// Private Smart Contract Security Assessment 02.12.2024 - 03.06.2024

Sustainable Leverage
Archimedes Finance

HALBORN

Sustainable Leverage - Archimedes Finance

Prepared by: HALBORN

Last Updated 04/25/2024

Date of Engagement by: February 12th, 2024 - March 6th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
19	1	2	3	8	5

TABLE OF CONTENTS

1. Introduction

Archimedes engaged our team to conduct a comprehensive security assessment of their smart contract ecosystem. This assessment was initiated to meticulously evaluate the security posture of the smart contracts as presented to us. Our assessment aimed to identify potential vulnerabilities, assess the overall security mechanisms in place, and provide actionable recommendations to enhance the security and functionality of the Archimedes smart contract infrastructure. The scope of this assessment was strictly limited to the smart contracts submitted for our review, ensuring a focused and detailed analysis of the contract's security aspects.

2. Assessment Summary

Halborn was provided about one week 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 `Archimedes Finance team`.

During the assessment, the following components and functionalities were scrutinized:

Smart Contract Versioning And Pragma Directives

- Use of floating pragma directives across contracts, potentially leading to compilation with untested Solidity versions.

Access Control And Security Mechanisms

- Implementation and enforcement of role-based access controls, particularly in sensitive functions like token swaps and vault operations.
- Absence of access control in critical functions, allowing unauthorized interactions.

Contract Initialization And Parameter Configuration

- Initialization patterns in upgradable contracts and the setting of crucial protocol parameters.
- Configuration and validation of strategy-specific parameters, including position lifetime and liquidation thresholds.

Swap Mechanisms And External Interactions

- Token swap logic within the UniV3SwapAdapter and its reliance on external protocols for execution.

- Validation of external contract addresses and parameters in swap functions to prevent interactions with uninitialized or malicious contracts.

Token Handling And Vault Mechanics

- Management of WBTC tokens within vault contracts, including borrowing, repayment, and fee collection mechanisms.
- Verification of token transfer operations to ensure they are conducted securely and with proper authorization.

Strategy Management And Position Handling

- Logic for setting, updating, and removing strategy configurations, emphasizing the impact on position management and liquidation.
- Handling of positions within the ledger, focusing on state transitions and the calculation of claimable amounts.

Error Handling And Input Validation

- Robustness of error handling mechanisms, including the use of custom errors for clarity and security.
- Validation checks for critical inputs and configurations to prevent invalid states or operations.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding 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).

4. 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.

4.1 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:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 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) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

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

4.3 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.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 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

SEVERITY	SCORE VALUE RANGE
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: SustainableLeverage

(b) Assessed Commit ID: cd55423

(c) Items in scope:

- src/internal/LeveragedStrategy.sol
- src/user_facing/PositionOpener.sol
- src/internal/PositionLedger.sol
- src/user_facing/PositionCloser.sol
- src/libs/ClosePositionInternal.sol
- src/user_facing/ExpiredVault.sol
- src/monitor_facing/base/ClosePositionBase.sol
- src/internal/LeverageDepositor.sol
- src/monitor_facing/PositionLiquidator.sol
- src/ports/swap_adapters/UniV3SwapAdapter.sol
- src/internal/ProtocolParameters.sol
- src/internal/OracleManager.sol
- src/libs/EventsLeverageEngine.sol
- src/user_facing/PositionToken.sol
- src/internal/SwapManager.sol
- src/monitor_facing/PositionExpirator.sol
- src/libs/ErrorsLeverageEngine.sol
- src/internal/WBTCVault.sol
- src/libs/PositionCallParams.sol
- src/libs/DependencyAddresses.sol
- src/ports/oracles/ChainlinkOracle.sol
- src/libs/ProtocolRoles.sol
- src/libs/Constants.sol
- thisisarchimedes/SustainableLeverage/pull/87/files

Out-of-Scope:

REMEDIATION COMMIT ID:

- 3ffd5223ffd522
- 3f5d9d63f5d9d6

- e2ef9e2e2ef9e2
- dc84060dc84060
- 01d0d1901d0d19
- e7f363ce7f363c
- 06084e206084e2
- 69f8cf469f8cf4
- e76b378e76b378
- 38a2e4d38a2e4d
- 9bf2cb39bf2cb3
- 56221da56221da
- a8f21c0a8f21c0
- ddcea5addcea5a

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	2	3	8	5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LACK OF ACCESS CONTROL IN WBTCVAULT CONTRACT	CRITICAL	SOLVED - 03/13/2024
UNRESTRICTED ACCESS TO SWAP FUNCTIONS	HIGH	SOLVED - 03/13/2024
MISSING ORACLE VALIDATION IN PRICE RETRIEVAL	HIGH	SOLVED - 03/13/2024
UNPROTECTED INITIALIZE FUNCTION IN UPGRADEABLE CONTRACTS	MEDIUM	SOLVED - 03/13/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
EXCESSIVE USE OF APPROVE WITH MAXIMUM ALLOWANCE	MEDIUM	RISK ACCEPTED
INCONSISTENT MINIMUM POSITION DURATION LOGIC	MEDIUM	SOLVED - 03/13/2024
UNVERIFIED SWAP ADAPTER ADDRESS	LOW	SOLVED - 03/13/2024
MISMATCHED STRATEGY POSITION LIFETIME AND MINIMUM POSITION DURATION	LOW	SOLVED - 03/13/2024
UNRESTRICTED POSITION ENTRY OVERWRITE	LOW	SOLVED - 03/13/2024
UNRESTRICTED STATE MODIFICATION OF NON-EXISTENT POSITIONS	LOW	SOLVED - 03/13/2024
GETTER FUNCTIONS WITHOUT EXISTENCE CHECK	LOW	NOT APPLICABLE
UNRESTRICTED EXIT FEE SETTING	LOW	SOLVED - 03/13/2024
LIQUIDATION FEE TRANSFER TO UNSET ADDRESS	LOW	RISK ACCEPTED
IMPROPER BALANCE VERIFICATION IN SWAPTOLVBTC	LOW	RISK ACCEPTED

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
REDUNDANT STATE VARIABLE ASSIGNMENT	INFORMATIONAL	SOLVED - 03/13/2024
REDUNDANT DATA ACCESS FUNCTION	INFORMATIONAL	SOLVED - 03/13/2024
INCOMPLETE IMPLEMENTATION AND REDUNDANT CODE	INFORMATIONAL	SOLVED - 03/13/2024
TYPO IN LEDGERENTRY STRUCTURE FIELD NAME	INFORMATIONAL	SOLVED - 03/13/2024
FLOATING PRAGMA IN SOLIDITY CONTRACTS	INFORMATIONAL	ACKNOWLEDGED

7. FINDINGS & TECH DETAILS

7.1 LACK OF ACCESS CONTROL IN WBTCVAULT CONTRACT

// CRITICAL

Description

The `WBTCVault` contract is designed to manage WBTC (Wrapped Bitcoin) tokens, allowing operations such as borrowing tokens to a specified address and repaying debt associated with an NFT ID. However, a critical security vulnerability arises from the absence of access control mechanisms on sensitive functions, namely `borrowAmountTo` and `repayDebt`. This omission allows any user to call these functions without restriction, potentially enabling unauthorized parties to transfer WBTC out of the vault to any address or repay debt without proper authorization.

The `borrowAmountTo` function, intended to enable borrowing operations, directly transfers WBTC to any specified address without validating the caller's permissions. Similarly, the `repayDebt` function allows tokens to be transferred from the caller to the contract without ensuring that the caller is authorized to make such repayments. This lack of access control effectively leaves the vault's funds unprotected, posing a significant risk of unauthorized fund drainage or manipulation of debt records.

BVSS

AO:A/AC:L/AX:L/C:N/I:C/A:C/D:C/Y:C/R:N/S:C (10.0)

Recommendation

To mitigate these vulnerabilities and secure the WBTC funds managed by the `WBTCVault` contract, the following recommendations should be implemented:

1. **Implement Role-Based Access Control (RBAC):** Utilize the `AccessControlUpgradeable` contract from OpenZeppelin to define and enforce role-based permissions for the `borrowAmountTo` and `repayDebt` functions. Specifically, create distinct roles for borrowing and debt repayment, such as `BORROWER_ROLE` and `REPAYMENT_ROLE`, and restrict function access to accounts that have been granted these roles.
2. **Initialization of Access Control in Constructor:** Since the `WBTCVault` contract uses the `AccessControlUpgradeable` pattern, ensure that access control is properly initialized in an `initialize` function (replacing the constructor in upgradeable contracts) and that initial roles are assigned to trusted accounts. This setup prevents unauthorized access immediately upon deployment.
3. **Secure Role Management:** Implement safeguards around role management functions to prevent unauthorized granting or revocation of roles. This typically involves restricting these capabilities to an admin role, which should be held by a governance mechanism or a highly trusted account.
4. **Audit and Testing:** Conduct a thorough security audit of the contract, focusing on access control logic and interactions with the WBTC token. Comprehensive testing, including unit tests and integration tests, should be performed to ensure that access controls are correctly enforced under various scenarios.

5. **Monitor and Review:** Regularly monitor contract interactions for unauthorized attempts to access restricted functions and review the roles and permissions setup to adapt to new security requirements or operational needs.
By implementing robust access control mechanisms, the `WBTCVault` contract can significantly enhance the security of the funds it manages, protecting against unauthorized access and potential exploitation.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

3ffd522ec363a71e43a28bfc43f715bf1f973781

7.2 UNRESTRICTED ACCESS TO SWAP FUNCTIONS

// HIGH

Description

The `UniV3SwapAdapter` contract's functions `swapToWbtc` and `swapFromWbtc` are critical operations that enable swapping tokens to and from WBTC on the Uniswap V3 protocol. However, these functions are currently accessible to any caller without any access control checks. This design flaw presents a significant security vulnerability, as it allows any external party to execute swaps using tokens held by the `UniV3SwapAdapter`, potentially draining the contract of valuable assets. Given that these functions assume tokens have already been transferred to the adapter, the lack of access restrictions means unauthorized users could exploit this assumption, redirecting the outcome of swaps to addresses under their control.

BVSS

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

Recommendation

To mitigate this vulnerability and safeguard the assets managed by the `UniV3SwapAdapter`, the following security measures are recommended:

1. **Implement Role-Based Access Control:** Utilize the `AccessControlUpgradeable` functionality inherited by the `UniV3SwapAdapter` to introduce role-based access control for the `swapToWbtc` and `swapFromWbtc` functions. Specifically, define an `INTERNAL_CONTRACT_ROLE` or a similar role dedicated to contracts within the protocol that are authorized to initiate swaps.
2. **Restrict Swap Function Calls:** Modify the `swapToWbtc` and `swapFromWbtc` functions to include a role check, ensuring that only callers with the appropriate role can execute these operations. This restriction should apply to contracts that are part of the protocol's operational flow, such as the `ClosePosition` contract or other internal mechanisms that require swapping functionality.
3. **Role Assignment and Management:** Ensure that roles are carefully managed, with the assignment of the `INTERNAL_CONTRACT_ROLE` being restricted to trusted contracts within the protocol. Use the `AccessControlUpgradeable` contract's functions to manage role assignments securely.
4. **Audit and Testing:** After implementing access controls, conduct thorough testing and potentially an external audit to verify that the access restrictions are correctly enforced. Testing should cover attempts to call the swap functions both with and without the required roles to ensure that unauthorized access is effectively blocked.

By enforcing strict access control on the `swapToWbtc` and `swapFromWbtc` functions, the `UniV3SwapAdapter` can significantly reduce the risk of unauthorized token swaps and asset drainage, ensuring that swap operations are executed as intended within the protocol's secure operational framework.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

3f5d9d659cb366d8a07cdd73b3b0f44dbf5a7bf6

7.3 MISSING ORACLE VALIDATION IN PRICE RETRIEVAL

// HIGH

Description

The `OracleManager` and `ChainlinkOracle` contracts lacks critical validation checks in the `getLatestTokenPriceInETH` and `getLatestTokenPriceInUSD` functions for ensuring the integrity and availability of oracle data. Specifically, it does not verify if an oracle address is set (non-zero) before attempting to access its methods, potentially leading to a revert if the address is zero. Additionally, there's no validation to ensure that the returned price is non-zero or to check the freshness of the data by examining the last update timestamp. This oversight can result in the system accepting stale or invalid price data, affecting the integrity of operations dependent on accurate token pricing.

BVSS

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

Recommendation

It's crucial to implement comprehensive validation within the `getLatestTokenPriceInETH` and `getLatestTokenPriceInUSD` functions to safeguard the contract against incorrect price data usage and to ensure data reliability. The recommendations are as follows:
1. Before attempting to access an oracle's methods, verify that the oracle address for the given token is set (i.e., not zero). This can prevent the contract from attempting to interact with an unset (zero address) oracle.
2. Incorporate checks to ensure that the price returned by the oracle is non-zero. Accepting a zero price can have detrimental effects on the system's operations, potentially leading to incorrect calculations or decision-making.
3. Validate the freshness of the oracle data by checking the last update timestamp against a predefined threshold. This step ensures that the contract only uses current data and mitigates the risk of relying on stale information.
Implementing these validations will enhance the contract's resilience against potential manipulation or errors in the oracle data, thus preserving the integrity and reliability of the pricing mechanism within the Leverage Engine system.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

e2ef9e2fed377aed5af432623afee6a6baa41d53

7.4 UNPROTECTED INITIALIZE FUNCTION IN UPGRADEABLE CONTRACTS

// MEDIUM

Description

In the context of upgradeable contracts such as `SwapManager`, `OracleManager`, `ProtocolParameters`, `UniV3SwapAdapter`, `PositionLedger`, `PositionOpener`, and `LeveragedStrategy`, a critical security oversight has been identified. The initializer functions in these contracts are not adequately protected, failing to invoke `_disableInitializers` provided by OpenZeppelin's upgradeable contracts library. This omission leaves the contracts vulnerable to unauthorized initialization or re-initialization, which could lead to several adverse scenarios. Malicious actors might exploit this vulnerability to seize control over these contracts by initializing them with arbitrary addresses or parameters, potentially facilitating phishing attacks or other forms of exploitation.
Moreover, the gravity of this issue escalates if any of these contracts contain or could be made to execute a `delegatecall` to untrusted contracts. This could enable attackers to execute arbitrary code within the context of these contracts, including self-destruct operations. Should such a critical contract be destroyed, it would render all dependent proxy contracts inoperative, effectively crippling the entire protocol.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (6.3)

Recommendation

To mitigate these vulnerabilities and enhance the security posture of the upgradeable contracts, the following remediation steps are recommended:
1. Ensure that all initializer functions in the upgradeable contracts explicitly call `_disableInitializers` after their initial setup logic. This action will prevent any further calls to the initializer functions, effectively guarding against unauthorized re-initialization attempts.
2. Conduct a thorough audit of all contracts, particularly focusing on those intended to be used as upgradeable proxies, to verify that `_disableInitializers` is correctly implemented and that no paths allow for unintended initialization.
By addressing this vulnerability promptly and following these recommendations, the security, and integrity of the upgradeable contracts and, by extension, the entire protocol can be significantly bolstered against potential attacks and unauthorized access.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

dc840606072610382193263ec02e25ec09d10f6a

7.5 EXCESSIVE USE OF APPROVE WITH MAXIMUM ALLOWANCE

// MEDIUM

Description

The `WBTCVault` contract's `setlvBTCPoolAddress` and `swapToWBTC` functions set maximum token allowances for both WBTC and LVBTC tokens on a new pool address and `curvePool`. This practice poses a security risk as it grants potentially excessive permissions that could be exploited if the pool address is compromised. Additionally, the function does not revoke the token allowance from a previously authorized pool address, increasing the risk of unauthorized access and potential token drainage if the old address becomes malicious or is otherwise compromised.

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:N/D:N/Y:N/R:N/S:C (6.3)

Recommendation

To mitigate the risks associated with excessive token allowances and to secure the contract against potential compromises:

- Implement a mechanism to reset allowances to zero before setting new ones. This is a defensive programming practice that can prevent old contracts from retaining access after a new address is authorized.
- Consider setting allowances to the exact amount needed per transaction rather than using `type(uint256).max`. This approach minimizes potential damage in the event of a security breach.
- Regularly audit and monitor approved addresses, and implement robust security checks and balances for address updates.

Remediation Plan

RISK ACCEPTED: The **Archimedes Finance team** accepted the risk of this finding.

7.6 INCONSISTENT MINIMUM POSITION DURATION LOGIC

// MEDIUM

Description

The `ProtocolParameters` contract initializes the `minPositionDurationInBlocks` variable to `0` by default, which creates a logical inconsistency with the `setMinPositionDurationInBlocks` function that enforces a minimum threshold of `1` block for setting this parameter. This discrepancy results in a situation where, if `setMinPositionDurationInBlocks` is not invoked to set a duration explicitly, the system's logic would consider the minimum position duration requirement as passed immediately upon position creation, as illustrated by the `isMinPositionDurationPassed` function in the `PositionCloser` contract. This behavior undermines the intended protective mechanism of enforcing a minimum holding period for positions, potentially exposing the system to manipulation or unintended interactions immediately after position creation.

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 resolve this inconsistency and ensure the protocol enforces a coherent and intentional minimum position duration, the following steps are recommended:

1. **Initialize with a Reasonable Default Value:** Set `minPositionDurationInBlocks` to a sensible default value that aligns with the protocol's operational security and user experience requirements. If the intention is to enforce a minimum duration for positions, initializing this variable to a value greater than `0` (e.g., the protocol's typical minimum duration or the `12` blocks mentioned in the TODO comment) can prevent logical inconsistencies before the administrative configuration.
2. **Documentation and Communication:** Clearly document the rationale and mechanism for the minimum position duration within the contract and any external protocol documentation. Inform stakeholders of the importance of setting this parameter to align with the protocol's risk management and operational strategies.
3. **Comprehensive Testing:** Implement thorough testing scenarios to validate the behavior of the minimum position duration logic under various conditions, including default initialization, explicit setting, and edge cases. This will ensure the protocol behaves as expected across all possible states.

By addressing this inconsistency, the protocol can enforce a minimum position duration effectively, enhancing its security posture and ensuring a consistent user experience.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

O1dOd19537332f4b5c17b7b60fcO738O6c61674d

7.7 UNVERIFIED SWAP ADAPTER ADDRESS

// LOW

Description

In the `SwapManager` contract, the `getSwapAdapterForRoute` function returns an ISwapAdapter instance based on the specified `SwapRoute` without verifying if the adapter's address has been set or if it points to the zero address (àddress(0)). This absence of validation can lead to scenarios where consuming contracts, such as `PositionOpener`, interact with the returned adapter without ensuring its validity. As a result, operations intended to execute through these adapters may fail silently, revert, or, in the worst case, lead to the loss of funds if the contracts are not designed to handle such cases properly. This risk is exacerbated in scenarios where the adapter is critical for executing swaps or other financial operations, as the assumption of a valid adapter address is integral to the correctness and safety of these operations.
In current scope, the code will revert when not set, it is recommended to handle this case scenario gratefully.

BVSS

AO:A/AC:L/AX:L/C:N/I:H/A:N/D:N/Y:N/R:P/S:U (3.8)

Recommendation

To mitigate the risks associated with returning an uninitialized or invalid swap adapter address, the following recommendations should be implemented:
1. **Implement Address Validation:** Modify the `getSwapAdapterForRoute` function to include a check that validates the adapter address is not `address(0)` before returning it. If the address is `address(0)`, the function should revert with a clear error message indicating that the adapter has not been set or is invalid.
2. **Centralize Validity Checks:** By ensuring the `getSwapAdapterForRoute` function itself validates the adapter address, all consuming contracts are inherently protected from interacting with invalid adapters. This centralization of validity checks reduces the burden on each consumer to individually verify adapter addresses, thereby enhancing the overall security posture of the system.
3. **Error Handling:** Utilize the `ErrorsLeverageEngine` library to define a custom error for this scenario, such as `AdapterNotSet` or `InvalidAdapterAddress`. This approach provides more informative feedback than a generic revert message and helps with diagnosing issues during development and auditing.
4. **Comprehensive Testing:** After implementing the validation logic, conduct thorough testing to ensure the function behaves as expected. Test cases should include scenarios where the adapter is not set, is set to `address(0)`, and is set to a valid address. This will validate that the system correctly handles each scenario and protects against invalid adapter interactions.
Implementing these recommendations will significantly enhance the safety and robustness of operations involving swap adapters within the system, ensuring that only valid adapters are used for critical financial operations.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

e7f363cb1f5c5985e6fea470a68e46903a66b156

7.8 MISMATCHED STRATEGY POSITION LIFETIME AND MINIMUM POSITION DURATION

// LOW

Description

The `LeveragedStrategy` contract specifies a `positionLifetime` for each strategy, defining the duration a position remains active before it can be considered for expiration or closure. Concurrently, the `ProtocolParameters` contract enforces a `minPositionDurationInBlocks`, setting a system-wide minimum threshold for the duration positions must remain open. A critical configuration inconsistency arises when a strategy's `positionLifetime` exceeds the `minPositionDurationInBlocks` specified in `ProtocolParameters`. This discrepancy could prematurely classify positions as expired by the monitoring system before users have the opportunity to close them, potentially leading to financial losses or an inability to manage positions effectively.

BVSS

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

Recommendation

To mitigate the risks associated with this configuration inconsistency and to ensure that the protocol operates as intended, allowing users adequate time to manage their positions, the following remediation steps are recommended:

1. **Validation Check on Strategy Configuration:** Implement a validation check within the `setStrategyConfig` function in the `LeveragedStrategy` contract to ensure that any configured `positionLifetime` does not exceed the `minPositionDurationInBlocks` obtained from the `ProtocolParameters`. If a proposed `positionLifetime` violates this rule, the function should revert with an informative error message.
2. **Comprehensive Testing:** Conduct thorough testing to ensure that the validation logic correctly prevents the configuration of a `positionLifetime` that exceeds the `minPositionDurationInBlocks`. Include tests for scenarios where `minPositionDurationInBlocks` is changed after strategies have been configured to verify that the system remains consistent.

By implementing these recommendations, the protocol can avoid the risks associated with mismatched position lifetimes and minimum duration thresholds, ensuring that users have sufficient time to close or manage their positions before they are considered expired.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

06084e20d84c9d7f51078c8e051f9539608ce0b3

7.9 UNRESTRICTED POSITION ENTRY OVERWRITE

// LOW

Description

The `PositionLedger` contract's `createNewPositionEntry` function allows for the creation or updating of ledger entries keyed by NFT ID without verifying the initial state of the position. This design leads to a critical vulnerability where an existing position's details could be overwritten without any checks, provided the caller has the necessary `INTERNAL_CONTRACT_ROLE` role. Given the consecutive nature of `nftID` assignment and the internal control flow, under normal operation, each NFT ID should correspond to a unique position that transitions from `UNINSTANTIATED` to `LIVE` without the possibility of overwriting. However, the lack of validation against the position's current state before creating or updating an entry means that if the function is called with an `nftID` that already has an associated live, expired, liquidated, or closed position, the existing entry could be unintentionally or maliciously overwritten. This oversight jeopardizes the integrity and availability of the ledger data, potentially leading to loss of funds, incorrect position tracking, and exploitation of the contract's state for fraudulent purposes.

BVSS

AO:S/AC:L/AX:L/C:N/I:C/A:H/D:N/Y:N/R:N/S:C (3.0)

Recommendation

To mitigate this vulnerability and prevent unauthorized overwriting of existing position entries, the following remediation steps are recommended:
1. Modify the `createNewPositionEntry` function to include a precondition check that ensures an entry for the provided `nftID` can only be created if it is in the `UNINSTANTIATED` state. This can be achieved by adding a check to verify that either the entry does not exist or, if it does, its `state` is `UNINSTANTIATED`. Since the default value for an uninitialized `PositionState` enum is `UNINSTANTIATED` (implicitly `0`), the check should confirm that either the entry is new or explicitly set to this state.
2. Implement an additional function or modifier that encapsulates this logic, ensuring that the contract's design robustly enforces the uniqueness and immutability of ledger entries once they move beyond the initial state.
3. Consider implementing event logging for the creation and modification of ledger entries to provide transparency and traceability of ledger operations. This would facilitate monitoring and auditing of ledger state transitions.
By enforcing these constraints, the `PositionLedger` contract will robustly prevent the accidental or malicious overwriting of position data, thereby preserving the integrity of the ledger and protecting against potential exploitation or loss.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

69f8cf4fe14e02ffe7d750babe735c91da571d30

7.10 UNRESTRICTED STATE MODIFICATION OF NON-EXISTENT POSITIONS

// LOW

Description

The `setPositionState` and `setClaimableAmount` functions in the `PositionLedger` contract permit the state of an arbitrary NFT ID to be updated without validation checks to ensure the NFT ID corresponds to an existing position. This flaw could lead to the manipulation of the ledger by setting the state of non-existent positions, including those potentially created in the future. The function fails to verify that the NFT ID's corresponding entry is not in the `UNINSTANTIATED` state before allowing the state modification, which could lead to inconsistencies within the ledger. Specifically, this could allow unauthorized or unintended modifications to the ledger's state, potentially leading to incorrect position tracking, the mishandling of positions, or exploitation where future positions could be preemptively set to an invalid state.

BVSS

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

Recommendation

To address this vulnerability and ensure the integrity of position state modifications within the `PositionLedger`, the following steps are recommended:

1. **Implement Validation Checks:** Modify the `setPositionState` function to include a validation check that ensures the NFT ID corresponds to an existing position. This involves verifying that the position's current state is not `UNINSTANTIATED`. Since the default state for uninitialized positions is `UNINSTANTIATED`, this check effectively ensures that the function operates on valid, previously created positions.
2. **Error Handling for Non-existent Positions:** Introduce error handling that reverts the transaction if an attempt is made to modify the state of a non-existent position. This could leverage custom error messages from `ErrorsLeverageEngine` to provide clear feedback on the nature of the failure.

By implementing these recommendations, the security and reliability of the `PositionLedger` contract can be significantly enhanced, protecting against potential manipulation or errors that could affect the management and tracking of positions within the system.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

e76b3787c7963b88b6a2b9ceeadb4d15da696068

7.11 GETTER FUNCTIONS WITHOUT EXISTENCE CHECK

// LOW

Description

The `PositionLedger` contract includes several getter functions such as `getCollateralAmount`, `getPositionState`, `getStrategyAddress`, `getDebtAmount`, `getStrategyShares`, `getClaimableAmount`, and `getOpenBlock`. These functions are designed to retrieve specific data points about a position from the ledger entries based on an NFT ID. However, they do not perform checks to verify the existence of the entry for the provided NFT ID before accessing its attributes. In Solidity, attempting to access a non-existent entry in a mapping will not cause a revert; instead, it will return default values (e.g., `0` for `uint256`, `address(0)` for `address`, and the first enum value for enums). This behavior could lead to misleading results, such as indicating that a position has zero collateral or is in the default state (`UNINSTANTIATED`), even if the position does not actually exist.

This lack of existence verification can cause confusion and potentially lead to incorrect assumptions or decisions based on the default data returned for non-existent positions. It could mislead external callers or other contracts interacting with the `PositionLedger` about the actual state of a position or the existence of a position associated with a given NFT ID.

BVSS

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

Recommendation

To ensure the integrity and reliability of the data returned by getter functions in the `PositionLedger` contract, it is recommended to implement existence checks within these functions. Specifically, the contract should:

1. **Verify Position Existence:** Before accessing and returning data for a position, each getter function should check if the position exists and is not in the `UNINSTANTIATED` state. This could be achieved by verifying that the position's state is different from the default `UNINSTANTIATED` state, or by introducing a separate mapping to track the existence of NFT IDs explicitly.
2. **Revert on Non-existent Positions:** If a getter function determines that the requested position does not exist, it should revert the transaction with an informative error message. This behavior will prevent external callers or other contracts from receiving misleading default values for non-existent positions.

By adopting these recommendations, the `PositionLedger` contract will enhance data accuracy and reliability, providing clear feedback to users and developers when attempting to access non-existent positions, thereby preventing potential confusion and misinterpretation of the contract's state.

Remediation Plan

NOT APPLICABLE: This is not an issue, since all the functions will revert to zero values for corresponding types.

7.12 UNRESTRICTED EXIT FEE SETTING

// LOW

Description

The `ProtocolParameters` contract's `setExitFee` function allows for the exit fee to be set without imposing an upper limit on its value. Since the exit fee is defined as a percentage of profits taken upon the exit of a position, expressed in basis points (where `10000` represents 100%), not enforcing a maximum limit of `10000` allows for the possibility of setting the fee to exceed 100%. This could result in scenarios where the fee calculation exceeds the total value of the profits, leading to unintended economic consequences for users, potentially rendering the protocol economically infeasible or exploitative.

BVSS

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

Recommendation

To mitigate potential risks associated with setting an excessively high exit fee and to ensure the protocol remains economically viable and fair to its users, it is recommended to implement a validation check within the `setExitFee` function. Specifically, the contract should:

1. **Implement a Maximum Fee Validation:** Before updating the exit fee, add a check to ensure the new fee does not exceed `10000` basis points, which corresponds to 100%. If the proposed fee exceeds this threshold, the function should revert with an informative error message.
2. **Error Handling:** Utilize the `ErrorsLeverageEngine` library to define and revert with a custom error message if the fee is too high, enhancing the clarity and robustness of error reporting.
3. **Testing and Documentation:** After implementing the maximum fee validation, thoroughly test the `setExitFee` function to ensure it correctly enforces the limit. Additionally, update the contract's documentation to clearly specify the fee's maximum allowable value, informing users and developers of this constraint.

By introducing these safeguards, the `ProtocolParameters` contract will ensure that the exit fee remains within reasonable bounds, protecting users from potentially excessive charges and maintaining the economic integrity of the protocol.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

38a2e4d964055e05378a43c89a993e00e1a7a819

7.13 LIQUIDATION FEE TRANSFER TO UNSET ADDRESS

// LOW

Description

In the `PositionLiquidator` contract, the `collectLiquidationFee` function is designed to transfer a calculated liquidation fee to a fee collector address. This address is retrieved from the `ProtocolParameters` contract via the `getFeeCollector` function. However, there's a potential vulnerability in the case where the `feeCollector` address has not been set (i.e., it is `address(0)`). Solidity allows transfers to `address(0)`, but such transfers are generally considered erroneous and can lead to the permanent loss of tokens because `address(0)` is a burn address from which transferred tokens cannot be recovered.
The lack of a check on the `feeCollector` address before executing the transfer in `collectLiquidationFee` means that if `getFeeCollector` returns `address(0)`, the liquidation fee will be sent to this address, effectively burning the tokens. This behavior could lead to unintended loss of funds, undermining the economic incentives and operational integrity of the liquidation process.

BVSS

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

Recommendation

To mitigate this risk and ensure that liquidation fees are only transferred to a valid, explicitly set fee collector address, the following remediation steps are recommended:
1. **Validate `feeCollector` Address:** Before performing the transfer in `collectLiquidationFee`, add a check to ensure that the `feeCollector` address is not `address(0)`. If it is, the function should revert with an informative error message. This can prevent the accidental burning of tokens by ensuring that the fee is only transferred to a valid address.
2. **Set a Default Fee Collector:** Consider initializing the `ProtocolParameters` contract with a default `feeCollector` address upon deployment or through an initial setup function. This can provide a failsafe mechanism to ensure that there is always a valid address set as the fee collector.
3. **Error Handling:** Utilize the `ErrorsLeverageEngine` library to define a custom error for an unset or invalid `feeCollector` address. This error should be thrown if the address validation fails, providing clear feedback on the nature of the failure.
4. **Testing and Documentation:** After implementing the address validation logic, conduct thorough testing to ensure that the `collectLiquidationFee` function behaves as expected across all scenarios, including with unset, set, and updated `feeCollector` addresses. Update the contract documentation to reflect the importance of setting a valid `feeCollector` address before initiating liquidations.
By incorporating these recommendations, the `PositionLiquidator` contract can safeguard against the unintended loss of liquidation fees, ensuring that they are correctly distributed according to the protocol's economical design and operational requirements.

Remediation Plan

RISK ACCEPTED: The **Archimedes Finance team** accepted the risk of this finding.

7.14 IMPROPER BALANCE VERIFICATION IN SWAPTOLVBTC

// LOW

Description

The `swapToLVBTC` function in the `WBTCVault` contract swaps WBTC for LVBTC through a pool exchange and then burns the entire LVBTC balance of the contract. This function only verifies the balance after the exchange is completed, relying on the entire new balance, which may lead to unintended behavior or potential vulnerabilities. If the balance is manipulated by an external event or there are discrepancies in the transaction execution (e.g., due to reentrancy or other state changes), the outcome could result in incorrect or excessive burning of LVBTC.

BVSS

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

Recommendation

To enhance the robustness and security of the `swapToLVBTC` function:

- Implement balance checks before and after the exchange to ensure that the expected amount of LVBTC has been correctly credited to the contract's account. This can be done by calculating the expected new balance based on the previous balance and the transaction amount, then comparing it to the actual new balance.
- Utilize a reentrancy guard to prevent potential attacks that could exploit the state changes during the execution of this function.
- Consider locking mechanisms or checks to validate that no other operations affecting the balance can be interleaved with this function’s execution, thus maintaining transaction atomicity.

Remediation Plan

RISK ACCEPTED: The **Archimedes Finance team** accepted the risk of this finding.

7.15 REDUNDANT STATE VARIABLE ASSIGNMENT

// INFORMATIONAL

Description

In the `ClosePositionInternal` contract, the `setDependenciesInternal` function redundantly sets the `positionLedger` state variable twice with the same value from the `dependencies.positionLedger` input. This redundancy does not impact the contract's functional behavior but results in unnecessary gas consumption during execution. Each state variable assignment in a smart contract costs gas, and redundant assignments without any functional necessity waste part of the transaction fees. In the context of Ethereum and other EVM-compatible blockchains, optimizing gas usage is crucial for minimizing operational costs and enhancing contract efficiency.

BVSS

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

Recommendation

To address the issue of gas inefficiency due to the redundant assignment of `positionLedger`, it is recommended to remove the duplicate assignment operation from the `setDependenciesInternal` function.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

9bf2cb36177091b006a7fde5db38496a1dfc03f5

7.16 REDUNDANT DATA ACCESS FUNCTION

// INFORMATIONAL

Description

The `PositionLedger` contract contains a public `entries` mapping that automatically generates a getter function, allowing for direct access to `LedgerEntry` data by providing an NFT ID. However, the contract also explicitly defines a `getPosition` function that serves the same purpose, to return the `LedgerEntry` for a given NFT ID. This explicit definition is redundant because Solidity automatically creates a getter for public state variables, including mappings. The redundancy of the `getPosition` function does not directly impact the contract's security but represents an unnecessary use of contract bytecode, potentially leading to increased deployment and execution costs due to the larger contract size. Additionally, maintaining unnecessary code increases the complexity and potential for errors in contract management and understanding.

BVSS

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

Recommendation

To optimize the `PositionLedger` contract and adhere to best practices in Solidity development, it is recommended to remove the `getPosition` function. This action will reduce the contract size, lower deployment and execution costs, and simplify the contract's interface without losing any functionality. Users and other contracts can directly access the `entries` mapping to retrieve position data, leveraging the automatically generated getter function provided by Solidity for public mappings.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

56221da257f114735ddeb0cafec33652dd2fe541

7.17 INCOMPLETE IMPLEMENTATION AND REDUNDANT CODE

// INFORMATIONAL

Description

Throughout the codebase, including the `PositionLedger` contract, several "TODO" comments indicate functions that are either not fully implemented or suggested for removal. These markers suggest areas of the code that may require further development, optimization, or outright deletion to meet the intended design and functionality goals. In a deployed smart contract, incomplete implementations and unused functions can pose risks, such as unnecessary gas costs, cluttered contract interfaces, and potential vectors for misunderstandings or misuse. Specifically, comments suggesting the removal of functions or indicating incomplete implementation highlight areas where the contract may not align with its intended operational design or could be optimized for efficiency and clarity.

For instance, the presence of functions marked for removal, such as certain getters or utility functions that have become redundant due to public state variables, suggests that the contract's current implementation includes unnecessary code. This can increase the contract size, leading to higher deployment and execution costs, and potentially obscure the contract's logical flow. Moreover, incomplete implementations indicated by TODOs may signal that the contract is not ready for production use, as critical functionality might be missing or behave unpredictably.

BVSS

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

Recommendation

To address these issues and ensure the contract aligns with best practices for smart contract development, the following steps are recommended:

1. **Review and Resolve TODOs:** Conduct a comprehensive review of the codebase to identify all TODO comments. Evaluate each item to determine the necessary action, whether completing the implementation, removing unnecessary functions, or refining existing logic.
2. **Remove Redundant Code:** For functions marked for removal or identified as redundant (e.g., due to overlapping functionality with automatically generated getters for public variables), carefully remove these functions after confirming they are not used elsewhere in the codebase. Reducing the contract's size can lower deployment and execution costs and simplify its interface.
3. **Complete Partial Implementations:** Where TODO comments indicate incomplete functionality, prioritize completing these sections of the code. Ensure that all intended features are fully implemented and tested to prevent deploying contracts with missing or placeholder functionality.
4. **Update Documentation:** After making changes based on TODOs, update the contract's documentation and inline comments to reflect the current state of the code, including any modifications to functions or logic. Clear documentation is crucial for future maintenance and understanding the contract's intended behavior.
5. **Comprehensive Testing:** After addressing all TODOs and making the necessary code modifications, conduct thorough testing to ensure that the contract functions as expected. Include unit tests, integration

tests, and, if applicable, formal verification to validate the contract's integrity and security.
By systematically addressing TODO comments and refining the contract's implementation, developers can enhance the contract's clarity, efficiency, and reliability, ensuring it is optimized for production deployment and use.

Remediation Plan

SOLVED: The **Archimedes Finance team** solved this issue.

Remediation Hash

a8f21c05a764d06540ed183623458e79e38dc4da

7.18 TYPO IN LEDGERENTRY STRUCTURE FIELD NAME

// INFORMATIONAL

Description

In the context of smart contract development, especially for a protocol managing financial transactions or positions, the accuracy and clarity of code are paramount. A typo in a variable or struct field name, while often considered a minor issue, can lead to confusion, errors in code handling, and potentially impact the interaction with other contracts or off-chain systems.
The `LedgerEntry` struct contains a field named `poistionOpenBlock`, which is intended to record the block number at which a position was opened. However, due to a typographical error, it's mislabeled as `poistionOpenBlock` instead of the correct `positionOpenBlock`. This mistake does not directly affect the contract's execution or security but may cause confusion or errors in maintenance, upgrade processes, or when the field is accessed or used by developers or external systems.

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 correct this issue and improve code readability and maintainability, rename the `poistionOpenBlock` field in the `LedgerEntry` struct to `positionOpenBlock`. This change should be made carefully to ensure that all references to this field in the contract are updated accordingly to reflect the new, correct name.

Remediation Plan

SOLVED: The **Archimedes Finance** team solved this issue.

Remediation Hash

ddcea5a9f51ee87e773a4110f8fb6be84d311ea4

7.19 FLOATING PRAGMA IN SOLIDITY CONTRACTS

// INFORMATIONAL

Description

The smart contracts within the protocol utilize a floating pragma statement, `pragma solidity >=0.8.21;`, indicating compatibility with any Solidity compiler version starting from 0.8.21 onwards. While this offers flexibility in compiler choice, it introduces risks associated with compiler version inconsistency, unintended behavior, or vulnerabilities in newer compiler versions that were not present or known at the time of contract development.
Floating pragmas can lead to scenarios where contracts are compiled with different, potentially untested, compiler versions across development, testing, and deployment environments. This variance increases the risk of introducing bugs or vulnerabilities that could compromise contract integrity or security.

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 mitigate the risks associated with floating pragma statements and ensure consistent, predictable behavior across all environments, the following remediation steps are recommended:
1. **Fixed Pragma Statement:** Adopt a fixed pragma statement that specifies a single, specific compiler version (e.g., `pragma solidity 0.8.21;`). This approach guarantees that contracts are compiled with a compiler version known to be compatible and tested, reducing the risk of unexpected behavior or vulnerabilities.
2. **Version Locking and Testing:** When updating to a new Solidity compiler version, thoroughly test contracts with the new version to ensure compatibility and security before updating the pragma statement to lock in the new version.
Adopting these recommendations will help ensure that smart contracts are compiled with a known, tested compiler version, reducing the risk of bugs or vulnerabilities introduced by compiler updates and contributing to the overall security and stability of the protocol.

Remediation Plan

ACKNOWLEDGED: The **Archimedes Finance** team acknowledged this finding.

8. REVIEW NOTES

Constants

- Is declaring 3 variables, `WBTC_ADDRESS`, `UNISWAPV3_ROUTER_ADDRESS`, `USDC_ADDRESS`. All of them mapped to the corresponding mainnet address.

ProtocolRoles

- Is is defining several roles used across the protocol as bytes32.

PositionToken

- Does extend `ERC721Enumerable` and `Ownable`.
- The `_tokenIdCounter` is incremented on `mint`. The token id does start at 1.
- `burn` can only be called by the owner of the token.

OracleManager

- Allows storing oracles based on the token address. Only the `ADMIN_ROLE` can perform those action. **However, the getters do not check for invalid or missing oracle, such address 0. Moreover, the returned value is not checked for outdated timestamp.**

PositionLedger

- Allows internal contracts to store and update NFT entries. It is the main ledger used to track positions.
- The setters do not perform existence checks on positions.

ProtocolParameters

- Allows the Admin to set `setExitFee` , `setFeeCollector` and `setMinPositionDurationInBlocks`.
- `minPositionDurationInBlocks` **should not default to 0.**

SwapManager

- The `getSwapAdapterForRoute` is being used on several points to fetch the adapter. However, the code does not check if the adapter is the zero address.

WBTCVault

- Neither `borrowAmountTo` or `repayDebt` do have any access control applied.
- All funds can be withdrawn by anyone.

PositionExpirator

- `expirePosition` can only be called by the `MONITOR_ROLE`. It will mark a position as expired if all the conditions are validated. The leftover will be transferred via `setNftIdVaultBalance` to the expired vault for user to claim.

ExpiredVault

- `Constants.WBTC_ADDRESS)` is used as an internal variable for wbtc. The direct constant could be used during initialisation to reduce gas costs.

- `deposit` can only be called by `INTERNAL_CONTRACT_ROLE`.
- `revertIfPositionIsntClaimableBySender` will verify nft ownership and make sure that state is not `EXPIRED` or `LIQUIDATED`.
- `claim` will decrease the balance after the `revertIfPositionIsntClaimableBySender` check.
- It will burn the nft after.

UniV3SwapAdapter

- Both `swapToWbtc` and `swapFromWbtc` are unrestricted. Parameters are controlled.

LeveragedStrategy

- Access control extend.
- `setDependencies` will grant `positionOpener` the `INTERNAL_CONTRACT_ROLE` role.
- `isPositionLiquidatableEstimation` will make sure that the nft is live.
- `getTokenValueFromWBTCAmount` does correctly transfer an WBTC amount to the amount of tokens required to fullfill it.
- `getWBTCValueFromTokenAmount` will correctly calculate the WBTC value based on the token amount. The following script was used to verify it with multiple test cases:

```
def get_wbtc_value_from_token_amount(token_amount, token_decimals, token_price_in_usd, wbtc_price_in_usd, token_oracle_decimals, wbtc_oracle_decimals, wbtc_decimals=8):

    # Calculate the unadjusted token value in WBTC

    token_value_in_usd = token_amount * token_price_in_usd

    unadjusted_token_value_in_wbtc = token_value_in_usd / wbtc_price_in_usd

    # Adjust for decimal differences

    adjusted_token_value_in_wbtc = adjust_decimals_to_wbtc_decimals(

        token_decimals,

        unadjusted_token_value_in_wbtc,

        token_oracle_decimals,

        wbtc_oracle_decimals,

        wbtc_decimals

    )

    return adjusted_token_value_in_wbtc
```

```
def adjust_decimals_to_wbtc_decimals(token_decimals, amount_unadjusted_decimals,
token_oracle_decimals, wbtc_oracle_decimals, wbtc_decimals):

    from_dec = token_decimals + token_oracle_decimals

    to_dec = wbtc_oracle_decimals + wbtc_decimals

    if from_dec > to_dec:

        adjustment_factor = 10 ** (from_dec - to_dec)

        return amount_unadjusted_decimals / adjustment_factor

    else:

        adjustment_factor = 10 ** (to_dec - from_dec)

        return amount_unadjusted_decimals * adjustment_factor

# Example usage

token_amount = 1e18  # Amount of the token

token_decimals = 18  # Decimals of the token

token_price_in_usd = 500e8  # Price of the token in USD

wbtc_price_in_usd = 30000e8  # Price of WBTC in USD

token_oracle_decimals = 8  # Oracle decimals for the token

wbtc_oracle_decimals = 8  # Oracle decimals for WBTC

# Calculate the value

wbtc_value = get_wbtc_value_from_token_amount(

    token_amount,

    token_decimals,

    token_price_in_usd,
```

```
    wbtc_price_in_usd,

    token_oracle_decimals,

    wbtc_oracle_decimals

)

print("WBTC Value (8 decimals):", wbtc_value)

print("WBTC Value (units):", wbtc_value / 1e8)  # Convert to WBTC (8 decimals
```

LeverageDepositor

- Does directly call `deposit` or `redeem` for the strategy.

PositionOpener

- `getSwapAdapterForRoute` is not verified for returning a valid address. However, any interaction under `swapWbtcToStrategyToken` will revert.
- `sendWbtcToSwapAdapter` will transfer from the vault to the adapter.
- `swapWbtcToStrategyToken` will swap using the strategy adapter to the underlaying token.
- `isSwapReturnedEnoughTokens` will verify that the swapped amount does not turn the position liquidable.

ClosePositionInternal

- `unwindPosition` does redeem from the strategy.
- `swapStrategyTokenToWbtc` does expect `strategyUnderlyingToken` tokens `strategyTokenAmount` amount to be already present on the contract.
- `collectExitFeesAfterDebt` will transfer to the fee collector the corresponding percentage of left over debt.
- `sendBalanceToUser` will revert if the request min wbtc are not meet. The leftover is transferred to the sender, checked to be the nft owner.

ClosePositionBase

- `setNftIdVaultBalance` will set the claimable amount and transfer the balance to the expired vault where the user can call `claim`. Only `Expirator` and `Liquidator` contracts do ever call this function.
- `repayPositionDebt` will return 0 in case the received is less or equal than the debt amount. The returned value is used as left over wbtc. Used on Expirator and Liquidator contracts.
- Allows admin to `setMonitor` and `setExpiredVault`.

PositionLiquidator

- The `liquidatePosition` function can be called as long as the position is "LIVE" and liquidable. It does work similarly to `expirePosition`. There is no way to do both, expiration and liquidation as the state will be changed.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.