() (contracts/core/Controller.sol#345-352) - Controller.getOtokenIndex(address,uint256,address) (contracts/core/Controller.sol#626-642) - Controller.oTokenIndexes (contracts/core/Controller.sol#110)	Medium

Finding	Impact
Reentrancy in Controller._runActions(Actions.ActionArgs[]) (contracts/core/Controller.sol#654-749): External calls: - _depositLong(Actions._parseDepositArgs(action)) (contracts/core/Controller.sol#721) - ControllerLib.depositLong(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),tmpVaultOtokenIndex,pool,whitelist) (contracts/core/Controller.sol#805-812) - OtokenInterface(_args.asset).burnOtoken(_args.from,burnAmount) (contracts/libs/ControllerLib.sol#138) - pool.transferToPool(_args.asset,_args.from,_args.amount - burnAmount) (contracts/libs/ControllerLib.sol#146) - _withdrawLong(Actions._parseWithdrawArgs(action)) (contracts/core/Controller.sol#726) - ControllerLib.withdrawLong(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),tmpVaultOtokenIndex,pool) (contracts/core/Controller.sol#836-842) - pool.transferToUser(_args.asset,_args.to,_args.amount) (contracts/libs/ControllerLib.sol#176) - _depositCollateral(Actions._parseDepositArgs(action)) (contracts/core/Controller.sol#728) - ControllerLib.depositCollateral(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),pool,whitelist) (contracts/core/Controller.sol#859) - pool.transferToPool(_args.asset,_args.from,_args.amount) (contracts/libs/ControllerLib.sol#207) - _withdrawCollateral(Actions._parseWithdrawArgs(action)) (contracts/core/Controller.sol#730) - ControllerLib.withdrawCollateral(_args,vaults[_args.owner][_args.vaultId],pool) (contracts/core/Controller.sol#874) - pool.transferToUser(_args.asset,_args.to,_args.amount) (contracts/libs/ControllerLib.sol#224) - _mintOtoken(Actions._parseMintArgs(action)) (contracts/core/Controller.sol#734) - ControllerLib.mintOtoken(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),tmpVaultOtokenIndex,pool,whitelist) (contracts/core/Controller.sol#894-901) - pool.transferToUser(_args.otoken,_args.to,longOffset) (contracts/libs/ControllerLib.sol#262) - OtokenInterface(_args.otoken).mintOtoken(_args.to,_args.amount - longOffset) (contracts/libs/ControllerLib.sol#269) - _burnOtoken(Actions._parseBurnArgs(action)) (contracts/core/Controller.sol#736) - ControllerLib.burnOtoken(_args,vaults[_args.owner][_args.vaultId],_getTmpVault(),tmpVaultOtokenIndex)	Medium

Finding	Impact
Reentrancy in Controller._mintOtoken(Actions.MintArgs) (contracts/core/Controller.sol#882-905): External calls: - ControllerLib.mintOtoken(_args,vaults[_args.owner][_args.vaultId] ,_getTmpVault(),tmpVaultOtokenIndex,pool,whitelist) (contracts/core/Controller.sol#894-901) State variables written after the call(s): - oTokenIndexes[_args.owner][_args.vaultId][_args.otoken] = _args.index (contracts/core/Controller.sol#903)Controller.oTokenIndexes (contracts/core/Controller.sol#110) can be used in cross function reentrancies: - Controller._cleanTmpVault() (contracts/core/Controller.sol#345-352) - Controller.getOtokenIndex(address,uint256,address) (contracts/core/Controller.sol#626-642) - Controller.oTokenIndexes (contracts/core/Controller.sol#110) - oTokenIndexes[address(0)][0][_args.otoken] = tmpVaultOtokenIndex (contracts/core/Controller.sol#904)Controller.oTokenIndexes (contracts/core/Controller.sol#110) can be used in cross function reentrancies: - Controller._cleanTmpVault() (contracts/core/Controller.sol#345-352) - Controller.getOtokenIndex(address,uint256,address) (contracts/core/Controller.sol#626-642) - Controller.oTokenIndexes (contracts/core/Controller.sol#110)	Medium
Reentrancy in Controller._burnOtoken(Actions.BurnArgs) (contracts/core/Controller.sol#912-929): External calls: - ControllerLib.burnOtoken(_args,vaults[_args.owner][_args.vaultId] ,_getTmpVault(),tmpVaultOtokenIndex) (contracts/core/Controller.sol#925) State variables written after the call(s): - oTokenIndexes[address(0)][0][_args.otoken] = tmpVaultOtokenIndex (contracts/core/Controller.sol#928)Controller.oTokenIndexes (contracts/core/Controller.sol#110) can be used in cross function reentrancies: - Controller._cleanTmpVault() (contracts/core/Controller.sol#345-352) - Controller.getOtokenIndex(address,uint256,address) (contracts/core/Controller.sol#626-642) - Controller.oTokenIndexes (contracts/core/Controller.sol#110)	Medium

Finding	Impact
ControllerLib.withdrawLong(Actions.WithdrawArgs,MarginVault.Vault,MarginVault.Vault,uint256,MarginPoolInterface) (contracts/libs/ControllerLib.sol#162-182) ignores return value by (expiry,isPut) = OtokenInterface(otoken).getOtokenDetails() (contracts/libs/ControllerLib.sol#171)	Medium
ControllerLib.depositLong(Actions.DepositArgs,MarginVault.Vault,MarginVault.Vault,uint256,MarginPoolInterface,WhitelistInterface) (contracts/libs/ControllerLib.sol#109-155) ignores return value by (strike,expiry,isPut) = OtokenInterface(_args.asset).getOtokenDetails() (contracts/libs/ControllerLib.sol#126)	Medium
ControllerLib.liquidate(Actions.LiquidateArgs,MarginPoolInterface,MarginCalculatorInterface,MarginVault.Vault) (contracts/libs/ControllerLib.sol#437-553) ignores return value by (isUnderCollatAfter,totalDebtAfter) = calculator.isLiquidatable(vault) (contracts/libs/ControllerLib.sol#530)	Medium
ChainLinkPricer.getPrice() (contracts/pricers/ChainlinkPricer.sol#83-88) ignores return value by (answer) = aggregator.latestRoundData() (contracts/pricers/ChainlinkPricer.sol#84)	Medium
YearnPricer.setExpiryPriceInOracle(uint256) (contracts/pricers/YearnPricer.sol#60-65) ignores return value by (underlyingPriceExpiry) = oracle.getExpiryPrice(address(underlying),_expiryTimestamp) (contracts/pricers/YearnPricer.sol#61)	Medium
Controller.getVaultExtended(address,uint256) (contracts/core/Controller.sol#556-595) ignores return value by (strikePrice,expiryTimestamp,isPut) = OtokenInterface(vault.oTokens[i_scope_0]).getOtokenDetails() (contracts/core/Controller.sol#580-581)	Medium
ChainLinkPricer.setExpiryPriceInOracle(uint256,uint80) (contracts/pricers/ChainlinkPricer.sol#70-76) ignores return value by (price,roundTimestamp) = aggregator.getRoundData(_roundId) (contracts/pricers/ChainlinkPricer.sol#71)	Medium

Finding	Impact
CompoundPricer.setExpiryPriceInOracle(uint256) (contracts/pricers/CompoundPricer.sol#58-63) ignores return value by (underlyingPriceExpiry) = oracle.getExpiryPrice(address(underlying),_expiryTimestamp) (contracts/pricers/CompoundPricer.sol#59)	Medium
ControllerLib.mintOtoken(Actions.MintArgs,MarginVault.Vault,MarginVault.Vault,uint256,MarginPoolInterface,WhitelistInterface) (contracts/libs/ControllerLib.sol#234-278) ignores return value by (strike,expiry,isPut) = OtokenInterface(_args.otoken).getOtokenDetails() (contracts/libs/ControllerLib.sol#246)	Medium
WstethPricer.setExpiryPriceInOracle(uint256) (contracts/pricers/WstethPricer.sol#71-76) ignores return value by (underlyingPriceExpiry) = oracle.getExpiryPrice(underlying,_expiryTimestamp) (contracts/pricers/WstethPricer.sol#72)	Medium
ControllerLib.burnOtoken(Actions.BurnArgs,MarginVault.Vault,MarginVault.Vault,uint256) (contracts/libs/ControllerLib.sol#285-310) ignores return value by (expiry,isPut) = OtokenInterface(otoken).getOtokenDetails() (contracts/libs/ControllerLib.sol#295)	Medium
MarginCalculator._getVaultDetails(MarginVault.Vault) (contracts/core/MarginCalculator.sol#898-1033) ignores return value by (strikePrice,expiryTimestamp,isPut) = OtokenInterface(_vault.oTokens[i_scope_0]).getOtokenDetails() (contracts/core/MarginCalculator.sol#964-965)	Medium
ControllerLib.redeem(Actions.RedeemArgs,MarginPoolInterface,OracleInterface,MarginCalculatorInterface,WhitelistInterface) (contracts/libs/ControllerLib.sol#317-343) ignores return value by (collateral,underlying,strike,expiry) = otoken.getOtokenDetails() (contracts/libs/ControllerLib.sol#329)	Medium
Controller.isSettlementAllowed(address) (contracts/core/Controller.sol#488-492) ignores return value by (collateral,underlying,strike,expiry) = OtokenInterface(_otoken).getOtokenDetails() (contracts/core/Controller.sol#489-490)	Medium

Finding	Impact
Controller.operate(Actions.ActionArgs[]) (contracts/core/Controller.sol#360-390) ignores return value by (isValidVault) = calculator.getExcessCollateral(_getTmpVault()) (contracts/core/Controller.sol#378)	Medium
ChainLinkPricer.getHistoricalPrice(uint80) (contracts/pricers/ChainlinkPricer.sol#95-98) ignores return value by (price,roundTimestamp) = aggregator.getRoundData(_roundId) (contracts/pricers/ChainlinkPricer.sol#96)	Medium
PayableProxyController.operate(Actions.ActionArgs[],address) (contracts/external/proxies/PayableProxyController.sol#51-85) ignores return value by controller.operate(_actions) (contracts/external/proxies/PayableProxyController.sol#75)	Medium
ControllerLib.settleVault(Actions.SettleVaultArgs,MarginPoolInterface,OracleInterface,MarginCalculatorInterface,MarginVault.Vault) (contracts/libs/ControllerLib.sol#350-411) ignores return value by (collateral,underlying,strike,None,expiry,isPut) = otoken.getOtokenDetails() (contracts/libs/ControllerLib.sol#385)	Medium
Controller.isLiquidatable(address,uint256) (contracts/core/Controller.sol#456-471) ignores return value by (isUnderCollat,totalDebt,payoutDebt,nakedDebt,dust) = ControllerLib.isLiquidatable(vault,calculator) (contracts/core/Controller.sol#468-469)	Medium
End of table for GammaProtocol	

core-v4

Slither results for core-v4	
Finding	**Impact**
Gmx2Hedger._sendCollateral(uint256) (contracts/core/hedgers/Gmx2Hedger.sol#659-676) uses arbitrary from in transferFrom: collateralToken.safeTransferFrom(hedgedPool,address(this),collateralAmount) (contracts/core/hedgers/Gmx2Hedger.sol#662-666)	High
TradeExecutor.executeActions(TradeExecutor.LocalVars,address) (contracts/core/TradeExecutor.sol#318-458) uses arbitrary from in transferFrom: vars.collateralAsset.safeTransferFrom(poolAddress,address(this),uint256(vars.traderCashflow)) (contracts/core/TradeExecutor.sol#360-364)	High

Finding	Impact
HedgedPool._redeemShares(address) (contracts/core/HedgedPool.sol#333-339) ignores return value by this.transfer(lpAddress,sharesAmount) (contracts/core/HedgedPool.sol#337)	High
HedgedPool.processOrder(ITradeExecutor.Order,address[],uint256,int256,int256) (contracts/core/HedgedPool.sol#484-526) uses a Boolean constant improperly: -notionalExposure[order.underlying][false] += exposureDiffCalls (contracts/core/HedgedPool.sol#504)	Medium
HedgedPool.getCashBuffer() (contracts/core/HedgedPool.sol#402-443) uses a Boolean constant improperly: -exposurePuts = notionalExposure[underlyingAsset][true] (contracts/core/HedgedPool.sol#409)	Medium
HedgedPool.getCashBuffer() (contracts/core/HedgedPool.sol#402-443) uses a Boolean constant improperly: -exposureCalls = notionalExposure[underlyingAsset][false] (contracts/core/HedgedPool.sol#408)	Medium
HedgedPool.configUnderlying(address,bool,uint256,uint256,uint256,uint256,uint256) (contracts/core/HedgedPool.sol#707-754) uses a Boolean constant improperly: -spotShockPercent[_underlying][false] = _spotShockPercentCalls (contracts/core/HedgedPool.sol#740)	Medium
HedgedPool.getCashBuffer() (contracts/core/HedgedPool.sol#402-443) uses a Boolean constant improperly: -cashBufferPuts = (((((uint256(- exposurePuts) * underlyingPrice) / (10 ** 8)) * spotShockPercent[underlyingAsset][true]) / 100 / CASH_BUFFER_LEVERAGE) * (10 ** numDecimals)) / (10 ** 8) (contracts/core/HedgedPool.sol#431-436)	Medium
HedgedPool.processOrder(ITradeExecutor.Order,address[],uint256,int256,int256) (contracts/core/HedgedPool.sol#484-526) uses a Boolean constant improperly: -notionalExposure[order.underlying][true] += exposureDiffPuts (contracts/core/HedgedPool.sol#505)	Medium
HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) uses a Boolean constant improperly: -notionalExposure[underlying][true] += Math.max(0,exposureAfterPuts - exposureBeforePuts) (contracts/core/HedgedPool.sol#200-203)	Medium

Finding	Impact
HedgedPool.getCashBuffer() (contracts/core/HedgedPool.sol#402-443) uses a Boolean constant improperly: -cashBufferCalls = (((((uint256(- exposureCalls) * underlyingPrice) / (10 ** 8)) * spotShockPercent[underlyingAsset][false]) / 100 / CASH_BUFFER_LEVERAGE) * (10 ** numDecimals)) / (10 ** 8) (contracts/core/HedgedPool.sol#418-423)	Medium
HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) uses a Boolean constant improperly: -notionalExposure[underlying][false] += Math.max(0,exposureAfterCalls - exposureBeforeCalls) (contracts/core/HedgedPool.sol#196-199)	Medium
HedgedPool.configUnderlying(address,bool,uint256,uint256,uint256,uint256,uint256) (contracts/core/HedgedPool.sol#707-754) uses a Boolean constant improperly: -spotShockPercent[_underlying][true] = _spotShockPercentPuts (contracts/core/HedgedPool.sol#741)	Medium
HedgedPool.getCashBuffer() (contracts/core/HedgedPool.sol#402-443) performs a multiplication on the result of a division: - cashBufferCalls = (((((uint256(- exposureCalls) * underlyingPrice) / (10 ** 8)) * spotShockPercent[underlyingAsset][false]) / 100 / CASH_BUFFER_LEVERAGE) * (10 ** numDecimals)) / (10 ** 8) (contracts/core/HedgedPool.sol#418-423)	Medium
HedgedPool.getCashBuffer() (contracts/core/HedgedPool.sol#402-443) performs a multiplication on the result of a division: - cashBufferPuts = (((((uint256(- exposurePuts) * underlyingPrice) / (10 ** 8)) * spotShockPercent[underlyingAsset][true]) / 100 / CASH_BUFFER_LEVERAGE) * (10 ** numDecimals)) / (10 ** 8) (contracts/core/HedgedPool.sol#431-436)	Medium
Gmx2Hedger (contracts/core/hedgers/Gmx2Hedger.sol#17-1116) has incorrect ERC20 function interface:Gmx2Hedger.transfer(address,uint256) (contracts/core/hedgers/Gmx2Hedger.sol#1109-1111)	Medium
HedgedPool._syncVaultMargin(uint256,OpynLib.MARGIN_UPDATE_TYPE) (contracts/core/HedgedPool.sol#669-692) uses a dangerous strict equality: - lastMarginUpdate[vaultId] == block.timestamp (contracts/core/HedgedPool.sol#674)	Medium

Finding	Impact
Reentrancy in HedgedPool._syncVaultMargin(uint256,OpynLib.MARGIN_UPDATE_TYPE) (contracts/core/HedgedPool.sol#669-692): External calls: - collateralChange = OpynLib.syncVaultMargin(controller,calculator,address(collateralToken),vaultId,updateType,getCollateralBalance(),MARGIN_HIGH_RANGE_PERCENT,MARGIN_LOW_RANGE_PERCENT) (contracts/core/HedgedPool.sol#676-685) State variables written after the call(s): - lastMarginUpdate[vaultId] = block.timestamp (contracts/core/HedgedPool.sol#688)HedgedPoolStorageV1.lastMarginUpdate (contracts/core/HedgedPoolStorage.sol#54) can be used in cross function reentrancies: - HedgedPool._syncVaultMargin(uint256,OpynLib.MARGIN_UPDATE_TYPE) (contracts/core/HedgedPool.sol#669-692)	Medium
Reentrancy in HedgedPool.setHedger(address,address) (contracts/core/HedgedPool.sol#623-633): External calls: - collateralToken.approve(address(hedgers[underlying]),0) (contracts/core/HedgedPool.sol#626) State variables written after the call(s): - hedgers[underlying] = hedger (contracts/core/HedgedPool.sol#629)HedgedPoolStorageV1.hedgers (contracts/core/HedgedPoolStorage.sol#31) can be used in cross function reentrancies: - HedgedPoolStorageV1.hedgers (contracts/core/HedgedPoolStorage.sol#31) - HedgedPool.setHedger(address,address) (contracts/core/HedgedPool.sol#623-633) - HedgedPool.syncMargin(address[]) (contracts/core/HedgedPool.sol#636-666)	Medium

Finding	Impact
Reentrancy in HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840): External calls: - collateralToken.approve(marginPool,0) (contracts/core/HedgedPool.sol#815) - collateralToken.approve(feeCollector,0) (contracts/core/HedgedPool.sol#818) - collateralToken.approve(tradeExecutor,0) (contracts/core/HedgedPool.sol#821) - IController(controller).setOperator(tradeExecutor,false) (contracts/core/HedgedPool.sol#822) State variables written after the call(s): - controller = addressBook.getController() (contracts/core/HedgedPool.sol#825)HedgedPoolStorageV1.controller (contracts/core/HedgedPoolStorage.sol#89) can be used in cross function reentrancies: - HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840) - HedgedPool._syncVaultMargin(uint256,OpynLib.MARGIN_UPDATE_TYPE) (contracts/core/HedgedPool.sol#669-692) - HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) - feeCollector = addressBook.getFeeCollector() (contracts/core/HedgedPool.sol#832)HedgedPoolStorageV1.feeCollector (contracts/core/HedgedPoolStorage.sol#96) can be used in cross function reentrancies: - HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840) - marginPool = addressBook.getMarginPool() (contracts/core/HedgedPool.sol#828)HedgedPoolStorageV1.marginPool (contracts/core/HedgedPoolStorage.sol#94) can be used in cross function reentrancies: - HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840) - tradeExecutor = addressBook.getTradeExecutor() (contracts/core/HedgedPool.sol#833)HedgedPoolStorageV1.tradeExecutor (contracts/core/HedgedPoolStorage.sol#97) can be used in cross function reentrancies: - HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840) - HedgedPool.configUnderlying(address,bool,uint256,uint256,uint256,uint256,uint256) (contracts/core/HedgedPool.sol#707-754) - HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) - HedgedPool.syncMargin(address[]) (contracts/core/HedgedPool.sol#636-666)	Medium

Finding	Impact
Reentrancy in HedgedPool._closeRound(uint256) (contracts/core/HedgedPool.sol#259-296): External calls: - sharesDiff -= int256(ILpManager(lpManager).closeWithdrawalRound(pricePerShare)) (contracts/core/HedgedPool.sol#269-271) State variables written after the call(s): - withdrawalRoundEnd += 604800 (contracts/core/HedgedPool.sol#273)HedgedPoolStorageV1.withdrawalRoundEnd (contracts/core/HedgedPoolStorage.sol#37) can be used in cross function reentrancies: - HedgedPool.__HedgedPool_init(address,address,address,string,string) (contracts/core/HedgedPool.sol#104-141) - HedgedPool._closeRound(uint256) (contracts/core/HedgedPool.sol#259-296) - HedgedPoolStorageV1.withdrawalRoundEnd (contracts/core/HedgedPoolStorage.sol#37)	Medium
Reentrancy in HedgedPool._closeRound(uint256) (contracts/core/HedgedPool.sol#259-296): External calls: - sharesDiff -= int256(ILpManager(lpManager).closeWithdrawalRound(pricePerShare)) (contracts/core/HedgedPool.sol#269-271) - sharesDiff += int256(ILpManager(lpManager).closeDepositRound(pricePerShare)) (contracts/core/HedgedPool.sol#279-281) State variables written after the call(s): - depositRoundEnd += 86400 (contracts/core/HedgedPool.sol#283)HedgedPoolStorageV1.depositRoundEnd (contracts/core/HedgedPoolStorage.sol#40) can be used in cross function reentrancies: - HedgedPool.__HedgedPool_init(address,address,address,string,string) (contracts/core/HedgedPool.sol#104-141) - HedgedPool._closeRound(uint256) (contracts/core/HedgedPool.sol#259-296) - HedgedPoolStorageV1.depositRoundEnd (contracts/core/HedgedPoolStorage.sol#40)	Medium

Finding	Impact
Reentrancy in HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229): External calls: - OpynLib.settle(controller,vaultId) (contracts/core/HedgedPool.sol#185) State variables written after the call(s): - lastSettledExpiry = expiry (contracts/core/HedgedPool.sol#217)HedgedPoolStorageV1.lastSettledExpiry (contracts/core/HedgedPoolStorage.sol#43) can be used in cross function reentrancies: - HedgedPool.__HedgedPool_init(address,address,address,string,string) (contracts/core/HedgedPool.sol#104-141) - HedgedPool._closeRound(uint256) (contracts/core/HedgedPool.sol#259-296) - HedgedPoolStorageV1.lastSettledExpiry (contracts/core/HedgedPoolStorage.sol#43) - HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) - notionalExposure[underlying][false] += Math.max(0,exposureAfterCalls - exposureBeforeCalls) (contracts/core/HedgedPool.sol#196-199)HedgedPoolStorageV1.notionalExposure (contracts/core/HedgedPoolStorage.sol#103) can be used in cross function reentrancies: - HedgedPool.getCashBuffer() (contracts/core/HedgedPool.sol#402-443) - HedgedPoolStorageV1.notionalExposure (contracts/core/HedgedPoolStorage.sol#103) - HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) - notionalExposure[underlying][true] += Math.max(0,exposureAfterPuts - exposureBeforePuts) (contracts/core/HedgedPool.sol#200-203)HedgedPoolStorageV1.notionalExposure (contracts/core/HedgedPoolStorage.sol#103) can be used in cross function reentrancies: - HedgedPool.getCashBuffer() (contracts/core/HedgedPool.sol#402-443) - HedgedPoolStorageV1.notionalExposure (contracts/core/HedgedPoolStorage.sol#103) - HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229)	Medium
HedgedPool._closeRound(uint256).sharesDiff (contracts/core/HedgedPool.sol#266) is a local variable never initialized	Medium

Finding	Impact
Gmx2Hedger.hedge(int256,uint256,uint256).currentLong (contracts/core/hedgers/Gmx2Hedger.sol#291) is a local variable never initialized	Medium
Gmx2Hedger._createOrderParamAddresses().swapPath (contracts/core/hedgers/Gmx2Hedger.sol#563) is a local variable never initialized	Medium
HedgedPool.getCashBuffer().underlyingPrice (contracts/core/HedgedPool.sol#407) is a local variable never initialized	Medium
Gmx2Hedger._changePosition(uint256,uint256,uint256,uint256,MarketUtils.MarketPrices).initialCollateralDelta (contracts/core/hedgers/Gmx2Hedger.sol#729) is a local variable never initialized	Medium
TradeExecutor.executeOrder(bytes32,address,ITradeExecutor.ExecuteOrderLeg[]).vars (contracts/core/TradeExecutor.sol#469) is a local variable never initialized	Medium
Gmx2Hedger.hedge(int256,uint256,uint256).currentShort (contracts/core/hedgers/Gmx2Hedger.sol#292) is a local variable never initialized	Medium
FeeCollector.collectFee(address,uint256,address).referrerAmount (contracts/core/FeeCollector.sol#55) is a local variable never initialized	Medium
Gmx2Hedger._orderParamAddresses().swapPath (contracts/core/hedgers/Gmx2Hedger.sol#580) is a local variable never initialized	Medium
HedgedPool.getCashBuffer().cashBufferPuts (contracts/core/HedgedPool.sol#412) is a local variable never initialized	Medium
Gmx2Hedger.hedge(int256,uint256,uint256).targetShort (contracts/core/hedgers/Gmx2Hedger.sol#294) is a local variable never initialized	Medium
Gmx2Hedger._syncDelta().totalSizeInTokens (contracts/core/hedgers/Gmx2Hedger.sol#465) is a local variable never initialized	Medium
Gmx2Hedger.hedge(int256,uint256,uint256).targetLong (contracts/core/hedgers/Gmx2Hedger.sol#293) is a local variable never initialized	Medium

Finding	Impact
Gmx2Hedger._changePosition(uint256,uint256,uint256,uint256,MarketUtils.MarketPrices).initialCollateralDelta_scope_0 (contracts/core/hedgers/Gmx2Hedger.sol#766) is a local variable never initialized	Medium
Gmx2Hedger._changePosition(uint256,uint256,uint256,uint256,MarketUtils.MarketPrices).vars (contracts/core/hedgers/Gmx2Hedger.sol#690) is a local variable never initialized	Medium
HedgedPool.getCashBuffer().cashBufferCalls (contracts/core/HedgedPool.sol#411) is a local variable never initialized	Medium
HedgedPool.configUnderlying(address,bool,uint256,uint256,uint256,uint256,uint256) (contracts/core/HedgedPool.sol#707-754) ignores return value by underlyingTokens.remove(_underlying) (contracts/core/HedgedPool.sol#744)	Medium
HedgedPool.withdrawCash() (contracts/core/HedgedPool.sol#361-367) ignores return value by (cashAmount) = ILpManager(lpManager).withdrawCash(msg.sender) (contracts/core/HedgedPool.sol#362)	Medium
TradeExecutor.submitOrder(ITradeExecutor.SubmitOrderParams,address,uint256,uint256,bool) (contracts/core/TradeExecutor.sol#168-235) ignores return value by orderKeys.add(orderKey) (contracts/core/TradeExecutor.sol#232)	Medium
TradeExecutor.executeActions(TradeExecutor.LocalVars,address) (contracts/core/TradeExecutor.sol#318-458) ignores return value by vars.controller.operate(actions1) (contracts/core/TradeExecutor.sol#455)	Medium
Gmx2Hedger._getPositinPnlCollateral(Position.Props,MarketUtils.MarketPrices) (contracts/core/hedgers/Gmx2Hedger.sol#529-552) ignores return value by (positionPnlUsd) = IReader(reader).getPositionPnlUsd(IDataStore(dataStore),marketProps,prices,key,0) (contracts/core/hedgers/Gmx2Hedger.sol#538-544)	Medium
HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840) ignores return value by collateralToken.approve(feeCollector,type()(uint256).max) (contracts/core/HedgedPool.sol#838)	Medium

Finding	Impact
HedgedPool.configUnderlying(address,bool,uint256,uint256,uint256,uint256,uint256) (contracts/core/HedgedPool.sol#707-754) ignores return value by ITradeExecutor(tradeExecutor).openMarginVault(_underlying,0,address(0)) (contracts/core/HedgedPool.sol#723-727)	Medium
TradeExecutor.executeActions(TradeExecutor.LocalVars,address) (contracts/core/TradeExecutor.sol#318-458) ignores return value by IERC20(vars.oTokens[i]).approve(vars.marginPoolAddress,uint256(vars.order.legs[i].amount)) (contracts/core/TradeExecutor.sol#407-410)	Medium
HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) ignores return value by activeOTokens.remove(oTokensByExpiry[underlying][expiry][io]) (contracts/core/HedgedPool.sol#211-213)	Medium
HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840) ignores return value by collateralToken.approve(tradeExecutor,0) (contracts/core/HedgedPool.sol#821)	Medium
TradeExecutor.openMarginVault(address,uint256,address) (contracts/core/TradeExecutor.sol#644-676) ignores return value by controller.operate(actions) (contracts/core/TradeExecutor.sol#672)	Medium
TradeExecutor.deleteOrder(bytes32,address) (contracts/core/TradeExecutor.sol#275-279) ignores return value by orderKeys.remove(orderKey) (contracts/core/TradeExecutor.sol#277)	Medium
HedgedPool.processOToken(address,uint256,uint256,address,uint256) (contracts/core/HedgedPool.sol#558-616) ignores return value by activeOTokens.add(oToken) (contracts/core/HedgedPool.sol#614)	Medium
TradeExecutor.deleteOrder(bytes32,address) (contracts/core/TradeExecutor.sol#275-279) ignores return value by accountOrderKeys[account].remove(orderKey) (contracts/core/TradeExecutor.sol#278)	Medium
Gmx2Hedger._sendCollateral(uint256) (contracts/core/hedgers/Gmx2Hedger.sol#659-676) ignores return value by IERC20(address(collateralToken)).approve(IExchangeRouter(exchangeRouter).router(),collateralAmount) (contracts/core/hedgers/Gmx2Hedger.sol#667-670)	Medium
HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840) ignores return value by collateralToken.approve(marginPool,0) (contracts/core/HedgedPool.sol#815)	Medium

Finding	Impact
HedgedPool.balanceOf(address) (contracts/core/HedgedPool.sol#453-461) ignores return value by (sharesRedeemable) = ILpManager(lpManager).getDepositStatus(address (this),account) (contracts/core/HedgedPool.sol#456-459)	Medium
TradeExecutor._openMarginVault(address,address) (contracts/core/TradeExecutor.sol#678-706) ignores return value by controller.operate(actions) (contracts/core/TradeExecutor.sol#699)	Medium
TradeExecutor.submitOrder(ITradeExecutor.SubmitOrderParams,address, uint256,uint256,bool) (contracts/core/TradeExecutor.sol#168-235) ignores return value by controller.operate(actions) (contracts/core/TradeExecutor.sol#215)	Medium
HedgedPool.configUnderlying(address,bool,uint256,uint256,uint256,uint256,uint256) (contracts/core/HedgedPool.sol#707-754) ignores return value by underlyingTokens.add(_underlying) (contracts/core/HedgedPool.sol#721)	Medium
TradeExecutor.executeActions(TradeExecutor.LocalVars,address) (contracts/core/TradeExecutor.sol#318-458) ignores return value by vars.controller.operate(actions2) (contracts/core/TradeExecutor.sol#456)	Medium
HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840) ignores return value by collateralToken.approve(feeCollector,0) (contracts/core/HedgedPool.sol#818)	Medium
HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) ignores return value by OpynLib.settle(controller,vaultId) (contracts/core/HedgedPool.sol#185)	Medium
Gmx2Hedger._calculateAdjustedExecutionPrice(Position.Props,MarketUtils.MarketPrices,bool,uint256,uint256,uint256,Order.OrderType) (contracts/core/hedgers/Gmx2Hedger.sol#985-1021) ignores return value by (executionPrice) = IGmxUtils(gmxUtils).getExecutionPrice(updateOrderParams,prices.indexTokenPrice) (contracts/core/hedgers/Gmx2Hedger.sol#1015-1018)	Medium
HedgedPool.setHedger(address,address) (contracts/core/HedgedPool.sol#623-633) ignores return value by collateralToken.approve(address(hedgers[underlying]),0) (contracts/core/HedgedPool.sol#626)	Medium

Finding	Impact
HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840) ignores return value by collateralToken.approve(marginPool,type()(uint256).max) (contracts/core/HedgedPool.sol#837)	Medium
TradeExecutor.executeActions(TradeExecutor.LocalVars,address) (contracts/core/TradeExecutor.sol#318-458) ignores return value by IERC20(vars.oTokens[i]).approve(vars.marginPoolAddress,uint256(-vars.order.legs[i].amount)) (contracts/core/TradeExecutor.sol#436-439)	Medium
TradeExecutor.submitOrder(ITradeExecutor.SubmitOrderParams,address, uint256,uint256,bool) (contracts/core/TradeExecutor.sol#168-235) ignores return value by accountOrderKeys[msg.sender].add(orderKey) (contracts/core/TradeExecutor.sol#233)	Medium
TradeExecutor.executeActions(TradeExecutor.LocalVars,address) (contracts/core/TradeExecutor.sol#318-458) ignores return value by vars.controller.operate(actions3) (contracts/core/TradeExecutor.sol#457)	Medium
HedgedPool.setHedger(address,address) (contracts/core/HedgedPool.sol#623-633) ignores return value by collateralToken.approve(hedger,type()(uint256).max) (contracts/core/HedgedPool.sol#630)	Medium
HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840) ignores return value by collateralToken.approve(tradeExecutor,type()(uint256).max) (contracts/core/HedgedPool.sol#839)	Medium
HedgedPool.updateSeriesPerExpirationLimit(uint256) (contracts/core/HedgedPool.sol#551-555) should emit an event for: - seriesPerExpirationLimit = _seriesPerExpirationLimit (contracts/core/HedgedPool.sol#554)	Low
Gmx2Hedger.__Gmx2Hedger_init(address,address,address,address,address,uint256)._hedgedPool (contracts/core/hedgers/Gmx2Hedger.sol#109) lacks a zero-check on : - hedgedPool = _hedgedPool (contracts/core/hedgers/Gmx2Hedger.sol#121)	Low
Gmx2Hedger.transfer(address,uint256).to (contracts/core/hedgers/Gmx2Hedger.sol#1109) lacks a zero-check on : - address(to).transfer(amount) (contracts/core/hedgers/Gmx2Hedger.sol#1110)	Low

Finding	Impact
Gmx2Hedger.__Gmx2Hedger_init(address,address,address,address,address,uint256)._market (contracts/core/hedgers/Gmx2Hedger.sol#112) lacks a zero-check on : - market = _market (contracts/core/hedgers/Gmx2Hedger.sol#116)	Low
HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) has external calls inside a loop: (exposureAfterCalls,exposureAfterPuts) = IController(controller).getVaultExposure(address(this),vaultId) (contracts/core/HedgedPool.sol#187-193)	Low
Gmx2Hedger._getPositinPnlCollateral(Position.Props,MarketUtils.MarketPrices) (contracts/core/hedgers/Gmx2Hedger.sol#529-552) has external calls inside a loop: (positionPnlUsd) = IReader(reader).getPositionPnlUsd(IDataStore(dataStore),marketProps,prices,key,0) (contracts/core/hedgers/Gmx2Hedger.sol#538-544)	Low
HedgedPool.getCollateralBalance() (contracts/core/HedgedPool.sol#396-400) has external calls inside a loop: collateralToken.balanceOf(address(this)) - ILpManager(lpManager).getCashLocked(address(this),true) (contracts/core/HedgedPool.sol#397-399)	Low
HedgedPool.syncMargin(address[]) (contracts/core/HedgedPool.sol#636-666) has external calls inside a loop: vaultId = ITradeExecutor(tradeExecutor).marginVaults(address(this),underlyingTokens.at(i),0) (contracts/core/HedgedPool.sol#645-649)	Low
TradeExecutor.executeActions(TradeExecutor.LocalVars,address) (contracts/core/TradeExecutor.sol#318-458) has external calls inside a loop: IERC20(vars.oTokens[i]).approve(vars.marginPoolAddress,uint256(- vars.order.legs[i].amount)) (contracts/core/TradeExecutor.sol#436-439)	Low
HedgedPool.syncMargin(address[]) (contracts/core/HedgedPool.sol#636-666) has external calls inside a loop: collateralMoved += Math.abs(IHedger(hedgers[underlying[i]]).sync()) (contracts/core/HedgedPool.sol#655)	Low
HedgedPool.getCashBuffer() (contracts/core/HedgedPool.sol#402-443) has external calls inside a loop: underlyingPrice = IOracle(oracle).getPrice(underlyingAsset) (contracts/core/HedgedPool.sol#429)	Low

Finding	Impact
TradeExecutor.executeActions(TradeExecutor.LocalVars,address) (contracts/core/TradeExecutor.sol#318-458) has external calls inside a loop: IERC20(vars.oTokens[i]).approve(vars.marginPoolAddress,uint256(vars.order.legs[i].amount)) (contracts/core/TradeExecutor.sol#407-410)	Low
HedgedPool.getCashBuffer() (contracts/core/HedgedPool.sol#402-443) has external calls inside a loop: underlyingPrice = IOracle(oracle).getPrice(underlyingAsset) (contracts/core/HedgedPool.sol#417)	Low
TradeExecutor.executeOrder(bytes32,address,ITradeExecutor.ExecuteOrderLeg[]) (contracts/core/TradeExecutor.sol#460-623) has external calls inside a loop: vars.poolMargin += (vars.calculator.getNakedMarginRequired(vars.order.underlying,address(vars.strikeAsset),address(vars.collateralAsset),uint256(leg.amount),leg.strike,vars.underlyingPrice,leg.expiration,vars.collateralDecimals,leg.isPut) * MARGIN_BUFFER_PERCENT) / 100 (contracts/core/TradeExecutor.sol#519-531)	Low
HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) has external calls inside a loop: (exposureBeforeCalls,exposureBeforePuts) = IController(controller).getVaultExposure(address(this),vaultId) (contracts/core/HedgedPool.sol#177-183)	Low
HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) has external calls inside a loop: vaultId = ITradeExecutor(tradeExecutor).marginVaults(address(this),underlying,0) (contracts/core/HedgedPool.sol#172-176)	Low

Finding	Impact
Reentrancy in HedgedPool._refreshConfigInternal() (contracts/core/HedgedPool.sol#812-840): External calls: - collateralToken.approve(marginPool,0) (contracts/core/HedgedPool.sol#815) - collateralToken.approve(feeCollector,0) (contracts/core/HedgedPool.sol#818) - collateralToken.approve(tradeExecutor,0) (contracts/core/HedgedPool.sol#821) - IController(controller).setOperator(tradeExecutor,false) (contracts/core/HedgedPool.sol#822) State variables written after the call(s): - calculator = addressBook.getMarginCalculator() (contracts/core/HedgedPool.sol#826) - lpManager = addressBook.getLpManager() (contracts/core/HedgedPool.sol#831) - oTokenFactory = addressBook.getOtokenFactory() (contracts/core/HedgedPool.sol#829) - oracle = addressBook.getOracle() (contracts/core/HedgedPool.sol#827) - orderUtil = addressBook.getOrderUtil() (contracts/core/HedgedPool.sol#830)	Low
Reentrancy in HedgedPool._closeRound(uint256) (contracts/core/HedgedPool.sol#259-296): External calls: - sharesDiff -= int256(ILpManager(lpManager).closeWithdrawalRound(pricePerShare)) (contracts/core/HedgedPool.sol#269-271) State variables written after the call(s): - pricePerShareCached = pricePerShare (contracts/core/HedgedPool.sol#275)	Low
Reentrancy in TradeExecutor._openMarginVault(address,address) (contracts/core/TradeExecutor.sol#678-706): External calls: - controller.operate(actions) (contracts/core/TradeExecutor.sol#699) State variables written after the call(s): - marginVaults[_owner][_underlyingAsset].push(vaultId) (contracts/core/TradeExecutor.sol#701)	Low

Finding	Impact
Reentrancy in TradeExecutor.submitOrder(ITradeExecutor.SubmitOrderParams,address,uint256,uint256,bool) (contracts/core/TradeExecutor.sol#168-235): External calls: - traderVaultId = _openMarginVault(msg.sender,params.underlying) (contracts/core/TradeExecutor.sol#189) - controller.operate(actions) (contracts/core/TradeExecutor.sol#699) - IERC20(collateralAsset).safeTransferFrom(msg.sender,address(this),traderDeposit) (contracts/core/TradeExecutor.sol#193-197) - IERC20(collateralAsset).safeApprove(addressBook.getMarginPool(),uint256(traderDeposit)) (contracts/core/TradeExecutor.sol#198-201) - controller.operate(actions) (contracts/core/TradeExecutor.sol#215) State variables written after the call(s): - orderKey = getNextKey() (contracts/core/TradeExecutor.sol#217) - nonce ++ (contracts/core/TradeExecutor.sol#239) - order.account = msg.sender (contracts/core/TradeExecutor.sol#220) - order.underlying = params.underlying (contracts/core/TradeExecutor.sol#221) - order.referrer = params.referrer (contracts/core/TradeExecutor.sol#222) - order.validUntil = params.validUntil (contracts/core/TradeExecutor.sol#223) - order.premiumLimit = params.premiumLimit (contracts/core/TradeExecutor.sol#224) - order.executionFee = msg.value (contracts/core/TradeExecutor.sol#225) - order.traderVaultId = traderVaultId (contracts/core/TradeExecutor.sol#226) - order.callbackReceiver = params.callbackReceiver (contracts/core/TradeExecutor.sol#227) - order.legs.push(params.legs[i]) (contracts/core/TradeExecutor.sol#230)	Low

Finding	Impact
Reentrancy in Gmx2Hedger._gmxPositionDecrease(bool,uint256,uint256,uint256,uint256) (contracts/core/hedgers/Gmx2Hedger.sol#857-896): External calls: - IExchangeRouter(exchangeRouter).sendWnt{value: executionFee}(orderVault,executionFee) (contracts/core/hedgers/Gmx2Hedger.sol#877-880) - key = _createMarketOrder(createOrderParamAddresses,createOrderParamNumbers,Order.OrderType.MarketDecrease,Order.DecreasePositionSwapType.SwapPnlTokenToCollateralToken,isLong,false,0) (contracts/core/hedgers/Gmx2Hedger.sol#882-890) - key = IExchangeRouter(exchangeRouter).createOrder(IOrderUtils.CreateOrderParams(createOrderParamAddresses,createOrderParamNumbers,orderType,decreasePositionSwap,isLong,shouldUnwrapNativeToken,referralCode)) (contracts/core/hedgers/Gmx2Hedger.sol#645-655) External calls sending eth: - IExchangeRouter(exchangeRouter).sendWnt{value: executionFee}(orderVault,executionFee) (contracts/core/hedgers/Gmx2Hedger.sol#877-880) State variables written after the call(s): - pendingOrders[key] = true (contracts/core/hedgers/Gmx2Hedger.sol#892) - pendingOrdersCount = pendingOrdersCount + 1 (contracts/core/hedgers/Gmx2Hedger.sol#893)	Low
Reentrancy in HedgedPool.processOrder(ITradeExecutor.Order,address[],uint256,int256,int256) (contracts/core/HedgedPool.sol#484-526): External calls: - IFeeCollector(feeCollector).collectFee(address(collateralToken),fee,order.referrer) (contracts/core/HedgedPool.sol#496-500) State variables written after the call(s): - notionalExposure[order.underlying][false] += exposureDiffCalls (contracts/core/HedgedPool.sol#504) - notionalExposure[order.underlying][true] += exposureDiffPuts (contracts/core/HedgedPool.sol#505) - processOToken(order.underlying,leg.strike,leg.expiration,oTokens[i],underlyingPrice) (contracts/core/HedgedPool.sol#518-524) - oTokensByExpiry[underlying][expiry].push(oToken) (contracts/core/HedgedPool.sol#615)	Low

Finding	Impact
Reentrancy in Gmx2Hedger.afterOrderCancellation(bytes32,Order.Props ,EventUtils.EventLogData) (contracts/core/hedgers/Gmx2Hedger.sol#218-242): External calls: - collateralToken.safeTransfer(hedgedPool,collateralToken.balanceOf (address(this))) (contracts/core/hedgers/Gmx2Hedger.sol#232-235) State variables written after the call(s): - _syncDelta() (contracts/core/hedgers/Gmx2Hedger.sol#238) - deltaCached = (totalSizeInTokens * int256(10 ** SIREN_DECIMALS)) / int256(10 ** underlyingDecimals) (contracts/core/hedgers/Gmx2Hedger.sol#490-492)	Low
Reentrancy in HedgedPool.configUnderlying(address,bool,uint256,uint 256,uint256,uint256,uint256) (contracts/core/HedgedPool.sol#707-754): External calls: - ITradeExecutor(tradeExecutor).openMarginVault(_underlying,0,addre ss(0)) (contracts/core/HedgedPool.sol#723-727) State variables written after the call(s): - allowedStrikeRanges[_underlying] = TokenStrikeRange(_minPercent,_maxPercent,_increment) (contracts/core/HedgedPool.sol#734-738) - spotShockPercent[_underlying][false] = _spotShockPercentCalls (contracts/core/HedgedPool.sol#740) - spotShockPercent[_underlying][true] = _spotShockPercentPuts (contracts/core/HedgedPool.sol#741)	Low

Finding	Impact
Reentrancy in Gmx2Hedger._gmxWithdrawCollateral(bool,uint256,uint256) (contracts/core/hedgers/Gmx2Hedger.sol#943-983): External calls: - IExchangeRouter(exchangeRouter).sendWnt{value: executionFee}(orderVault,executionFee) (contracts/core/hedgers/Gmx2Hedger.sol#964-967) - key = _createMarketOrder(createOrderParamAddresses,createOrderParamNumbers,Order.OrderType.MarketDecrease,Order.DecreasePositionSwapType.SwapPnlTokenToCollateralToken,isLong,false,0) (contracts/core/hedgers/Gmx2Hedger.sol#969-977) - key = IExchangeRouter(exchangeRouter).createOrder(IOrderUtils.CreateOrderParams(createOrderParamAddresses,createOrderParamNumbers,orderType,decreasePositionSwap,isLong,shouldUnwrapNativeToken,referralCode)) (contracts/core/hedgers/Gmx2Hedger.sol#645-655) External calls sending eth: - IExchangeRouter(exchangeRouter).sendWnt{value: executionFee}(orderVault,executionFee) (contracts/core/hedgers/Gmx2Hedger.sol#964-967) State variables written after the call(s): - pendingOrders[key] = true (contracts/core/hedgers/Gmx2Hedger.sol#979) - pendingOrdersCount = pendingOrdersCount + 1 (contracts/core/hedgers/Gmx2Hedger.sol#980)	Low
Reentrancy in FeeCollector._withdrawFee(address,address) (contracts/core/FeeCollector.sol#89-99): External calls: - IERC20(feeAsset).safeTransfer(msg.sender,feeAmount) (contracts/core/FeeCollector.sol#96) Event emitted after the call(s): - FeeWithdrawn(referrer,feeAsset,feeAmount) (contracts/core/FeeCollector.sol#98)	Low

Finding	Impact
Reentrancy in HedgedPool.__HedgedPool_init(address,address,address,string,string) (contracts/core/HedgedPool.sol#104-141): External calls: - _refreshConfigInternal() (contracts/core/HedgedPool.sol#133) - collateralToken.approve(marginPool,0) (contracts/core/HedgedPool.sol#815) - collateralToken.approve(feeCollector,0) (contracts/core/HedgedPool.sol#818) - collateralToken.approve(tradeExecutor,0) (contracts/core/HedgedPool.sol#821) - IController(controller).setOperator(tradeExecutor,false) (contracts/core/HedgedPool.sol#822) - IController(controller).setOperator(tradeExecutor,true) (contracts/core/HedgedPool.sol#834) - collateralToken.approve(marginPool,type()(uint256).max) (contracts/core/HedgedPool.sol#837) - collateralToken.approve(feeCollector,type()(uint256).max) (contracts/core/HedgedPool.sol#838) - collateralToken.approve(tradeExecutor,type()(uint256).max) (contracts/core/HedgedPool.sol#839) Event emitted after the call(s): - HedgedPoolInitialized(_strikeToken,_collateralToken,_tokenName,_tokenSymbol) (contracts/core/HedgedPool.sol#135-140)	Low
Reentrancy in HedgedPool.setHedger(address,address) (contracts/core/HedgedPool.sol#623-633): External calls: - collateralToken.approve(address(hedgers[underlying]),0) (contracts/core/HedgedPool.sol#626) - collateralToken.approve(hedger,type()(uint256).max) (contracts/core/HedgedPool.sol#630) Event emitted after the call(s): - HedgerSet(underlying,hedger) (contracts/core/HedgedPool.sol#632)	Low

Finding	Impact
Reentrancy in Gmx2Hedger._gmxPositionDecrease(bool,uint256,uint256,uint256,uint256) (contracts/core/hedgers/Gmx2Hedger.sol#857-896): External calls: - IExchangeRouter(exchangeRouter).sendWnt{value: executionFee}(orderVault,executionFee) (contracts/core/hedgers/Gmx2Hedger.sol#877-880) - key = _createMarketOrder(createOrderParamAddresses,createOrderParamNumbers,Order.OrderType.MarketDecrease,Order.DecreasePositionSwapType.SwapPnlTokenToCollateralToken,isLong,false,0) (contracts/core/hedgers/Gmx2Hedger.sol#882-890) - key = IExchangeRouter(exchangeRouter).createOrder(IOrderUtils.CreateOrderParams(createOrderParamAddresses,createOrderParamNumbers,orderType,decreasePositionSwap,isLong,shouldUnwrapNativeToken,referralCode)) (contracts/core/hedgers/Gmx2Hedger.sol#645-655) External calls sending eth: - IExchangeRouter(exchangeRouter).sendWnt{value: executionFee}(orderVault,executionFee) (contracts/core/hedgers/Gmx2Hedger.sol#877-880) Event emitted after the call(s): - GmxDecreaseOrderCreated(key,isLong) (contracts/core/hedgers/Gmx2Hedger.sol#895)	Low
Reentrancy in TradeExecutor.submitOrder(ITradeExecutor.SubmitOrderParams,address,uint256,uint256,bool) (contracts/core/TradeExecutor.sol#168-235): External calls: - traderVaultId = _openMarginVault(msg.sender,params.underlying) (contracts/core/TradeExecutor.sol#189) - controller.operate(actions) (contracts/core/TradeExecutor.sol#699) - IERC20(collateralAsset).safeTransferFrom(msg.sender,address(this),traderDeposit) (contracts/core/TradeExecutor.sol#193-197) - IERC20(collateralAsset).safeApprove(addressBook.getMarginPool(),uint256(traderDeposit)) (contracts/core/TradeExecutor.sol#198-201) - controller.operate(actions) (contracts/core/TradeExecutor.sol#215) Event emitted after the call(s): - OrderCreated(orderKey,order) (contracts/core/TradeExecutor.sol#234)	Low

Finding	Impact
Reentrancy in TradeExecutor._openMarginVault(address,address) (contracts/core/TradeExecutor.sol#678-706): External calls: - controller.operate(actions) (contracts/core/TradeExecutor.sol#699) Event emitted after the call(s): - MarginVaultOpened(_owner,_underlyingAsset,vaultId) (contracts/core/TradeExecutor.sol#703)	Low
Reentrancy in Gmx2Hedger.afterOrderCancellation(bytes32,Order.Props ,EventUtils.EventLogData) (contracts/core/hedgers/Gmx2Hedger.sol#218-242): External calls: - collateralToken.safeTransfer(hedgedPool,collateralToken.balanceOf (address(this))) (contracts/core/hedgers/Gmx2Hedger.sol#232-235) Event emitted after the call(s): - GmxOrderCanceled(key) (contracts/core/hedgers/Gmx2Hedger.sol#241)	Low
Reentrancy in FeeCollector.collectFee(address,uint256,address) (contracts/core/FeeCollector.sol#50-76): External calls: - IERC20(feeAsset).safeTransferFrom(msg.sender,address(this),feeAmo unt) (contracts/core/FeeCollector.sol#66) Event emitted after the call(s): - FeeCollected(msg.sender,referrer,feeAsset,feeAmount,referrerAmoun t) (contracts/core/FeeCollector.sol#69-75)	Low

Finding	Impact
Reentrancy in Gmx2Hedger._gmxWithdrawCollateral(bool,uint256,uint256) (contracts/core/hedgers/Gmx2Hedger.sol#943-983): External calls: - IExchangeRouter(exchangeRouter).sendWnt{value: executionFee}(orderVault,executionFee) (contracts/core/hedgers/Gmx2Hedger.sol#964-967) - key = _createMarketOrder(createOrderParamAddresses,createOrderParamNumbers,Order.OrderType.MarketDecrease,Order.DecreasePositionSwapType.SwapPnlTokenToCollateralToken,isLong,false,0) (contracts/core/hedgers/Gmx2Hedger.sol#969-977) - key = IExchangeRouter(exchangeRouter).createOrder(IOrderUtils.CreateOrderParams(createOrderParamAddresses,createOrderParamNumbers,orderType,decreasePositionSwap,isLong,shouldUnwrapNativeToken,referralCode)) (contracts/core/hedgers/Gmx2Hedger.sol#645-655) External calls sending eth: - IExchangeRouter(exchangeRouter).sendWnt{value: executionFee}(orderVault,executionFee) (contracts/core/hedgers/Gmx2Hedger.sol#964-967) Event emitted after the call(s): - GmxWithdrawlOrderCreated(key,isLong) (contracts/core/hedgers/Gmx2Hedger.sol#982)	Low
Reentrancy in HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229): External calls: - OpynLib.settle(controller,vaultId) (contracts/core/HedgedPool.sol#185) Event emitted after the call(s): - ExpirySettled(expiry) (contracts/core/HedgedPool.sol#219)	Low
Reentrancy in HedgedPool.configUnderlying(address,bool,uint256,uint256,uint256,uint256,uint256) (contracts/core/HedgedPool.sol#707-754): External calls: - ITradeExecutor(tradeExecutor).openMarginVault(_underlying,0,address(0)) (contracts/core/HedgedPool.sol#723-727) Event emitted after the call(s): - UnderlyingConfigured(_underlying,_enabled,_minPercent,_maxPercent,_increment) (contracts/core/HedgedPool.sol#747-753)	Low

Finding	Impact
TradeExecutor.validateExpirations(uint256,ITradeExecutor.OptionLeg[]) (contracts/core/TradeExecutor.sol#146-161) uses timestamp for comparisons Dangerous comparisons: - validUntil <= block.timestamp (contracts/core/TradeExecutor.sol#150) - leg.expiration <= block.timestamp (contracts/core/TradeExecutor.sol#156)	Low
HedgedPool._closeRound(uint256) (contracts/core/HedgedPool.sol#259-296) uses timestamp for comparisons Dangerous comparisons: - lastSettledExpiry + 604800 < block.timestamp (contracts/core/HedgedPool.sol#261) - withdrawalRoundEnd <= block.timestamp (contracts/core/HedgedPool.sol#268) - depositRoundEnd <= block.timestamp (contracts/core/HedgedPool.sol#278)	Low
HedgedPool.settleAll() (contracts/core/HedgedPool.sol#160-229) uses timestamp for comparisons Dangerous comparisons: - expiry <= block.timestamp (contracts/core/HedgedPool.sol#168)	Low
HedgedPool._syncVaultMargin(uint256,OpynLib.MARGIN_UPDATE_TYPE) (contracts/core/HedgedPool.sol#669-692) uses timestamp for comparisons Dangerous comparisons: - lastMarginUpdate[vaultId] == block.timestamp (contracts/core/HedgedPool.sol#674)	Low
End of table for core-v4	

- As a result of the tests carried out with the Slither tool, some results were obtained and reviewed by Halborn. Based on the results reviewed, some vulnerabilities were determined to be false positives.

THANK YOU FOR CHOOSING

// HALBORN