SMART CONTRACT AUDIT REPORT

for

Alpha Homora V2 for Avalanche

Prepared By: Yiqun Chen

PeckShield

November 8, 2021

Document Properties

Client	Alpha Finance Lab
Title	Smart Contract Audit Report
Target	Alpha Homora V2 for Avalanche
Version	1.0
Author	Xuxian Jiang
Auditors	Patrick Liu, Jing Wang, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 8, 2021	Xuxian Jiang	Final Release
1.0-rc	Novebmer 6, 2021	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Alpha Homora V2 for Avalanche` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Alpha Homora V2 for Avalanche

`Alpha HomoraV2` is a leading leveraged yield farming and leveraged liquidity providing protocol and the current version has seamless integration with the `Cream Finance` lending protocol. With the planned deployment on `Avalanche`, it enables lenders to earn high interest and the lending interest rate comes from leveraged yield farmers borrowing to yield farm (or provide liquidity). From another perspective, yield farmers can get even higher farming `APY` and trading fees `APY` from taking on leveraged yield farming positions. The audited protocol makes a number of innovations from the earlier version by supporting multi-assets lending and borrowing, multiple farming pools (e.g., `Sushiswap`, `Uniswap`, `Pangolin`, `Curve`, `TraderJoe`, etc), and `BYOT` (`bring your own LP tokens`).

The basic information of the `Alpha Homora V2 for Avalanche` protocol is as follows:

Table 1.1: Basic Information of Alpha Homora V2 for Avalanche

Item	Description
Name	Alpha Finance Lab
Website	https://alphafinance.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 8, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/AlphaFinanceLab/alpha-homora-v2-avax-private-contract.git (fc90fe7)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/AlphaFinanceLab/alpha-homora-v2-avax-private-contract.git (ae53091)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Alpha Homora V2 for Avalanche protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	4	■■■■
Informational	2	■■
Total	8	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Audit Findings of Alpha Homora V2 for Avalanche Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Proper Liquidity Removal in ibETHRouterV2	Business Logic	Resolved
PVE-002	Low	Strengthened Validation in WLiquidityGauge::encodeId()	Coding Practices	Resolved
PVE-003	Informational	Improved Logic in SafeAggregatorOracle::getSafeETHPx()	Business Logic	Resolved
PVE-004	Informational	Improved Gas in TraderJoeSpell::removeLiquidityWMasterChef()	Coding Practices	Resolved
PVE-005	Low	Suggested Revert On Impossible Situations in CurveOracle	Coding Practices	Resolved
PVE-006	Low	Meaningful Events For Important States Change	Coding Practices	Confirmed
PVE-007	Low	Improved Validation in BasicSpell And ProxyOracle	Coding Practices	Resolved
PVE-008	Medium	Trust on Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Proper Liquidity Removal in ibETHRouterV2

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ibETHRouterV2`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

To facilitate the user interaction, the `Alpha HomoraV2 (AVAX)` protocol provides an `ibETHRouterV2` contract to efficiently add and remove the liquidity. While examining the logic, we notice one essential function `removeLiquidityETHAlpha()` needs to be revised.

To elaborate, we show below the full implementation of the `removeLiquidityToken()` function. This function implements a rather straightforward logic in firstly removing the liquidity from the `ibETHv2-Alpha` pool (lines 259-267), then sending the received `Alpha` to the designated recipient (line 268), and next withdraw the received `ibETHv2` back to the native token (lines 269 – 272). However, it comes to our attention that the `Alpha` is sent to the `msg.sender`, not the designated recipient `to` (line 268)!

```
251   function removeLiquidityETHAlpha(
252     uint liquidity,
253     uint minETH,
254     uint minAlpha,
255     address to,
256     uint deadline
257   ) external {
258     lpToken.transferFrom(msg.sender, address(this), liquidity);
259     router.removeLiquidity(
260       address(alpha),
261       address(ibETHv2),
262       liquidity,
263       minAlpha,
```

```
      0,
      address(this),
      deadline
    );
    alpha.transfer(msg.sender, alpha.balanceOf(address(this)));
    ibETHv2.withdraw(ibETHv2.balanceOf(address(this)));
    uint ethBalance = address(this).balance;
    require(ethBalance >= minETH, '!minETH');
    (bool success, ) = to.call{value: ethBalance}(new bytes(0));
    require(success, '!eth');
  }
```

Listing 3.1: `ibETHRouterV2::removeLiquidityETHAlpha()`

Recommendation Use the right recipient in the handling logic of `removeLiquidityETHAlpha()`.

Status This issue has been resolved. The team confirms that the contract is not deployed and remains unused.

3.2 Strengthened Validation in WLiquidityGauge::encodeId()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `WLiquidityGauge`
- Category: Coding Practices [6]
- CWE subcategory: CWE-561 [3]

Description

The `Alpha HomoraV2 (AVAX)` protocol has developed a number of investment-related strategies (in the name of `spells`) as well as a few `wrappers` to hold the custody of leveraged positions. One specific `wrapper` is `WLiquidityGauge`, which supports `LiquidityGauge`, the liquidity gauge contract to participate in the `Curve` liquidity mining and reward.

The protocol has a novel design in efficiently keeping track of the rewards with the `ERC1155`-based tokens. In particular, the reward information is directly encoded in the `ERC1155` token ID. We show below the two related functions to encode and decode the token ID.

```
  /// @dev Encode pid, gid, crvPerShare to a ERC1155 token id
  /// @param pid Curve pool id (10-bit)
  /// @param gid Curve gauge id (6-bit)
  /// @param crvPerShare CRV amount per share, multiplied by 1e18 (240-bit)
  function encodeId(
    uint pid,
    uint gid,
    uint crvPerShare
```

```
) public pure returns (uint) {
  require(pid < (1 << 10), 'bad pid');
  require(gid < (1 << 6), 'bad gid');
  require(crvPerShare < (1 << 240), 'bad crv per share');
  return (pid << 246)  (gid « 240)  crvPerShare;
}

/// @dev Decode ERC1155 token id to pid, gid, crvPerShare
/// @param id Token id to decode
function decodeId(uint id)
  public
  pure
  returns (
    uint pid,
    uint gid,
    uint crvPerShare
  )
{
  pid = id >> 246; // First 10 bits
  gid = (id >> 240) & (63); // Next 6 bits
  crvPerShare = id & ((1 << 240) - 1); // Last 240 bits
}
```

Listing 3.2: `WLiquidityGauge::encodeId()/decodeId()`

Our analysis shows that the `gauges ID (gid)` is currently encoded with 6 bits in the middle. However, each pool has at most `10 gauges`, which implies 4 bits should serve the purpose.

Recommendation Revise the above encoding/decoding scheme with 4 bits for `gauges ID (gid)`, instead of current 6.

Status This issue has been confirmed.

3.3 Improved Logic in SafeAggregatorOracle::getSafeETHPx()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SafeAggregatorOracle`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

The `Alpha HomoraV2 (AVAX)` protocol makes novel contributions in efficiently and reliably computing the prices of various pool tokens of `Uniswap` and `Curve`. Accordingly, the protocol comes with a number of well-designed oracle subsystem. In the following, we examine a specific one `SafeAggregatorOracle`.

In particular, we show below the key `getSafeETHPx(0)` function. This function return safe token price relative to `WAVAX`, multiplied by `2**112` with price deviation success status. It is currently designed to support at most 3 oracle sources per token. We notice when all possible three sources are deviated from the threshold, the current implementation returns `((prices[1] + prices[2])/ 2, false)` (line 67). With the purpose of returning the average price from these sources, the average for return can be improved as `(prices[1] + prices[2] + prices[3])/ 3)`.

```
21  function getSafeETHPx(address token) public view returns (uint, bool) {
22    uint candidateSourceCount = aggOracle.primarySourceCount(token);
23    require(candidateSourceCount > 0, 'no primary source');
24    uint[] memory prices = new uint[](candidateSourceCount);
25
26    // Get valid oracle sources
27    uint validSourceCount = 0;
28    for (uint idx = 0; idx < candidateSourceCount; idx++) {
29      try IBaseOracle(aggOracle.primarySources(token, idx)).getETHPx(token) returns (
          uint px) {
30        prices[validSourceCount++] = px;
31      } catch {}
32    }
33    require(validSourceCount > 0, 'no valid source');
34    for (uint i = 0; i < validSourceCount - 1; i++) {
35      for (uint j = 0; j < validSourceCount - i - 1; j++) {
36        if (prices[j] > prices[j + 1]) {
37          (prices[j], prices[j + 1]) = (prices[j + 1], prices[j]);
38        }
39      }
40    }
41    uint maxPriceDeviation = aggOracle.maxPriceDeviations(token);
42
43    // Algo:
44    // - 1 valid source --> return price
45    // - 2 valid sources
46    //     --> if the prices within deviation threshold, return average
47    //     --> else revert
48    // - 3 valid sources --> check deviation threshold of each pair
49    //     --> if all within threshold, return median
50    //     --> if one pair within threshold, return average of the pair
51    //     --> if none, revert
52    // - revert otherwise
53    if (validSourceCount == 1) {
54      return (prices[0], true); // if 1 valid source, return
55    } else if (validSourceCount == 2) {
56      return ((prices[0] + prices[1]) / 2, (prices[1] * 1e18) / prices[0] <=
          maxPriceDeviation); // return average price with price deviation status
57    } else if (validSourceCount == 3) {
58      bool midMinOk = (prices[1] * 1e18) / prices[0] <= maxPriceDeviation;
59      bool maxMidOk = (prices[2] * 1e18) / prices[1] <= maxPriceDeviation;
60      if (midMinOk && maxMidOk) {
61        return (prices[1], true); // if 3 valid sources, and each pair is within thresh,
            return median with price deviation success status
```

```
      } else if (midMinOk) {
        return ((prices[0] + prices[1]) / 2, true); // return average of pair within thresh with price deviation success status
      } else if (maxMidOk) {
        return ((prices[1] + prices[2]) / 2, true); // return average of pair within thresh with price deviation success status
      } else {
        return ((prices[1] + prices[2]) / 2, false); // return average of pair out of thresh with price deviation fail status
      }
    } else {
      revert('more than 3 valid sources not supported');
    }
  }
}
```

Listing 3.3: `SafeAggregatorOracle::getSafeETHPx()`

Recommendation Improve the above `getSafeETHPx()` return to properly return the average in the unlikely situation when all oracle sources are deviated from the threshold.

Status This issue has been confirmed. Since the proper `false` status is returned, we agree that this issue can be simply kept as is.

3.4 Improved Gas in TraderJoeSpell::removeLiquidityWMasterChef()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `TraderJoeSpell`
- Category: Coding Practices [6]
- CWE subcategory: CWE-561 [3]

Description

As mentioned earlier, the `Alpha HomoraV2 (AVAX)` protocol has a number of `spell` contracts that are designed to provide a consistent interface to support a variety of liquidity pools, including `Uniswap`, `Sushiswap`, `Pangolin`, and `Curve`. These `Spell` contracts inherit from the same `BasicSpell` contract with the essential functionality to interact with `HomoraBank`. (Note `HomoraBank` holds all collateral-related funds and maintains the necessary solvency of open positions.)

During our analysis with the `TraderJoeSpell` contract, we notice a key routine `removeLiquidityWMasterChef()` can be improved for gas efficiency. Specifically, it is designed to remove liquidity from `TraderJoe`. This function is well guarded with necessary validation to ensure the provided arguments are sound and consistent. Specifically, the verification of the `lp` (line 378) makes a cross-contract invocation to ensure it is expected. However, this cross-contract call can be avoided as the `lp` information can be reliably and safely returned from the previous `poolInfo()` call.

```
368   function removeLiquidityWMasterChef(
369     address tokenA,
370     address tokenB,
371     RepayAmounts calldata amt
372   ) external {
373     address lp = getAndApprovePair(tokenA, tokenB);
374     (, address collToken, uint collId, ) = bank.getCurrentPositionInfo();
375     (uint pid, ) = wmasterchef.decodeId(collId);
376     (, , , , address rewarder) = wmasterchef.chef().poolInfo(pid);
377     require(whitelistedRewarders[rewarder], 'rewarder not whitelisted');
378     require(IWMasterChefJoeV2(collToken).getUnderlyingToken(collId) == lp, 'incorrect
          underlying');
379     require(collToken == address(wmasterchef), 'collateral token & wmasterchef
          mismatched');
380 
381     // 1. Take out collateral
382     bank.takeCollateral(address(wmasterchef), collId, amt.amtLPTake);
383     wmasterchef.burn(collId, amt.amtLPTake);
384 
385     // 2-8. remove liquidity
386     removeLiquidityInternal(tokenA, tokenB, amt, lp);
387 
388     // 9. Refund joe
389     doRefund(joe);
390   }
```

Listing 3.4: TraderJoeSpell :: removeLiquidityWMasterChef()

Recommendation Avoid unnecessary gas cost when the `lp` can be reused and cached from an earlier cross-contract call.

Status This issue has been confirmed.

3.5 Suggested Revert On Impossible Situations in CurveOracle

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: CurveOracle
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.3, the protocol makes novel contributions in efficiently and reliably computing the prices of various pool tokens of Uniswap and Curve. The CurveOracle contract provides the pool token valuation on Curve-related pools.

In the following, we use the getETHPx() function from CurveOracle. As a defined interface to return the value of the given Curve pool token as ETH per unit (multiplied by 2**112), this function is permissive in allowing for tokens with a variety of decimals, even for ones larger than 18. Since all tokens supported in Curve pools are currently less than or equal to 18, we suggest to validate the decimals and consider reverting the transaction if the encountered decimal is larger than 18. A new oracle may be added later if such tokens with larger than 18 decimals are added into Curve pools for trading.

```
/// @dev Return the value of the given input as ETH per unit, multiplied by 2**112.
/// @param lp The ERC-20 LP token to check the value.
function getETHPx(address lp) external view override returns (uint) {
  address pool = poolOf[lp];
  require(pool != address(0), 'lp is not registered');
  UnderlyingToken[] memory tokens = ulTokens[lp];
  uint minPx = type(uint).max;
  uint n = tokens.length;
  for (uint idx = 0; idx < n; idx++) {
    UnderlyingToken memory ulToken = tokens[idx];
    uint tokenPx = base.getETHPx(ulToken.token);
    if (ulToken.decimals < 18) tokenPx = tokenPx / (10**(18 - uint(ulToken.decimals)))
        ;
    if (ulToken.decimals > 18) tokenPx = tokenPx * (10**(uint(ulToken.decimals) - 18))
        ;
    if (tokenPx < minPx) minPx = tokenPx;
  }
  require(minPx != type(uint).max, 'no min px');
  // use min underlying token prices
  return (minPx * ICurvePool(pool).get_virtual_price()) / 1e18;
}
```

Listing 3.5: CurveOracle::getETHPx()

Recommendation Validate the decimals and disallow currently non-existent ones.

Status This issue has been confirmed.

3.6 Meaningful Events For Important States Change

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HomoraBank`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `HomoraBank` contract as an example. This contract has a public function `accrue()` that is used to trigger interest accrual for the given bank. While examining the events that reflect the protocol dynamic, we notice there is a lack of emitting important events that reflect important state changes or abnormal protocol situation. Specifically, when the `totalDebt != debt` (line 225) is met, there is no respective event being emitted to reflect the anomaly situation.

```
216   function accrue(address token) public override {
217     Bank storage bank = banks[token];
218     require(bank.isListed, 'bank not exist');
219     uint totalDebt = bank.totalDebt;
220     uint debt = ICErc20(bank.cToken).borrowBalanceCurrent(address(this));
221     if (debt > totalDebt) {
222       uint fee = ((debt - totalDebt) * feeBps) / 10000;
223       bank.totalDebt = debt;
224       bank.reserve = bank.reserve + doBorrow(token, fee);
225     } else if (totalDebt != debt) {
226       // We should never reach here because CREAMv2 does not support *repayBorrowBehalf*
227       // functionality. We set bank.totalDebt = debt nonetheless to ensure consistency. But do
228       // note that if *repayBorrowBehalf* exists, an attacker can maliciously deflate debt
229       // share value and potentially make this contract stop working due to math overflow.
230       bank.totalDebt = debt;
231     }
232   }
```

Listing 3.6: `HomoraBank::accrue()`

Moreover, a number of `setter` functions are used to configure various protocol parameters. It is helpful to emit related events to facilitate off-chain monitoring and analytics. Example functions include the following: `setWhitelistSpells()`, `setWhitelistTokens()`, `setWhitelistUsers()`, and `setAllowContractCalls()`.

```
function setAllowContractCalls(bool ok) external onlyGov {
  allowContractCalls = ok;
}

/// @dev Set whitelist spell status
/// @param spells list of spells to change status
/// @param statuses list of statuses to change to
function setWhitelistSpells(address[] calldata spells, bool[] calldata statuses)
  external
  onlyGov
{
  require(spells.length == statuses.length, 'spells & statuses length mismatched');
  for (uint idx = 0; idx < spells.length; idx++) {
    whitelistedSpells[spells[idx]] = statuses[idx];
  }
}
```

Listing 3.7: `HomoraBank::setAllowContractCalls()`

Recommendation Properly emit an alert event to indicate a situation that should not occur, hence warranting an immediate follow-up investigation.

Status This issue has been confirmed.

3.7 Improved Validation in BasicSpell And ProxyOracle

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ProxyOracle`, `BasicSpell`
- Category: Coding Practices [6]
- CWE subcategory: CWE-561 [3]

Description

As mentioned in Section 3.3, Alpha Homora V2 for Avalanche supports a number of `Spell` contracts with inheritance from the same `BasicSpell`. To standardize the interaction with `HomoraBank`, `BasicSpell` defines the following interfaces, i.e., `doTransmit()/doTransmitETH()`, `doBorrow()/doRepay()`, `doPutCollateral()/doTakeCollateral()`, and `doRefund()/doRefundETH()`.

While examining the defined interfaces, we notice the `doTakeCollateral()` implementation can be improved. To elaborate, we show below its code snippet. The logic is rather straightforward in making a call to take collateral tokens from the bank, i.e., `HomoraBank`.

```
112 /// @dev Internal call to take collateral tokens from the bank.
113 /// @param token The token to take back.
114 /// @param amount The amount to take back.
115 function doTakeCollateral(address token, uint amount) internal {
116   if (amount > 0) {
117     if (amount == type(uint).max) {
118       (, , , amount) = bank.getCurrentPositionInfo();
119     }
120     bank.takeCollateral(address(werc20), uint(uint160(token)), amount);
121     werc20.burn(token, amount);
122   }
123 }
```

Listing 3.8: BasicSpell :: doTakeCollateral ()

When the given `amount` equals `uint(-1)`, the `doTakeCollateral()` routine queries current collateral size of the current position and then takes all back collateral tokens. Note that we can better validate the given `amount` and filter out illegitimate requests. Specifically, any amount larger than the current position's `collateralSize` can be rejected (excluding `uint(-1)` that denotes `collateralSize`).

Moreover, the `convertForLiquidation()` function of the `ProxyOracle` contract may be improved by validating both `tokenIn` and `tokenOut`. Currently, only the input `tokenOut` argument is validated in the function.

Recommendation Validate the given amount and filter out invalid requests.

Status Since the amount is also used in the following `werc20.burn(token, amount)` (line 121), any unnecessarily large amount will be blocked. The team decides to keep as is.

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Alpha HomoraV2 (AVAX)` protocol, there is a privileged `governor` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
  function setWhitelistSpells(address[] calldata spells, bool[] calldata statuses)
    external
    onlyGov
  {
    require(spells.length == statuses.length, 'spells & statuses length mismatched');
    for (uint idx = 0; idx < spells.length; idx++) {
      whitelistedSpells[spells[idx]] = statuses[idx];
    }
  }

  /// @dev Set whitelist token status
  /// @param tokens list of tokens to change status
  /// @param statuses list of statuses to change to
  function setWhitelistTokens(address[] calldata tokens, bool[] calldata statuses)
    external
    onlyGov
  {
    require(tokens.length == statuses.length, 'tokens & statuses length mismatched');
    for (uint idx = 0; idx < tokens.length; idx++) {
      if (statuses[idx]) {
        // check oracle suppport
        require(support(tokens[idx]), 'oracle not support token');
      }
      whitelistedTokens[tokens[idx]] = statuses[idx];
    }
  }
```

Listing 3.9: Example `Setters` in the `HomoraBank` Contract

Apparently, if the privileged `governor` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

```
contract TransparentUpgradeableProxyImpl is TransparentUpgradeableProxy {
  constructor(
    address _logic,
    address _admin,
    bytes memory _data
  ) payable TransparentUpgradeableProxy(_logic, _admin, _data) {}
}
```

Listing 3.10: `TransparentUpgradeableProxyImpl::constructor()`

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Alpha Homora V2 for Avalanche` protocol. The system presents a clean and consistent design that makes it distinctive and valuable when compared with current yield farming offerings. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.