// Private Smart Contract Security Assessment 03.08.2024 - 04.05.2024

Native Aqua

Native

HALBORN

Native Aqua - Native

Prepared by: HALBORN

Last Updated 08/08/2024

Date of Engagement by: March 8th, 2024 - April 5th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
14	4	1	1	4	4

TABLE OF CONTENTS

8. Review Notes
9. Automated Testing

1. Introduction

Native engaged our security analysis team to conduct an in-depth examination of their smart contract ecosystem. The primary objective was to rigorously assess the security framework of Native's smart contracts to identify any vulnerabilities, evaluate the current security measures, and provide actionable recommendations to enhance the security and operational performance of their smart contract architecture. The scope of our assessment was strictly limited to the smart contracts provided, ensuring a focused and thorough analysis of the security aspects of the contracts in question.

2. Assessment Summary

Our engagement with Native spanned approximately four weeks, during which we allocated one full-time security engineer with extensive experience in blockchain and smart contract security, advanced penetration testing skills, and a deep understanding of various blockchain protocols to the project. The goals of this assessment were to verify the correct functionality of smart contract functions and to identify any potential security issues within the smart contracts aimed to:

- Verify that smart contract functions operate as intended.
- Identify potential security issues within the smart contracts.

Our in-depth security assessment led to the discovery of several vulnerabilities and areas for improvement, notably:

- **Rounding Issue and Price Inflation in Aqua LP Token Contract**: Uncovered a vulnerability related to rounding issues and price inflation upon the first deposit into an empty pool, akin to the "First Deposit Bug" found in Compound V2. Recommendations include initial LP token distribution adjustments and ensuring non-zero shares minting.
- **Inconsistent ETH Refund Handling**: Identified inconsistencies between `exactInputSingle` and `exactInput` functions in handling ETH refunds to callers, leading to potential inefficiencies and risks.
- **Risks Associated with Non-validated Pool Upgrades**: Highlighted the danger of upgrading pools to incompatible or erroneous implementations without proper validation checks, suggesting the implementation of safeguards similar to those used by Compound.
- **Improper Handling of ETH Refunds in `refundETHRecipient`**: Recommended enhancing the `refundETHRecipient` function to allow specifying the refund amount, instead of refunding the entire contract balance.

These findings underline the need for meticulous attention to contract logic, especially concerning financial calculations, upgrade mechanisms, and ETH handling procedures. Addressing these issues is crucial for mitigating potential vulnerabilities and ensuring the reliability and security of the system.

3. Test Approach And Methodology

A combination of manual and automated testing techniques was employed to ensure a comprehensive assessment. Manual testing was prioritized for its effectiveness in identifying logical and implementation flaws, while automated testing provided extensive code coverage and quick identification of common security pitfalls. The assessment involved:

- An in-depth review of the smart contracts' architecture and purpose.
- A thorough manual code review and walkthrough.
- Functional and connectivity analysis using tools like `solgraph`.
- Custom script-based manual testing and testnet deployment using `Foundry`.

This executive summary encapsulates the critical findings and recommendations from our security assessment of Native's smart contract ecosystem. Addressing the identified issues and implementing the suggested remediation will significantly contribute to the security, reliability, and trustworthiness of Native's smart contract platform.

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
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
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

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
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
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: native-contracts

(b) Assessed Commit ID: 8f05963

(c) Items in scope:

- NativeRouter.sol
- libraries/AquaVaultLogic.sol
- Aqua/AquaVault.sol
- NativeRfqPool.sol
- Aqua/AquaLpToken.sol
- NativePoolFactory.sol
- Aqua/AquaVaultSignatureCheck.sol
- Aqua/ChainlinkPriceOracle.sol
- Native-org/native-contracts/pull/89
- Native-org/native-contracts/pull/91

Out-of-Scope:

REMEDIATION COMMIT ID:

- 6162eca6162eca
- https:/https:/
- https:/https:/
- 31606a431606a4
- 76873a376873a3
- 587ae3c587ae3c
- d7daa6bd7daa6b
- c5ddcc4c5ddcc4
- a143103a143103
- 2f3ac392f3ac39
- aad00b4aad00b4
- d1abdc8d1abdc8

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
4	1	1	4	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PRICE INFLATION IN AQUA LP TOKEN CONTRACT	CRITICAL	SOLVED - 03/31/2024
RISK DUE TO INCLUSION OF UNSUPPORTED MARKETS	CRITICAL	RISK ACCEPTED - 03/31/2024
INSUFFICIENT VALIDATION IN TRADERFQT FUNCTION	CRITICAL	SOLVED - 07/04/2024
INCORRECT EXCHANGE RATE DUE TO UNUPDATED NETBORROW IN REPAY FUNCTION	CRITICAL	SOLVED - 03/19/2024
LACK OF SIGNATURE VERIFICATION FOR TRADER IDENTITY	HIGH	RISK ACCEPTED - 03/31/2024
INADEQUATE REPRESENTATION OF TREASURIES WITH MULTIPLE POOLS	MEDIUM	SOLVED - 03/31/2024
SINGLE-STEP OWNERSHIP TRANSITION RISK IN CONTRACTS	LOW	PARTIALLY SOLVED - 03/31/2024
NON-VALIDATED POOL UPGRADES	LOW	SOLVED - 03/31/2024
MISSING VERIFICATION OF AQUAVAULT ADDRESS IN AQUALPTOKEN DURING MARKET SUPPORT	LOW	SOLVED - 03/31/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
INCONSISTENT ETH REFUND HANDLING	LOW	SOLVED - 03/31/2024
HARDCODED FUNCTION SELECTORS RISK	INFORMATIONAL	SOLVED - 03/31/2024
LACK OF REGISTERED POOL VALIDATION	INFORMATIONAL	SOLVED - 03/31/2024
IMPROPER HANDLING OF ETH REFUNDS	INFORMATIONAL	SOLVED - 03/31/2024
TYPOS AND DOCUMENTATION	INFORMATIONAL	SOLVED - 03/31/2024

7. FINDINGS & TECH DETAILS

7.1 PRICE INFLATION IN AQUA LP TOKEN CONTRACT

// CRITICAL

Description

The Aqua LP Token contract, derived from Compound's CErc20 model, exhibits vulnerabilities related to rounding issues and price inflation when the pool is empty. These issues arise from the mechanism used to calculate the exchange rate between the underlying assets and the LP tokens, particularly during the first deposit in an empty pool. The vulnerabilities are similar to those identified in Compound V2 and its forks, as detailed by Akshay Srivastav in his analysis ("First Deposit Bug in CompoundV2 and Its Forks", `https://akshaysrivastav.hashnode.dev/first-deposit-bug-in-compoundv2-and-its-forks`).
The core of the problem lies in the `exchangeRateStoredInternal` function, which is responsible for calculating the exchange rate of LP tokens. When the total supply of LP tokens (`_totalSupply`) is zero (indicating an empty pool), the function defaults to returning the `initialExchangeRateMantissa`, ignoring the actual amount of assets deposited. This behavior can lead to disproportionate allocation of LP tokens for the first depositor, creating a scenario where the first deposit inflates the price of LP tokens due to inadequate share dilution by sending underlying tokens to the pool/vault.
This vulnerability can be exploited in a scenario where an attacker becomes the first depositor in an empty pool. By depositing a minimal amount, the attacker receives an inflated number of LP tokens relative to their deposit. Subsequent deposits by other users further dilute the pool's value, potentially allowing the attacker to withdraw a disproportionate amount of assets.

Proof of Concept

https://github.com/akshaysrivastav/first-deposit-bug-compv2

BVSS

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

Recommendation

To mitigate these vulnerabilities and ensure a fair and secure token distribution mechanism, the following recommendations are proposed:
1. **Initial LP Token Distribution to Zero Address**: Inspired by Uniswap V2's solution to a similar problem, modify the LP token distribution logic to send the first minimum liquidity LP tokens to the zero address (`https://github.com/Uniswap/v2-core/blob/master/contracts/UniswapV2Pair.sol#L119-L124`). This approach ensures share dilution from the outset, preventing price inflation and rounding issues. Implementing this recommendation involves adding a condition in the token minting function to check if the total supply is zero and, if so, sending the first batch of LP tokens to the zero address.
2. **Non-zero Shares Validation**: Enforce a requirement that the number of LP tokens (`shares`) minted during any deposit transaction is non-zero. This can be achieved by adding a `require` statement to validate `_shares != 0` before completing the minting process.

3. **Atomic Initialization and Deposit via Periphery Contract**: Create a periphery contract containing a wrapper function that atomically calls `initialize()` (if necessary) and `deposit()`. This ensures that the initial deposit into the pool triggers the appropriate share dilution.
4. **Deposit in `initialize()` Function**: Modify the `initialize` function to include a call to `deposit()`, effectively prefunding the pool with an initial amount. This can achieve a similar effect to directly sending LP tokens to the zero address, ensuring immediate share dilution.
By adopting these recommendations, the Aqua LP Token contract can eliminate the rounding issue and price inflation vulnerability, enhancing the fairness and security of the token distribution mechanism. These measures will help maintain the integrity of the liquidity pool and protect against potential exploitation.

Remediation Plan

SOLVED: The **Native team** solved this issue.

Remediation Hash

`6162eca9e414eaaa7c2b8a1d874c24b37593ef96`

7.2 RISK DUE TO INCLUSION OF UNSUPPORTED MARKETS

// CRITICAL

Description

The function `getCredit` within the `AquaVaultLogic` library plays a crucial role in assessing the credit status of traders by evaluating their positions against the price oracle. However, a critical vulnerability arises when an unsupported market (one for which the oracle returns a zero price) is added to the supported markets through the `supportMarket` function in `AquaVault`. This addition is irreversible due to the protocol's design, which does not allow for the removal of supported markets once added.
The consequence of adding an unsupported market is dire: all non-whitelisted traders become incapable of performing actions that would adjust the collateral factor or allow the withdrawal of tokens/collaterals. This situation effectively locks the protocol, making it impossible for these traders to manage their investments, posing a severe risk to the liquidity and functionality of the platform. Although the `supportMarket` function is protected by admin-only access, the potential impact of inadvertently adding an unsupported market could paralyze the entire protocol.

BVSS

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

Recommendation

To mitigate this vulnerability and prevent the inclusion of unsupported markets, several measures should be implemented:
1. **Pre-Validation of Markets**: Introduce a comprehensive pre-validation process for markets before they are added to the protocol.
2. **Reversible Market Support**: Amend the protocol to allow for the removal or deactivation of markets. This flexibility can prevent permanent lock-in effects caused by unsupported markets and provide a way to rectify inadvertent additions. However, this method will require studying and verifying the implications on the collateral calculations and probably require bigger refactoring.
3. **Administrative Safeguards**: Although the `supportMarket` function is admin-protected, enhance the safeguards by requiring multi-signature confirmation or introducing a timelock mechanism for market additions. This would add another layer of oversight and reduce the risk of erroneous market support.
4. **Audit and Monitoring Systems**: Develop and maintain robust audit trails and monitoring systems for market additions and modifications. Regular audits can help identify potential risks associated with market support changes, while real-time monitoring can flag unexpected oracle responses or protocol behaviors.
By implementing these recommendations, the protocol can significantly reduce the risk associated with the addition of unsupported markets, thereby protecting traders' ability to manage their investments and ensuring the overall health and stability of the platform.

Remediation Plan

RISK ACCEPTED: The **Native team** accepted the risk associated with the inclusion of unsupported markets and has implemented changes to address this concern. Specifically, off-chain markets will now

be tracked on-chain but will be excluded from credit calculations (commit: `13a2eb6b28ec38333ec6243559ca9d8f1d26470e`). However, this adjustment introduces a new risk: the potential for liquidating positions that are, in reality, solvent. This issue arises because the price of off-chain markets, when considered for credit assessment, will default to zero. Consequently, if a trader holds a combination of off-chain and on-chain markets that would collectively result in a positive credit, but the on-chain markets alone yield a negative credit due to oracle data, liquidation could still be triggered. While liquidators are considered trusted entities within this system, this change inadvertently creates an opportunity for the unwarranted liquidation of otherwise healthy positions.

7.3 INSUFFICIENT VALIDATION IN TRADERFQT FUNCTION

// CRITICAL

Description

An external security researcher alerted Native of a critical vulnerability in the `tradeRFQT` function of the `NativeRouter` contract. This issue is documented here for trackability regarding the remediation process.

The `quote.externalSwapCalldata` attribute can be exploited to send the `sellerToken` to the router instead of the pool treasury. Consequently, the pool treasury will still send the `buyerToken` to the receiver, but it won't receive the corresponding `sellerToken`, leading to temporary asset lock within the router. If WETH is the `sellerToken`, it can be swapped for any other token in the pool treasury using the same method. The "stuck" WETH can be reclaimed via `unwrapWETH9`, effectively allowing the attacker to gain both the original WETH and additional tokens from the treasuries.

Proof of Concept

https://gist.github.com/0xJuancito/efa8237b1ea3657fbf3d709c8a48be69

BVSS

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

Recommendation

Remediation Plan

SOLVED: The **Native team** solved this issue. To address the vulnerability, the `tradeRFQT` function was modified to ensure that the `payee` address is correctly determined using the `isRfqPool` function instead of relying on the `externalSwapCalldata`. This change ensures that the `sellerToken` is always transferred to the appropriate RFQT pool treasury or the router for external swaps. The updated function implementation includes:

1. **Validation of the RFQT Quote:** The `_validateRFQTQuote` function is called to ensure the integrity and validity of the provided quote.
2. **Determination of Payee:** The `payee` address is set based on the result of the `isRfqPool` function. If the `quote.pool` is a valid RFQT pool, the `payee` is set to the pool's treasury address. Otherwise, it defaults to the router's address for handling external swaps.
3. **Handling ETH and Token Transfers:** The function properly manages the transfer of the `sellerToken` or ETH (wrapped as WETH) to the determined `payee`. This includes processing widget fees and ensuring that the amounts are correctly transferred.

By implementing these changes, the function ensures that the assets are correctly routed to the intended destinations, thereby mitigating the risk of unauthorized asset transfers and temporary asset locks within the router.

Remediation Hash

https://github.com/Native-org/native-contracts/pull/171

7.4 INCORRECT EXCHANGE RATE DUE TO UNUPDATED NETBORROW IN REPAY FUNCTION

// CRITICAL

Description

The `repay` function within the smart contract fails to update the `netBorrow` amount after a repayment has been made. This oversight leads to the `AquaLpToken` not reflecting the actual state of borrowings, resulting in an inaccurate and potentially manipulated exchange rate. Since the exchange rate of LP tokens is a crucial factor in determining the value of tokens during swaps, borrowings, and repayments, its inaccuracy can have significant repercussions.
An attacker could exploit this vulnerability by manipulating the system to acquire tokens at a lower price than their actual value, leveraging the discrepancy between the real and reported state of net borrowings. This could destabilize the system, affecting the integrity and stability of the entire platform, leading to financial losses for users and undermining trust in the platform's reliability.

BVSS

AO:A/AC:L/AX:M/C:N/I:C/A:M/D:N/Y:N/R:N/S:C (9.4)

Recommendation

To mitigate this vulnerability and safeguard the system's integrity, the following steps are recommended:
1. **Update NetBorrow on Repayments**: Modify the `repay` function to correctly update the `netBorrow` amount whenever a repayment is made. This ensures that the `AquaLpToken` accurately reflects the current state of borrowings, leading to a correct calculation of the exchange rate.
2. **Audit and Testing**: Conduct a thorough audit of the smart contract, focusing on functions that affect financial calculations and state updates like borrowings, repayments, and exchange rate calculations. Implement comprehensive testing scenarios to cover edge cases and potential attack vectors.
3. **Real-time Monitoring**: Implement real-time monitoring mechanisms to detect anomalies in transaction patterns, especially those related to repayments and exchange rate calculations. Anomalies could indicate attempts to exploit the system, allowing for quick response and mitigation.
Addressing this vulnerability promptly is crucial in maintaining the platform's security integrity, protecting users' assets, and ensuring the system's long-term viability.

Remediation Plan

SOLVED: The **Native team** solved this issue.

Remediation Hash

https://github.com/Native-org/native-contracts/pull/87

7.5 LACK OF SIGNATURE VERIFICATION FOR TRADER IDENTITY

// HIGH

Description

The smart contract system in question allows for operations that rely on verifying a "signer" to perform actions ostensibly on behalf of a trader. This process, however, is flawed due to its reliance on off-chain requests that do not necessitate active participation or verification from the purported trader. Consequently, this opens up the possibility for malicious entities to forge requests, potentially allowing unauthorized actions to be executed under the guise of any trader within the system. The core of this vulnerability lies in the absence of direct signature verification from the trader, thereby undermining the integrity and authenticity of actions performed within the contract.
The current mechanism, without a stringent verification process involving the trader's direct participation, lacks the robustness to safeguard against identity spoofing or repudiation issues. This vulnerability poses significant risks, including unauthorized transactions, manipulation of account states, and potential loss or compromise of assets, all of which directly impact the confidentiality, integrity, and availability of the smart contract ecosystem.

BVSS

AO:A/AC:L/AX:L/C:M/I:M/A:N/D:N/Y:N/R:N/S:C (7.8)

Recommendation

To mitigate this vulnerability, the protocol should integrate a mechanism requiring traders to actively participate in the signature generation process for all sensitive actions pertaining to their account or assets. This approach involves:
1. **Enhancing On-Chain Verification:** Implement an on-chain mechanism that requires each transaction or sensitive action to be accompanied by a valid signature from the trader. This signature should be generated based on the transaction details and a private key uniquely owned by the trader.
2. **Off-Chain Signature Requirement:** Before any transaction is processed or forwarded to the smart contract, the backend API should validate a signature provided by the trader. This step ensures that the trader has indeed authorized the specific action, adding another layer of security and authenticity verification.
3. **Comprehensive Authentication Flow:** Establish a robust authentication flow where traders sign a message or transaction payload with their private key. The smart contract system should then verify this signature against the trader's public key or address stored or referenced within the blockchain. This process ensures that only the rightful owner of an account can initiate transactions or changes to their state within the contract.
4. **Documentation and User Education:** Update documentation and user guides to clearly explain the importance of signature verification and the steps required for traders to securely generate and manage their keys.
Adopting these measures will significantly enhance the security posture of the smart contract system by ensuring actions are verifiably consented to by the account holders, thereby preventing unauthorized access and manipulation.

Remediation Plan

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

7.6 INADEQUATE REPRESENTATION OF TREASURIES WITH MULTIPLE POOLS

// MEDIUM

Description

In the `NativePoolFactory` contract, the `treasuryToPool` mapping is used to associate treasury addresses with pool addresses. This design assumes a one-to-one relationship between treasuries and pools. However, the contract also introduces the concept of `isMultiPoolTreasury`, allowing a single treasury to be associated with multiple pools. The current mapping structure does not support this multiplicity effectively, as it can only map a treasury to a single pool address. Consequently, when a treasury is used for multiple pools, the `treasuryToPool` will only retain a reference to the last pool created with that treasury address. This limitation hinders the ability of the contract to accurately represent the relationship between treasuries and pools, particularly for querying all pools associated with a given treasury.

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 and accurately track the association between treasuries and their multiple pools, it is recommended to modify the `treasuryToPool` mapping to map treasury addresses to an array of pool addresses. This change will enable the contract to maintain a comprehensive record of all pools associated with a given treasury, reflecting the `isMultiPoolTreasury` functionality accurately.

- **Modify Mapping Structure**:

Change the `treasuryToPool` mapping from `mapping(address => address)` to `mapping(address => address[])`. This alteration requires adjustments to all contract functions that interact with `treasuryToPool`, ensuring they correctly handle an array of pool addresses.

- **Adjust Function Logic**:

- In `createNewPool`, append the new pool address to the array of pools associated with the treasury.
- Modify `getPool` to either return the array of pool addresses or consider introducing a new function, such as `getPoolsByTreasury`, to provide this functionality explicitly.
- Ensure that functions which interact with `treasuryToPool` are updated to iterate over or otherwise properly manage an array of addresses.

Remediation Plan

SOLVED: The **Native team** solved this issue.

Remediation Hash

3160 6a4ef76d624071e230bfe46286a21a631dbe

7.7 SINGLE-STEP OWNERSHIP TRANSITION RISK IN CONTRACTS

// LOW

Description

Contracts such as `NativePool`, `NativeFactory`, and `NativeRouter` utilize `OwnableUpgradeable` for ownership management. This single-step ownership transition mechanism presents a risk where a contract could inadvertently be left without an owner, either through malicious intent or accidental transfer to a zero address. Lack of ownership compromises the contract's administrative functionalities, including upgradability, making it impossible to apply future patches or improvements. This scenario can be especially detrimental for contracts that are meant to be upgradeable, as it freezes their logic and state, potentially leaving them vulnerable to exploitation or making them obsolete over time.

BVSS

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

Recommendation

To mitigate the risks associated with single-step ownership transitions and enhance contract security, it is recommended to adopt a two-step ownership transition mechanism, such as OpenZeppelin's `Ownable2Step`. This approach introduces an additional step in the ownership transfer process, requiring the new owner to accept ownership before the transition is finalized. The process typically involves the current owner calling a function to nominate a new owner, and the nominee then calling another function to accept ownership.
Implementing `Ownable2Step` provides several benefits:
1. **Reduces Risk of Accidental Loss of Ownership**: By requiring explicit acceptance of ownership, the risk of accidentally transferring ownership to an incorrect or zero address is significantly reduced.

2. **Enhanced Security**: It adds another layer of security by ensuring that the new owner is prepared and willing to take over the responsibilities associated with contract ownership.

3. **Flexibility in Ownership Transitions**: Allows for a smoother transition of ownership, as the nominee has the opportunity to prepare for the acceptance of their new role.
By adopting `Ownable2Step`, contract administrators can ensure a more secure and controlled process for transferring ownership, safeguarding against the risks associated with accidental or unauthorized ownership changes.

Partially Solved: This change would impact the `NativeRouter`, `NativePool`, `NativePoolFactory`, they are already deployed and live on many chains. So we incline not to change them. For `AquaVault` and `NativeRfqPool`, changes were made on commit hash `76873a37ffd2c7af2678abe733627e108b52d902.`

Remediation Plan

PARTIALLY SOLVED: The **Native team** partially solved as the change would impact the `NativeRouter`, `NativePool`, `NativePoolFactory`, which are already deployed and live on many chains. For `AquaVault` and `NativeRfqPool`, the change was made under `76873a37ffd2c7af2678abe733627e108b52d902`

Remediation Hash

76873a37ffd2c7af2678abe733627e108b52d902

7.8 NON-VALIDATED POOL UPGRADES

// LOW

Description

The functions `upgradePool` and `upgradePools` in the `NativePoolFactory` contract allow for the upgrade of pool implementations to new versions. While this feature is crucial for the evolution and maintenance of the pool contracts, it introduces a significant risk: if an upgrade is performed to an incorrect or incompatible implementation, it could render the pool contracts non-functional. This risk is heightened by the current lack of validation checks on the new implementation, relying solely on the discretion of the admin performing the upgrade.
Without validating that the new implementation adheres to expected interfaces or contains specific functional signatures, there's a potential for severe disruptions in the pool's operations. Such disruptions could range from the loss of critical functionality to the complete halting of the pool's ability to process transactions, potentially leading to financial losses and undermining user trust in the platform.

BVSS

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

Recommendation

To mitigate the risks associated with invalidated pool upgrades, the following measures are recommended:
1. **Implementation Validation Checks**: Introduce validation checks within the `_upgradePool` internal function to verify that the new implementation is compatible and functional. This could involve checking for the presence of specific function signatures or calling a designated function in the new implementation that returns a known value, indicating compatibility.
2. **Safe Upgrade Patterns**: Consider adopting established upgrade patterns that include safety checks, such as the use of a proxy pattern with a "test call" to the new implementation, before finalizing the upgrade. This approach allows for the verification of the new implementation's functionality in a controlled manner.
By incorporating these recommendations, the `NativePoolFactory` can significantly reduce the risks associated with pool upgrades, ensuring that new implementations enhance the platform's functionality without compromising its stability and reliability.

Remediation Plan

SOLVED: The **Native team** solved this issue.

Remediation Hash

587ae3c7f551a9c5538501b694564235911c29cb

7.9 MISSING VERIFICATION OF AQUAVAULT ADDRESS IN AQUALPTOKEN DURING MARKET SUPPORT

// LOW

Description

The `supportMarket` function in the AquaVault smart contract does not verify if the `aquaLpToken` contract has its `aquaVault` property set to the current AquaVault contract address (àddress(this)). This verification is crucial to ensure that the LP Token contract is correctly linked to and callable from the AquaVault contract. Without this check, there's a risk that an LP Token might be supported without having a proper operational link back to the AquaVault, leading to potential operational issues where the AquaVault expects to interact with the LP Token but cannot due to the incorrect setup.

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 vulnerability, it is recommended to introduce a verification step in the `supportMarket` function that checks if the `aquaLpToken`'s associated `aquaVault` property or equivalent is set to `address(this)`, ensuring the LP Token is correctly linked to the AquaVault contract. This could be done by defining a function or public variable in the `AquaLpToken` contract that returns the address of the associated AquaVault, and then calling this function within `supportMarket` to verify the address. If the verification fails, the contract should revert the transaction to prevent unsupported or incorrectly configured LP Tokens from being added. This ensures integrity in the linkage between AquaVault and its supported markets, preventing operational failures and maintaining the contract's security posture.

Remediation Plan

SOLVED: The **Native team** solved this issue.

Remediation Hash

d7daa6b78fa7e0568c2f9441b45ec48b9b545031

7.10 INCONSISTENT ETH REFUND HANDLING

// LOW

Description

The `NativeRouter` contract's `exactInputSingle` and `exactInput` functions both involve operations that can lead to ETH refunds to the caller. However, they handle these refunds inconsistently, which may lead to unintended behavior or inefficiencies:
- **`exactInputSingle`** refunds ETH using `address(this).balance`, potentially sending all ETH held by the contract to the caller.
- **`exactInput`**, conversely, calculates the refund based on `state.initialEthBalance` and the current balance, aiming to refund only the excess ETH sent for the transaction.
This discrepancy can lead to situations where `exactInputSingle` might inadvertently refund more ETH than intended, especially if the contract holds ETH for reasons other than the current transaction. Such behavior not only introduces inefficiency by moving unrelated ETH but can also complicate the contract's financial logic and tracking.

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 harmonize the behavior of these functions and ensure that only the correct amount of ETH is refunded, the following modifications are recommended:
1. **Consistent Refund Calculation**: Align the ETH refund logic in both functions to use a consistent method that refunds only the excess ETH sent by the caller for the transaction. This approach minimizes the risk of unintentionally depleting the contract's ETH balance.
2. **Implement Refund Utility Function**: Create a utility function within the contract that calculates the excess ETH to be refunded based on the initial balance and the amount spent during the transaction. Use this utility function in both `exactInputSingle` and `exactInput` to handle refunds. This ensures consistency and simplifies the contract code and facilitates future maintenance or adjustments.
3. **Documentation and Comments**: Update the contract documentation and inline comments to clearly explain the refund logic, including how the contract calculates and executes ETH refunds. Clear documentation ensures that future developers and auditors can easily understand and verify the contract's behavior.
Adopting these recommendations will help ensure that the `NativeRouter` contract handles ETH refunds consistently and accurately, minimizing risks related to ETH management and improving the overall robustness of the contract's financial operations.

Remediation Plan

SOLVED: The **Native team** solved this issue.

Remediation Hash

c5ddcc494a75630943708fcc6c24c8e8386f6031

7.11 HARDCODED FUNCTION SELECTORS RISK

// INFORMATIONAL

Description

In the `NativePoolFactory` contract, the use of hardcoded function selectors (`INIT_RFQ_SELECTOR`, `UPGRADE_SELECTOR`, and `INIT_SELECTOR`) poses a maintainability and correctness risk. These selectors are meant to correspond to specific function signatures for initializing and upgrading pools. If the signatures of these functions change—for instance, due to a contract upgrade or functionality modification—the hardcoded selectors would no longer match, potentially leading to incorrect behavior or the inability to call the intended functions.
Hardcoded selectors require manual updates whenever the associated function signatures are modified, increasing the risk of human error and oversight. This could lead to discrepancies between the actual function selectors and the hardcoded values, disrupting the contract's intended operations.

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 mitigate these risks and enhance the contract's maintainability and correctness, it is recommended to generate function selectors dynamically using the Solidity compiler's built-in `.selector` property. This approach ensures that the selectors are automatically updated to match their corresponding function signatures, eliminating the manual maintenance burden and reducing the potential for errors. The modifications would involve replacing the hardcoded constants with dynamic references, as shown below:
- **Dynamic Selector Generation**:
- Replace the hardcoded `INIT_SELECTOR`, `INIT_RFQ_SELECTOR`, and `UPGRADE_SELECTOR` with dynamic selectors derived from the actual functions. For example, if there's a function `initializePool` in a contract `PoolContract`, use `PoolContract.initializePool.selector` to dynamically obtain the selector.
Implementing dynamic selector generation enhances the contract's resilience to changes and reduces the likelihood of operational issues caused by outdated or incorrect selectors. This approach aligns with best practices for contract development, promoting code maintainability and reliability.
- **Example Implementation**:

```
contract PoolContract {

    function initializePool(/* parameters */) external {/* implementation */}

}

contract NativePoolFactory {

    // Dynamically obtain the selector
```

```
    bytes4 public constant INIT_SELECTOR = PoolContract.initializePool.selector;

    // Similar changes for INIT_RFQ_SELECTOR and UPGRADE_SELECTOR

}
```

Remediation Plan

SOLVED: The **Native team** solved this issue.

Remediation Hash

a143103bb952a015dcf8c306d247552ee0769c28

7.12 LACK OF REGISTERED POOL VALIDATION

// INFORMATIONAL

Description

The `NativePoolFactory` contract introduces functionalities to pause and upgrade pools through the `pausePools`, `upgradePool`, and `upgradePools` functions. However, these functions currently lack validation to ensure that the pools being operated on are indeed registered within the factory's `pools` mapping. This oversight means that arbitrary pool addresses, which are not necessarily created or managed by the factory, could be passed to these functions. While this does not pose a direct security risk in terms of asset loss or contract exploitation, it could lead to unintended behavior, such as attempting to pause or upgrade non-existent or unrelated pools. This misalignment could result in wastage of gas fees, administrative confusion, and potential disruption of intended factory operations.

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 vulnerability and ensure that only valid, factory-registered pools can be paused or upgraded, it is recommended to implement the following changes:
1. **Validate Pool Registration**: Amend the `pausePools`, `upgradePool`, and `upgradePools` functions to include a check that verifies each provided pool address against the factory's `pools` mapping. If an address is not recognized as a registered pool, the function should revert the transaction.
2. **Error Handling and Feedback**: Provide clear error messages for revert conditions to help administrators understand why a transaction failed. For example, use a revert reason like "Pool is not registered" when an unregistered pool address is provided.
By implementing these recommendations, the `NativePoolFactory` can improve the robustness and reliability of its pool management functionalities, ensuring that only legitimate, registered pools are subject to administrative operations.

Remediation Plan

SOLVED: The **Native team** solved this issue.

Remediation Hash

2f3ac3940c08fe4a32e9c1801836e11f1ce0fd09

7.13 IMPROPER HANDLING OF ETH REFUNDS

// INFORMATIONAL

Description

The `refundETHRecipient` function in the `NativeRouter` contract is designed to transfer the entire ETH balance of the contract to a specified recipient. This functionality lacks flexibility and may not align with the intended use cases, such as partial refunds or managing ETH balances for different purposes within the same contract. Refunding the entire contract balance can lead to unintended consequences, including disrupting the contract's operational liquidity or inadvertently transferring ETH meant for other functions or commitments.

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 potential issues and enhance the functionality of the `refundETHRecipient` method, the following changes are recommended:

1. **Allow Specifying Refund Amount**: Modify the `refundETHRecipient` function to accept an additional parameter specifying the amount of ETH to be refunded. This change provides greater control over the refund process, enabling partial refunds and preventing the unintended transfer of the entire contract balance.
2. **Event Logging**: Emit an event upon successful execution of a refund, detailing the recipient address and the refunded amount. Event logging facilitates tracking and auditing of refund transactions, enhancing transparency and accountability.

By adopting these recommendations, the `NativeRouter` contract can achieve a more secure, flexible, and controlled approach to managing ETH refunds, thereby reducing risks and accommodating a wider range of operational scenarios.

Remediation Plan

SOLVED: The **Native team** solved this issue.

Remediation Hash

aad00b455ae52c30bd0e629183fba0a310e79b5f

7.14 TYPOS AND DOCUMENTATION

// INFORMATIONAL

Description

NativeRouter

- `ExcatInputExecutionState` is a typo. The correct spelling should be `ExactInputExecutionState`. This typo is found in function signatures and indicates a misspelling of a struct intended for tracking execution state.

AquaVaultLogic

1. `/// @notice Extract operational logic from AquaVault to reduce contract size. The operation would be delegateCall from AquaVault`

- Typo: "delegateCall" should be "ðelegatecall" (to match Solidity's function name exactly) and consider clarifying as "The operation would be performed via a `delegatecall` from AquaVault."

2. `/// @dev The funciton can be used to provide allowance and also revoke allowance (put amount is 0)`

- Typo: "funciton" should be "function".

3. `/// @dev Update amount of netSwapBorrow should be the sum of all traders on that token, excluing reserve:`

- Typo: "excluing" should be "excluding".

4. `// if the existing posistion is 0, position shouldn't be updated, revert by denom 0`

- Typo: "posistion" should be "position".

5. `// check here that LP price doesn't increase for more than 1%`

- Suggestion for clarity: Consider rephrasing to "Check that the LP price does not increase by more than 1%."

6. `/// @notice logic of repaying the short positions`

- Grammatical consistency: Consider capitalizing the first word for consistency with other comments, i.e., "Logic of repaying the short positions."

7. `/// @notice logic of settling for long and short positions`

- Similar to the previous point, consider capitalizing for consistency: "Logic of settling for long and short positions."

8. `/// @notice logic of adding collateral`

- Again, for consistency: "Logic of adding collateral."

9. `/// @notice logic of removing collateral`

- Consistency: "Logic of removing collateral."

10. `/// @notice logic of liquidating positions`

- Consistency: "Logic of liquidating positions."

AquaVault

1. `/// @notice Called by NativeRfqPool to update positions and LP token borrow values`

- Typo: "AuqaLpToken" in the comment `/// @notice Called by AuqaLpToken to transfer the underlying asset to recipient` should be corrected to "AquaLpToken".

2. Modifier `NonZero` might conventionally be named as `nonZero` to align with Solidity's naming conventions, which typically start function and modifier names with a lowercase letter.

3. `/// @notice Called by admin to whilselist or blacklist a trader`

- Typo: "whilselist" should be "whitelist".

4. Comment consistency and clarity could be improved across the board. For example, the `/// @dev` and `/// @notice` comments could be made more descriptive and corrected for typos to enhance understanding of the contract functions and their purposes.
5. `function setAdmin(address newAdmin) external onlyAdmin NonZero(newAdmin) {`
- The modifier `NonZero` is used correctly, but ensuring that it's clear from its name that it checks for non-zero addresses might improve code readability. Consider renaming it to `ensureNonZeroAddress` or adding a comment to describe its purpose more explicitly if not renaming.
6. `/// @notice Called by epoch updater to update the LP token borrow value`
- The comment ends abruptly with "d" at `@param traderPositionUpdate d`. It should be completed for clarity.
7. The contract generally follows good Solidity practices, but ensuring consistency in comment quality and addressing the minor typos will improve its readability and maintainability.

NativeRfqPool

1. In the comment:
- Original: `// This follows the exsiting NativePool order singature format`
- Corrected: `// This follows the existing NativePool order signature format`
- "exsiting" should be "existing".
- "singature" should be "signature".

NativePoolFactory

1. The custom error message `onlyOwnerOrPauserCanCall()` used in the `onlyOwnerOrPauser` modifier does not follow the convention of starting error messages with a capital letter. Moreover, it lacks explicit declaration as a custom error at the top of the contract, which would improve gas efficiency over revert strings if that was the intention. If it's meant to be a revert string, it should be in quotes.
2. In the `createNewPool` function:
- The error message `NotMultiPoolTreasuryAndBindedToOtherPool(address treasuryAddress)` contains a typo: "Binded" should be "Bound" to be grammatically correct.
3. In the `setRegistry` function, the error message `ZeroAddressInput()` is clear but could be more descriptive, e.g., `RegistryAddressCannotBeZero()` to explicitly state the context of the error.
4. Throughout the contract, the consistency of checking zero addresses could be improved by using a modifier for this purpose, as seen with `onlyOwnerOrPauser`. This would reduce code duplication and improve readability.
5. The use of unchecked blocks in loops, such as in `pausePools` and `upgradePools`, is a conscious choice to optimize gas. However, including a brief comment about this choice could be beneficial for future readers and maintainers of the code, explaining that the risk of overflow is considered negligible.
6. The documentation through comments is minimal, particularly for public functions and events. Expanding on the comments to explain the purpose and functionality of key functions and the rationale behind certain design choices could greatly aid understanding and maintainability.
7. `createNewRfqPool`'s variable `rfgPoolImpl` might be a typo and should possibly be `rfqPoolImpl`, assuming it stands for "Request for Quote Pool Implementation". Consistency in naming helps avoid confusion.

AquaVaultSignatureCheck

- Original: `/// The nonce does not follow an incremental pattern so the order does need to be excuted in order.`
- Correction: `/// The nonce does not follow an incremental pattern so the order does not need to be executed in order.`

- "excuted" should be corrected to "executed".
- It seems there's a missing "not" before "need to be executed", based on the context suggesting that the order of execution is not required due to the non-incremental pattern of the nonce.

Score

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

Recommendation

It is recommended to fix the typos in the code in order to improve its readability and maintainability.

Remediation Plan

SOLVED: The **Native team** solved this issue.

Remediation Hash

`d1abdc8be8d425c067c61add13fc578ce62c7021`

8. REVIEW NOTES

AquaLpToken.Sol

- Does extend a custom `CToken`:
- No `reentrancy` issue on `doTransferOut` on the `borrowFresh` function.
- It does not implement `exchangeRateStoredInternal` or `accrueInterest`, the `AquaLpToken` will implement it.
- All functions relying on `accrualBlockNumber` are moved to `accrualBlockTimestamp`. It uses timestamps instead. Timestamps comparison is not recommended. However, in ethereum all blocks will have a different timestamp and higher than parent. This means that two transactions on the same block cannot have a different timestamp.
- Does extend a custom `CErc20`:
- The `sweepToken` is modified with before/after balance checks.
- `getCashPrior`, `doTransferIn` and `doTransferOut` are left to the `AquaLpToken` for implementation.

> According to the [Ethereum Yellow paper](https://ethereum.github.io/yellowpaper/paper.pdf), there is no limit to how far the blocks are in time, but each timestamp should be greater than the timestamp of its parent. Consequently, it explains that if the state of the time-dependent event can vary by 15 seconds and maintain integrity, it is safe to use the block.timestamp parameter in the contract.

- Valid `initialize` process for `CErc20` and `CToken`. Same code used, adding `aquaVault` storage.
- **Anyone can create an `AquaLpToken` and `aquaVault`.**
- `_authorizeUpgrade` is limited to only admin and verifies that the new implementation is really a `isCToken`.
- `getCashPrior` will return the balance of the `aquaVault`.
- `doTransferIn` does transfer to the vault instead and uses `safeTransferFrom`.
- `doTransferOut` will use the vault `pay` function to pay the `to` address.
- It should be extremely caution with `pay` function, as there is no reentrancy protection.

AquaVaultSignatureCheck.Sol

- Does extend OZ `EIP712` contract.
- `setAquaVault` only callable by admin.
- All `verify*` functions are using the `onlyAquaVault` modifier.
- `verifySettleSignature`:
- Will check the `deadline` with current timestamp.
- Verify that the nonce hasn't been used `isNonceUsed`.
- The nonce look random generated, sandwitch attacks could be a possibility as orders are not FIFO verifiable.
- All `SettlementRequest` struct data is used as the `msgHash`.
- `verifyRemoveCollateralSignature`:
- Will check the `deadline` with current timestamp.
- Verify that the nonce hasn't been used `isNonceUsed`. (same mapping as `verifySettleSignature`)
- All `RemoveCollateralRequest` struct data is used as the `msgHash`.
- `verifyLiquidationSignature`:
- Same logic check steps as `verifySettleSignature` and `verifyRemoveCollateralSignature`. All the `LiquidationRequest` data is being verified.

- There could be type confusion between `SettlementRequest` and `RemoveCollateralRequest` signatures. It should be possible to interchange request types as they have the same datatypes and same positions.

AquaVault.Sol And AquaVaultLogic.Sol

- It is using a customized `Comptroller` version were all requires are moved to custom errors. `_initializeMarket` and unused functions are removed.
- `_authorizeUpgrade` is only on admin hands. Also the new address i verified to be a `comptroller`.
- `onlyTraderOrSettler` will prevent from executing a request if the sender is not a trader or the call isn't being executed by the `traderSettlers` for that trader.
- The actions are signed by `signer`, set with `setSigner` by admin only.
- **This address has the power to control any user address. Trader is not providing neither participating in the signature generation/verification on the process.**
- `liquidate` can only be called by a liquidator trader.
- The `setAllowance` could probably verify that the `markets[token].isListed` is supported. (only admin functionality).
- `addCollateral` will iterate over the list of tokens, verify they are part of a valid market and transfer the users amount to the `AquaVault`. The collateral can be credited to a non-sender trader.
- `removeCollateral`, will verify the `RemoveCollateralRequest` action and signature using `AquaVaultSignatureCheck`. It will not have underflow as per compiler version when decrementing the amounts.
- If the trader is no longer whitelisted the credit for liquidation is validated, otherwise this is delegated entirely to the off-chain API that will provide a signature.

```
    │   └─ ← ()
    ├─ [12911] 0x2f08910E0eaE2d057fB8b0160E86ff505E584832::removeCollateral(RemoveCollateralRequest({ nonce: 1, deadline: 1000000000 [1e9], trader: 
0x0000000000000000000000000000000000010001, tokens: [TokenAmountUint({ tokenAddress: 0x72384992222BE015DE0146a6D7E5dA0E19d2Ba49, amount: 10 })], rec
ipient: 0x0000000000000000000000000000000000010001 }), 0x)
    │   ├─ [12677] AquaVault::removeCollateral(RemoveCollateralRequest({ nonce: 1, deadline: 1000000000 [1e9], trader: 0x000000000000000000000000000
0000000010001, tokens: [TokenAmountUint({ tokenAddress: 0x72384992222BE015DE0146a6D7E5dA0E19d2Ba49, amount: 10 })], recipient: 0x0000000000000000000
0000000000000000010001 }), 0x) [delegatecall]
    │   │   ├─ [0] AquaVaultSignatureCheck::verifyRemoveCollateralSignature(RemoveCollateralRequest({ nonce: 1, deadline: 1000000000 [1e9], trader: 
0x0000000000000000000000000000000000010001, tokens: [TokenAmountUint({ tokenAddress: 0x72384992222BE015DE0146a6D7E5dA0E19d2Ba49, amount: 10 })], rec
ipient: 0x0000000000000000000000000000000000010001 }), 0x, ADMIN: [0x183530D9DBB051e4894686C261F39f3aA82aBD83])
    │   │   │   └─ ← 0x0000000000000000000000000000000000000000000000000000000000000001
    │   │   ├─ [2717] AquaVaultLogic::abbe0382(0000000000000000000000000000000000000000000000000000000000a00000000000000000000000000000000000
000000000000000000000001000000000000000000000000000000000000000000007900000000000000000000000000000000000000000000000000000000000000
00000090000000000000000000000000000000000000000000000000078000000000000000000000000000000000000000000000000000000000100000000000000
0000000000072384992222be015de0146a6d7e5da0e19d2ba490000000000000000000000000000000000000000000000000000a0000000000000000000000000000000
00000000000000000000000001000100000000000000000000000000000000000000000000000100010000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000bee7563f91265b39626b09acccb543ce995cad90000000000000000000000000000000000000000000000000000000a0000000000
00000000000000000000000000000000000000000200000000000000000000000072384992222be015de0146a6d7e5da0e19d2ba49000000000000000000000000007b0aa
1e6fcd181d45c94ac62901722231074d8d4) [delegatecall]
    │   │   │   └─ ← 0x4e487b710000000000000000000000000000000000000000000000000000000000000011
    │   │   └─ ← panic: arithmetic underflow or overflow (0x11)
    │   └─ ← panic: arithmetic underflow or overflow (0x11)
    └─ ← panic: arithmetic underflow or overflow (0x11)

Test result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 2.32ms

Ran 1 test suites: 0 tests passed, 1 failed, 0 skipped (1 total tests)

Failing tests:
Encountered 1 failing test in test/halborn/AquaVaultHalborn.t.sol:AquaVaultHalborn
```

- `getCredit` will validate the trader position. It does iterate over all `allMarkets` array. Since this array cannot remove elements all positions are verified.
- If the market doesn't have a position or collateral it is skipped.
- If the market doesn't have an oracle, the trader using that market should be "whitelisted" and the market not supported on-chain.
- **There is a critical risk that if a none on-chain supported market is ever added/supported on-chain ALL none-whitelisted traders will not be able to perform an action that would reduce the collateral factor or withdraw tokens/collaterals. This is due the oracle returning zero and erroring. This is not reversible as supported markets cannot be removed.**

- If the trader has a position for this token it will calculate a credit value. All tokens/prices are set to the same amount of decimals. This means that adding/subtracting from credit value with the `getUnderlyingPrice` data is safe. A token whose decimal amount is `x` less, its credit is increased by a factor of `1ex`. If the position is long, it is factored using `collateralFactorMantissa` for the `lpToken` and `ETHER_SCALE` to remove the mantissa precision. The entire value is then factored down by `ETHER_SCALE` again.
- `liquidate` is only callable by liquidators and via a valid signature of the action. If the trader is whitelisted, the signature is assumed valid and liquidation will take place anyway without verifying the credit score. After the collateral decrease due to liquidation the position for the trader is verified again to be in a valid state.
- `settle` will verify the signature as this operation can withdraw funds. The position is first updated and then the credit validated if not a whitelisted trader. Afterward, for each position update the funds are transferred from/to depending on the value. If amount is added, then funds are transferred from the caller to the trader position, otherwise funds are transferred from the vault to the recipient.
- `repay` does allow to increase the position health by providing a new amount, the code will check that the position is not positive afterwards, it can only reduce debt and increase health. **However, the code does NOT update the `netBorrow` amount.**
- `positionEpochUpdate` can be called every 8 hours. It does allow the `epochUpdater` to set a new value for the traders positions, the change cannot be more than `1%` on either side, increase or decrease.
- For the `lpTokenValueUpdate` array, the netBorrow stored value cannot decrease, only increase by a `1%`. Since `exchangeRateStored` does also factor the `totalReserves` the reserve update is also indirectly checked here.

ChainlinkPriceOracle.Sol

- `setPriceFeed` can set a feeds oracle token price.
- The decimal feeds for chainlink are less than `18` decimals. Which will prevent any underflow on `getUnderlyingPrice`.
- `getUnderlyingPrice` does correctly factor in decimal for feed and token.

NativePoolFactory.Sol

- Does inherit from `NoDelegateCallUpgradable`, but does not use any of the functionalities.
- `_authorizeUpgrade` is restricted to owner.
- `pause` can be called by the owner or the pauser (set by the owner).
- `setPoolImplementation` allows to set the factory implementation reference.
- The `addMultiPoolTreasury` and `removeMultiPoolTreasury` functions can be used to mark a treasury address as usable by multiple pools.
- On the `createNewPool` function the init selector is correct and does match the signature. It is recommended that the `NativePoolFactory` variables `bytes4 public constant` for selectors to be dynamic from the actual selector in case the init signature ever changes.
- `upgradePool` and its batchable version `upgradePools`, allows to upgrade the pool to a given implementation. There is the risk that those upgrades leave the contract into a un-funcional state. Admins should take extra precautions on the implementation set.

NativeRfqPool.Sol

- Can receive native tokens.

- UUPS upgrading can only be done by the factory. It also checks for `isNativeRfqPool` on the new implementation pool.
- `poolFactory` is set to the caller of `initialize`. The factory does perform this action atomically.
- Updaters and setters do have ownership/modifier checks.

NativeRouter.Sol

- Can receive funds, only from `WETH9` (unwrapping).
- `setWidgetFeeSigner` set the signature address to check against, only owner.
- `_authorizeUpgrade` to only owner.
- `setWeth9Unwrapper` to set `WETH9` unwrapper, only owner.
- `pause` can be called by the owner or a pauser address. Meanwhile, `unpause` is owner restricted.
- `onlyEOAorWhitelistContract` is valid implemented:

`MSG.SENDER == TX.ORIGIN`	`CONTRACTCALLERWHITELISTENABLED`	`CONTRACTCALLERWHITELIST[MSG.SENDER]`	RESULT
True	True or False	True or False	Does not revert (EOA call)
False	False	True or False	Does not revert (Contract call, whitelist check
False	True	True	Does not revert (Whitelisted contract call)
False	True	False	Reverts (Unwhitelisted contract call)

- `setExternalRouterWhitelist` allows setting multiple external routers at once. Only owner.
- `processWidgetFee`: If the user has paid, the funds are transferred from the contract (which funds are coming from callers wrapping), otherwise from the caller.
- **There is a discrepancy on `exactInputSingle` and `exactInput` native funds transfer with `safeTransferETH`. Moreover, code is not using the before/after swap balances.**
- `swapCallback` will verify that the caller is really the order buyer. The callback origin is verified being a pool with `verifyCallback` .
- `exactInputSingle`:
- `hasMultiplePools` will check for bytes being larger than `HOP_SIZE`.
- If `msg.value` is given, it is wrapped in `WETH` and the state set to `hasAlreadyPaid`.
- The order caller should be the sender.
- `exactInput`:
- `verifyWidgetFeeSignature` and `amountIn` checks.
- Decodes the first order and performs `sellerToken` and `amountIn` checks. If `msg.value` it does wrap using `WETH`.
- The order caller should be the sender.
- The first `verifyWidgetFeeSignature` does implicitly check all orders for being valid, no need to verify each order on the loop.
- It will correctly skip order until `hasMultiplePools` returns false. The `amountOut` is set to the latest order swapped.
- `effectiveSellerTokenAmount` can not be higher than `sellerTokenAmount`. The latter is verified on a signature.
- `deadlineTimestamp` is also verified for expiration.
- `tradeRFQT` will verify if `msg.value` is provided, if so, it will wrap `WETH`. Otherwise `sellerToken` will be used (multiHop).

- Will verify the internal quote signature and update nonce, marking it as already used.

9. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified the smart contracts in the repository and was able to compile them correctly into their ABIs and binary format, Slither was run against the 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.

The security team assessed all findings identified by the Slither software, however, findings with severity `Information` and `Optimization` are not included in the below results for the sake of report readability.

```
INFO:Detectors:
NativeRfqPool._transferFromTreasury(address,address,uint256) (contracts/NativeRfqPool.sol#172-180) uses arbitrary from in transferFrom: IERC20(weth).safeTransferFrom(treasury,address(this),value) (contracts/NativeRfqPool.sol#174)
NativeRfqPool._transferFromTreasury(address,address,uint256) (contracts/NativeRfqPool.sol#172-180) uses arbitrary from in transferFrom: IERC20(token).safeTransferFrom(treasury,receiver,value) (contracts/NativeRfqPool.sol#178)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
NativePool.executeSwapFromTreasury(uint256,Orders.Order,address) (contracts/NativePool.sol#444-453) uses arbitrary from in transferFrom: TransferHelper.safeTransferFrom(_order.buyerToken,treasury,recipient,amount) (contracts/NativePool.sol#449)
NativeRfqPool._transferFromTreasury(address,address,uint256) (contracts/NativeRfqPool.sol#172-180) uses arbitrary from in transferFrom: IERC20(weth).safeTransferFrom(treasury,address(this),value) (contracts/NativeRfqPool.sol#174)
NativeRfqPool._transferFromTreasury(address,address,uint256) (contracts/NativeRfqPool.sol#172-180) uses arbitrary from in transferFrom: IERC20(token).safeTransferFrom(treasury,receiver,value) (contracts/NativeRfqPool.sol#178)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
NativePool.executeSwapFromTreasury(uint256,Orders.Order,address) (contracts/NativePool.sol#444-453) uses a dangerous strict equality:
        - require(bool,string)((treasuryBalanceInitial - treasuryBalanceFinal) == amount,Swap amount not match) (contracts/NativePool.sol#452)
NativePool.executeSwapToTreasury(Orders.Order,uint256,uint256,bytes) (contracts/NativePool.sol#455-482) uses a dangerous strict equality:
        - require(bool,string)((treasuryBalanceFinal - treasuryBalanceInitial) == sellerTokenAmount,Swap amount not match) (contracts/NativePool.sol#479)
ExternalSwapRouterUpgradeable.calculateTokenAmount(uint256,Orders.Order) (contracts/ExternalSwapRouterUpgradeable.sol#42-62) uses a dangerous strict equality:
        - sellerTokenAmount == 0 || buyerTokenAmount == 0 || flexibleAmount == 0 (contracts/ExternalSwapRouterUpgradeable.sol#49)
ExternalSwapRouterUpgradeable.calculateTokenAmount(uint256,Orders.Order) (contracts/ExternalSwapRouterUpgradeable.sol#42-62) uses a dangerous strict equality:
        - buyerTokenAmount == 0 (contracts/ExternalSwapRouterUpgradeable.sol#58)
ExternalSwapRouterUpgradeable.externalSwap(Orders.Order,uint256,address,address,bytes) (contracts/ExternalSwapRouterUpgradeable.sol#64-118) uses a dangerous strict equality:
        - flexibleAmount == 0 (contracts/ExternalSwapRouterUpgradeable.sol#75)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
NativeRouter.exactInputInternal(uint256,address,NativeRouter.SwapCallbackData,bytes).amount1Delta (contracts/NativeRouter.sol#345) is a local variable never initialized
NativeRouter.exactInput(INativeRouter.ExactInputParams).state (contracts/NativeRouter.sol#231) is a local variable never initialized
NativeRouter.exactInputInternal(uint256,address,NativeRouter.SwapCallbackData,bytes).amount0Delta (contracts/NativeRouter.sol#344) is a local variable never initialized
NativeRouter.exactInput(INativeRouter.ExactInputParams).fallbackSwapData (contracts/NativeRouter.sol#263) is a local variable never initialized
NativeRouter.exactInputSingle(INativeRouter.ExactInputParams).state (contracts/NativeRouter.sol#174) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
NativeRouter.swapCallback(int256,int256,bytes) (contracts/NativeRouter.sol#107-126) ignores return value by (order) = data.orders.decodeFirstOrder() (contracts/NativeRouter.sol#117)
NativeRouter.swapCallback(int256,int256,bytes) (contracts/NativeRouter.sol#107-126) ignores return value by CallbackValidation.verifyCallback(factory,order.buyer) (contracts/NativeRouter.sol#122)
NativeRouter.exactInputSingle(INativeRouter.ExactInputParams) (contracts/NativeRouter.sol#156-215) ignores return value by (order) = params.orders.decodeFirstOrder() (contracts/NativeRouter.sol#172)
NativeRouter.exactInput(INativeRouter.ExactInputParams) (contracts/NativeRouter.sol#218-307) ignores return value by (order) = params.orders.decodeFirstOrder() (contracts/NativeRouter.sol#229)
NativeRouter.exactInput(INativeRouter.ExactInputParams) (contracts/NativeRouter.sol#218-307) ignores return value by (order,None) = params.orders.decodeFirstOrder() (contracts/NativeRouter.sol#289)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

```
INFO:Detectors:
NativePool.executeSwapFromTreasury(uint256,Orders.Order,address) (contracts/NativePool.sol#444-453) uses arbitrary from in transferFrom: TransferHelper.safeTransferFrom(_order.buyerToken,treasury,recipient,amount) (contracts/NativePool.sol#449)
NativeRfqPool._transferFromTreasury(address,address,uint256) (contracts/NativeRfqPool.sol#172-180) uses arbitrary from in transferFrom: IERC20(weth).safeTransferFrom(treasury,address(this),value) (contracts/NativeRfqPool.sol#174)
NativeRfqPool._transferFromTreasury(address,address,uint256) (contracts/NativeRfqPool.sol#172-180) uses arbitrary from in transferFrom: IERC20(token).safeTransferFrom(treasury,receiver,value) (contracts/NativeRfqPool.sol#178)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
NativePool.executeSwapFromTreasury(uint256,Orders.Order,address) (contracts/NativePool.sol#444-453) uses a dangerous strict equality:
        - require(bool,string)((treasuryBalanceInitial - treasuryBalanceFinal) == amount,Swap amount not match) (contracts/NativePool.sol#452)
NativePool.executeSwapToTreasury(Orders.Order,uint256,uint256,bytes) (contracts/NativePool.sol#455-482) uses a dangerous strict equality:
        - require(bool,string)((treasuryBalanceFinal - treasuryBalanceInitial) == sellerTokenAmount,Swap amount not match) (contracts/NativePool.sol#479)
ExternalSwapRouterUpgradeable.calculateTokenAmount(uint256,Orders.Order) (contracts/ExternalSwapRouterUpgradeable.sol#42-62) uses a dangerous strict equality:
        - sellerTokenAmount == 0 || buyerTokenAmount == 0 || flexibleAmount == 0 (contracts/ExternalSwapRouterUpgradeable.sol#49)
ExternalSwapRouterUpgradeable.calculateTokenAmount(uint256,Orders.Order) (contracts/ExternalSwapRouterUpgradeable.sol#42-62) uses a dangerous strict equality:
        - buyerTokenAmount == 0 (contracts/ExternalSwapRouterUpgradeable.sol#58)
ExternalSwapRouterUpgradeable.externalSwap(Orders.Order,uint256,address,address,bytes) (contracts/ExternalSwapRouterUpgradeable.sol#64-118) uses a dangerous strict equality:
        - flexibleAmount == 0 (contracts/ExternalSwapRouterUpgradeable.sol#75)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
NativeRouter.exactInput(INativeRouter.ExactInputParams).fallbackSwapData (contracts/NativeRouter.sol#263) is a local variable never initialized
NativeRouter.exactInputSingle(INativeRouter.ExactInputParams).state (contracts/NativeRouter.sol#174) is a local variable never initialized
NativeRouter.exactInputInternal(uint256,address,NativeRouter.SwapCallbackData,bytes).amount1Delta (contracts/NativeRouter.sol#345) is a local variable never initialized
NativeRouter.exactInput(INativeRouter.ExactInputParams).state (contracts/NativeRouter.sol#231) is a local variable never initialized
NativeRouter.exactInputInternal(uint256,address,NativeRouter.SwapCallbackData,bytes).amount0Delta (contracts/NativeRouter.sol#344) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
INFO:Detectors:
NativeRouter.swapCallback(int256,int256,bytes) (contracts/NativeRouter.sol#107-126) ignores return value by (order) = data.orders.decodeFirstOrder() (contracts/NativeRouter.sol#117)
NativeRouter.swapCallback(int256,int256,bytes) (contracts/NativeRouter.sol#107-126) ignores return value by CallbackValidation.verifyCallback(factory,order.buyer) (contracts/NativeRouter.sol#122)
NativeRouter.exactInputSingle(INativeRouter.ExactInputParams) (contracts/NativeRouter.sol#156-215) ignores return value by (order) = params.orders.decodeFirstOrder() (contracts/NativeRouter.sol#172)
NativeRouter.exactInput(INativeRouter.ExactInputParams) (contracts/NativeRouter.sol#218-307) ignores return value by (order) = params.orders.decodeFirstOrder() (contracts/NativeRouter.sol#229)
NativeRouter.exactInput(INativeRouter.ExactInputParams) (contracts/NativeRouter.sol#218-307) ignores return value by (order,None) = params.orders.decodeFirstOrder() (contracts/NativeRouter.sol#289)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

The findings obtained as a result of the Slither scan were reviewed. The vulnerabilities identified by Slither were determined to be false-positives.

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.