SMART CONTRACT AUDIT REPORT

for

ALPHA FINANCE LAB

Prepared By: Shuxiao Wang

PeckShield

January 20, 2021

Document Properties

Client	Alpha Finance Lab
Title	Smart Contract Audit Report
Target	Alpha Homora V2
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 20, 2021	Xuxian Jiang	Final Release
1.0-rc	January 18, 2021	Xuxian Jiang	Release Candidate
0.3	January 15, 2021	Xuxian Jiang	Add More Findings #2
0.2	January 10, 2021	Xuxian Jiang	Add More Findings #1
0.1	January 5, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the **Alpha Homora V2** protocol, we in the report outline 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

`Alpha Homora` is a leveraged yield farming and leveraged liquidity providing protocol launched on Ethereum mainnet. It enables ETH lenders to earn high interest on ETH and the lending interest rate comes from leveraged yield farmers (or liquidity providers) borrowing these ETH 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. And liquidity providers can get even higher trading fees `APY` from taking on leveraged liquidity providing positions. `Alpha Homora V2` makes a number of innovations from the earlier version by supporting multi-assets lending and borrowing, multiple farming pools (e.g., `Sushiswap`, `Uniswap`, `Balancer`, `Curve`, etc), and `BYOT` (`bring your own LP tokens`).

The basic information of Alpha Homora V2 is as follows:

Table 1.1: Basic Information of Alpha Homora V2

Item	Description
Issuer	Alpha Finance Lab
Website	https://alphafinance.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 20, 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/homora-v2 (17879ae)

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

- https://github.com/AlphaFinanceLab/homora-v2 (aac0ae7)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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) [9], 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 Alpha Homora V2 implementation. 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	6	■■■■■■
Informational	2	■■
Total	10	

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, 6 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Audit Findings of Alpha Homora V2 Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Proper Allowance Cancellation in HomoraBank::setCToken()	Business Logic	Resolved
PVE-002	Low	Improved Corner Cases in HomoraMath::sqrt()	Coding Practices	Resolved
PVE-003	Low	Tighter Restriction of ensureApprove()	Security Features	Resolved
PVE-004	Informational	Improved Sanity Checks in BasicSpell::doTakeCollateral()	Coding Practices	Resolved
PVE-005	Informational	Immutable States If Only Set at Constructor()	Coding Practices	Resolved
PVE-006	Medium	Better Slippage Control/Possible DoS in SushiswapSpellV1/UniswapV2SpellV1 Repay	Time and State	Resolved
PVE-007	Low	Improved HouseHoldSpell::repayETH()	Business Logic	Resolved
PVE-008	Low	Timely poke() in HomoraBank::resolveReserve()	Time and State	Resolved
PVE-009	Low	Lack of ETH-Related Handling in CurveSpellV1()	Business Logic	Resolved
PVE-010	Medium	Proper Handling of Old Borrows in HomoraBank::setCToken()	Business Logic	Resolved

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 Allowance Cancellation in HomoraBank::setCToken()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: HomoraBank
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

The Alpha Homora V2 protocol is designed to seamlessly support CREAMv2 for lending. Accordingly, it maintains a mapping from a supported token to its cToken counterpart. This mapping can be modified through governance. For illustration, we show below the setCToken() routine that updates the cToken contract address to a new one.

```
/// @dev Upgrade cToken contract address to a new address. Must be used with care!
/// @param token The underlying token for the bank.
/// @param cToken The address of the cToken smart contract.
function setCToken(address token, address cToken) external onlyGov {
  Bank storage bank = banks[token];
  require(!cTokenInBank[cToken], 'cToken already exists');
  require(bank.isListed, 'bank not exists');
  cTokenInBank[bank.cToken] = false;
  cTokenInBank[cToken] = true;
  IERC20(bank.cToken).safeApprove(cToken, 0);
  IERC20(token).safeApprove(cToken, 0);
  IERC20(token).safeApprove(cToken, uint(-1));
  bank.cToken = cToken;
  emit SetCToken(token, cToken);
}
```

Listing 3.1: HomoraBank::setCToken()

This routine has a basic logic in firstly validating the legitimacy of the given `token` and the new `cToken` (lines 327 – 328), then canceling previous allowance on the old `cToken` (line 331), next setting up the allowance on the new `cToken` (lines 332 – 333), and finally saving the new mapping (line 334).

It comes to our attention that the cancellation of previous allowance has taken the wrong arguments. In particular, the proper cancellation should be about `token`, i.e., `IERC20(token).safeApprove(bank.cToken, 0)`, instead of current `IERC20(bank.cToken).safeApprove(cToken, 0)`.

Recommendation Properly cancel the allowance on the previous `cToken` when the mapping is updated. An example revision is shown below. It should be mentioned that the `setCToken()` routine also needs to take care of clearing the old debt balance, an issue we will elaborate on Section 3.10.

```
322 /// @dev Upgrade cToken contract address to a new address. Must be used with care!
323 /// @param token The underlying token for the bank.
324 /// @param cToken The address of the cToken smart contract.
325 function setCToken(address token, address cToken) external onlyGov {
326   Bank storage bank = banks[token];
327   require(!cTokenInBank[cToken], 'cToken already exists');
328   require(bank.isListed, 'bank not exists');
329   cTokenInBank[bank.cToken] = false;
330   cTokenInBank[cToken] = true;
331   IERC20(token).safeApprove(bank.cToken, 0);
332   IERC20(token).safeApprove(cToken, 0);
333   IERC20(token).safeApprove(cToken, uint(-1));
334   bank.cToken = cToken;
335   emit SetCToken(token, cToken);
336 }
```

Listing 3.2: HomoraBank::setCToken()

Status This issue has been fixed as the affected `setCToken()` routine has been removed in the following PR: `62`.

3.2 Improved Corner Cases in HomoraMath::sqrt()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HomoraMath`
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [3]

Description

The Alpha Homora V2 protocol has developed the fair reserve notion to properly evaluate the valuation of pool tokens (`lptoken`) of various liquidity pools, e.g., `Uniswap`, `Sushiswap`, `Balancer`, and `Curve`.

The key idea is to obtain fair prices of associated assets, next safely compute backwards from fair asset prices to fair asset reserves, and finally calculate the pool token price.

In the above computation, there is a constant need of calculating the integer square root of a given number, i.e., the familiar `sqrt()` function. The `sqrt()` function, implemented in `HomoraMath`, follows the `Babylonian` method for calculating the integer square root. Specifically, for a given x, we need to find out the largest integer z such that $z^2 <= x$.

```
20  function sqrt(uint x) internal pure returns (uint y) {
21    uint z = (x + 1) / 2;
22    y = x;
23    while (z < y) {
24      y = z;
25      z = (x / z + z) / 2;
26    }
27  }
```

Listing 3.3: HomoraMath::sqrt()

We show above current `sqrt()` implementation. The initial value of z to the iteration was given as $z = (x + 1)/2$, which results in an integer overflow when $x = uint256(-1)$. In other words, the overflow essentially sets z to zero, leading to a `division by zero` in the calculation of $z = (x/z + z)/2$ (line 25).

Note that this does not result in an incorrect return value from `sqrt()`, but does cause the function to revert unnecessarily when the above corner case occurs. Meanwhile, it is worth mentioning that if there is a `divide by zero`, the execution or the contract call will be thrown by executing the `INVALID` opcode, which by design consumes all of the gas in the initiating call. This is different from `REVERT` and has the undesirable result in causing unnecessary monetary loss.

To address this particular corner case, We suggest to change the initial value to $z = x/2 + 1$, making `sqrt()` well defined over its all possible inputs.

Recommendation Revise the above calculation to avoid the unnecessary integer overflow.

Status This issue has been fixed in the following PR (with a further optimized implementation): `63`.

3.3 Tighter Restriction of ensureApprove()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BasicSpell`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In Alpha Homora V2, there are a number of `Spell` contracts that are designed to provide a consistent interface to support a variety of liquidity pools, including `Uniswap`, `Sushiswap`, `Balancer`, 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 `BasicSpell` contract, we notice a helper routine, i.e., `ensureApprove()`. As the name indicates, it is designed to ensure that the `Spell` contract approves the given spender to spend all of its tokens. For illustration, we show below its full implementation.

```
/// @dev Ensure that the spell approve the given spender to spend all of its tokens.
/// @param token The token to approve.
/// @param spender The spender to allow spending.
/// NOTE: This is safe because spell is never built to hold fund custody.
function ensureApprove(address token, address spender) public {
  if (!approved[token][spender]) {
    IERC20(token).safeApprove(spender, uint(-1));
    approved[token][spender] = true;
  }
}
```

Listing 3.4: BasicSpell :: ensureApprove()

It comes to our attention that this routine is defined as `public`, which means any one can invoke it to add any one to be the spender. While the `Spell` contract is not holding any user funds, it is still desirable to not expose unnecessary functionalities or properly restrict the caller of `ensureApprove()`. In fact, it is feasible to define the function `private` without affecting current functionality in any way.

Recommendation Define the `ensureApprove()` as `private`, instead of current `public`.

Status With the intention of making the `ensureApprove()` function public so others can call to save users from spending gas, the team decides to keep as is.

3.4 Improved Sanity Checks in BasicSpell::doTakeCollateral()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BasicSpell`
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [3]

Description

As mentioned in Section 3.3, Alpha Homora V2 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`.

```
108   /// @dev Internal call to take collateral tokens from the bank.
109   /// @param token The token to take back.
110   /// @param amount The amount to take back.
111   function doTakeCollateral(address token, uint amount) internal {
112     if (amount > 0) {
113       if (amount == uint(-1)) {
114         (, , , amount) = bank.getPositionInfo(bank.POSITION_ID());
115       }
116       bank.takeCollateral(address(werc20), uint(token), amount);
117       werc20.burn(token, amount);
118     }
119   }
```

Listing 3.5: 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`).

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 117), any unnecessarily large amount will be blocked. The team decides to keep as is.

3.5 Immutable States If Only Set at Constructor()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [3]

Description

Since version 0.6.5, Solidity introduces the feature of declaring a state as immutable. An immutable state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as immutable is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an immutable state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of immutable states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show the key state variables defined in SushiswapSpellV1. If there is no need to dynamically update these key state variables, e.g., factory and router, they can be declared as immutable for gas efficiency.

```
contract SushiswapSpellV1 is BasicSpell {
  using SafeMath for uint;
  using HomoraMath for uint;

  IUniswapV2Factory public factory;
  IUniswapV2Router02 public router;

    ...
}
```

Listing 3.6: SushiswapSpellV1.sol

Similarly, we can define the states factory and router in UniswapV2SpellV1 as immutable too.

Recommendation Revisit the state variable definition and make good use of immutable/constant states.

Status This issue has been fixed in the following PR: 65.

3.6 Better Slippage Control/Possible DoS in SushiswapSpellV1/UniswapV2SpellV1 Repay

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [2]

Description

As a leveraged yield farming and leveraged liquidity providing protocol, Alpha Homora V2 allows users to borrow from the integrated `CREAMv2` platform. The borrow position requires later repayment before the user can take back the collateral. During our analysis on the repayment logic, we notice the built-in slippage control can be improved.

For illustration, we show below the `removeLiquidityInternal()` routine from the `SushiswapSpellV1` contract. This routine is tasked to remove liquidity from the supported `Sushiswap` pool. In order to minimize the trade to meet the repayment requirement, it has an internal optimization logic (step 5 in lines 260 – 268) to convert one token to another (via `swapTokensForExactTokens()`).

```
229   function removeLiquidityInternal(
230     address tokenA,
231     address tokenB,
232     RepayAmounts calldata amt
233   ) internal {
234     address lp = getPair(tokenA, tokenB);
235     uint positionId = bank.POSITION_ID();

237     uint amtARepay = amt.amtARepay;
238     uint amtBRepay = amt.amtBRepay;
239     uint amtLPRepay = amt.amtLPRepay;

241     // 2. Compute repay amount if MAX_INT is supplied (max debt)
242     if (amtARepay == uint(-1)) {
243       amtARepay = bank.borrowBalanceCurrent(positionId, tokenA);
244     }
245     if (amtBRepay == uint(-1)) {
246       amtBRepay = bank.borrowBalanceCurrent(positionId, tokenB);
247     }
248     if (amtLPRepay == uint(-1)) {
249       amtLPRepay = bank.borrowBalanceCurrent(positionId, lp);
250     }

252     // 3. Compute amount to actually remove
253     uint amtLPToRemove = IERC20(lp).balanceOf(address(this)).sub(amt.amtLPWithdraw);
```

```
255     // 4. Remove liquidity
256     (uint amtA, uint amtB) =
257       router.removeLiquidity(tokenA, tokenB, amtLPToRemove, 0, 0, address(this), now);

259     // 5. MinimizeTrading to repay debt
260     if (amtA < amtARepay && amtB >= amtBRepay) {
261       address[] memory path = new address[](2);
262       (path[0], path[1]) = (tokenB, tokenA);
263       router.swapTokensForExactTokens(amtARepay.sub(amtA), uint(-1), path, address(this), now);
264     } else if (amtA >= amtARepay && amtB < amtBRepay) {
265       address[] memory path = new address[](2);
266       (path[0], path[1]) = (tokenA, tokenB);
267       router.swapTokensForExactTokens(amtBRepay.sub(amtB), uint(-1), path, address(this), now);
268     }

270     // 6. Repay
271     doRepay(tokenA, amtARepay);
272     doRepay(tokenB, amtBRepay);
273     doRepay(lp, amtLPRepay);

275     // 7. Slippage control
276     require(IERC20(tokenA).balanceOf(address(this)) >= amt.amtAMin);
277     require(IERC20(tokenB).balanceOf(address(this)) >= amt.amtBMin);
278     require(IERC20(lp).balanceOf(address(this)) >= amt.amtLPWithdraw);

280     // 8. Refund leftover
281     doRefundETH();
282     doRefund(tokenA);
283     doRefund(tokenB);
284     doRefund(lp);
285   }
```

Listing 3.7: SushiswapSpellV1:: removeLiquidityInternal ()

Note that it operates on the AMM-backed pool and naturally leads to slippage. Further, it is possible to be externally influenced (e.g., by sandwiched attacks). Note that the internal optimization logic to minimize the trade incorrectly computes the arguments to `swapTokensForExactTokens()`. Specifically, the conditional check should not validate against `amtA < amtARepay && amtB >= amtBRepay` (line 260) and `amtA >= amtARepay && amtB < amtBRepay` (line 264). Instead the comparison should be `amtA < amtADesired && amtB >= amtBDesired` (line 260) and `amtA >= amtADesired && amtB < amtBDesired` (line 264). And accordingly, the intended token amount for conversion should be `amtADesired.sub(amtA)` or `amtBDesired.sub(amtB)`, instead of current `amtARepay.sub(amtA)` (line 263) or `amtBRepay.sub(amtB)` (line 268).

Also that the external influence could exploit the built-in slippage control to foil legitimate repayment. A similar issue also exists in adding liquidity to the pool. We need to emphasize that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user. As a mitigation, Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of liquidity providers.

Status This issue has been fixed in the following PR: `60`.

3.7 Improved HouseHoldSpell::repayETH()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `HouseHoldSpell`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

Among the set of `Spell` contracts, `HouseHoldSpell` is an interesting one with minimal implementation (see the code snippet below). However, it contains a full implementation that conforms to the standard API interfaces to interact with `HomoraBank`, i.e., `doTransmit()/doTransmitETH()`, `doBorrow()/doRepay()`, and `doPutCollateral()/doTakeCollateral()`.

```
contract HouseHoldSpell is BasicSpell {
  constructor(
    IBank _bank,
    address _werc20,
    address _weth
  ) public BasicSpell(_bank, _werc20, _weth) {}

  function borrowETH(uint amount) external {
    doBorrow(weth, amount);
    doRefundETH();
  }

  function borrow(address token, uint amount) external {
    doBorrow(token, amount);
    doRefund(token);
  }

  function repayETH(uint amount) external payable {
    doTransmitETH();
```

```
    doRepay(weth, amount);
  }

  function repay(address token, uint amount) external {
    doTransmit(token, amount);
    doRepay(token, IERC20(token).balanceOf(address(this)));
  }

  function putCollateral(address token, uint amount) external {
    doTransmit(token, amount);
    doPutCollateral(token, IERC20(token).balanceOf(address(this)));
  }

  function takeCollateral(address token, uint amount) external {
    doTakeCollateral(token, amount);
    doRefund(token);
  }
}
```

Listing 3.8: HouseHoldSpell

It comes to our attention that the logic of `repayETH()` can be improved when the given amount is less than the transferred `msg.value`. In this case, the remaining `ETH`, i.e., `msg.value - amount`, will be left on the contract. A better solution will be to refund the remaining amount, if any, back to the user.

Recommendation Revise the `repayETH()` logic to refund remaining `ETH` if any.

```
  function repayETH(uint amount) external payable {
    doTransmitETH();
    doRepay(weth, amount);
  }
}
```

Listing 3.9: HouseHoldSpell::repayETH()

Status This issue has been fixed in the following PR: `66`.

3.8 Timely poke() in HomoraBank::resolveReserve()

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: HomoraBank
- Category: Time and State [6]
- CWE subcategory: CWE-362 [2]

Description

In Alpha Homora V2, the HomoraBank contract is designed to be the main entry for interaction with users. In particular, one entry routine, i.e., execute(), takes user calls and dispatches to the designated caster, which further invokes specified Spell contracts. This approach is flexible to accommodate dynamic additions of new Spell contracts and other functionalities.

In the following, we examine the borrow() operation that allows farming users to take a leveraged position in borrowing funds from the integrated CREAMv2. It emphasizes in its doBorrow() routine the need of ensuring that cToken interest should be accrued up to this block before calling doBorrow().

```
/// @dev Borrow tokens from that bank. Must only be called while under execution.
/// @param token The token to borrow from the bank.
/// @param amount The amount of tokens to borrow.
function borrow(address token, uint amount) external override inExec poke(token) {
  Bank storage bank = banks[token];
  require(bank.isListed, 'bank not exists');
  Position storage pos = positions[POSITION_ID];
  uint totalShare = bank.totalShare;
  uint totalDebt = bank.totalDebt;
  uint share = totalShare == 0 ? amount : amount.mul(totalShare).div(totalDebt);
  bank.totalShare = bank.totalShare.add(share);
  uint newShare = pos.debtShareOf[token].add(share);
  pos.debtShareOf[token] = newShare;
  if (newShare > 0) {
    pos.debtMap |= (1 << uint(bank.index));
  }
  IERC20(token).safeTransfer(msg.sender, doBorrow(token, amount));
  emit Borrow(POSITION_ID, msg.sender, token, amount, share);
}
```

Listing 3.10: HomoraBank::borrow()

```
/// @dev Internal function to perform borrow from the bank and return the amount
    received.
/// @param token The token to perform borrow action.
/// @param amountCall The amount use in the transferFrom call.
/// NOTE: Caller must ensure that cToken interest was already accrued up to this block
    .
function doBorrow(address token, uint amountCall) internal returns (uint) {
```

```
    Bank storage bank = banks[token]; // assume the input is already sanity checked.
    uint balanceBefore = IERC20(token).balanceOf(address(this));
    require(ICErc20(bank.cToken).borrow(amountCall) == 0, 'bad borrow');
    uint balanceAfter = IERC20(token).balanceOf(address(this));
    bank.totalDebt = bank.totalDebt.add(amountCall);
    return balanceAfter.sub(balanceBefore);
  }
```

Listing 3.11: HomoraBank::doBorrow()

This is necessary as if the cToken interest is not accrued to the current block, the bank's debt will simply increase without HomoraBank knowing it. The may result in a slightly higher debt share (but not much) for previous borrowers.

Meanwhile, we notice the presence of another routine resolveReserve() that is used to resolve pendingReserve. This routine calls doBorrow(), but without accruing the cToken interest to the current block!

```
  /// @dev Trigger reserve resolve by borrowing the pending amount for reserve.
  /// @param token The underlying token to trigger reserve resolve.
  function resolveReserve(address token) public lock {
    Bank storage bank = banks[token];
    require(bank.isListed, 'bank not exists');
    uint pendingReserve = bank.pendingReserve;
    bank.pendingReserve = 0;
    bank.reserve = bank.reserve.add(doBorrow(token, pendingReserve));
  }
```

Listing 3.12: HomoraBank::removeLiquidityInternal()

Recommendation Revise the resolveReserve() routine by adding the poke() modifier. An example revision is shown below:

```
  /// @dev Trigger reserve resolve by borrowing the pending amount for reserve.
  /// @param token The underlying token to trigger reserve resolve.
  function resolveReserve(address token) public lock poke(token) {
    Bank storage bank = banks[token];
    require(bank.isListed, 'bank not exists');
    uint pendingReserve = bank.pendingReserve;
    bank.pendingReserve = 0;
    bank.reserve = bank.reserve.add(doBorrow(token, pendingReserve));
  }
```

Listing 3.13: Revised HomoraBank::removeLiquidityInternal()

Status This issue has been fixed in the following PR: 67.

3.9 Lack of ETH-Related Handling in CurveSpellV1

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CurveSpellV1`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

In Section 3.7, we have examined a specific `HouseHoldSpell` contract. In this section, we examine another `Spell` contract, i.e., `CurveSpellV1`. This `CurveSpellV1` contract aims to seamlessly support farming with `Curve` pool tokens. Currently, there are more than 20 `Curve` pools that provide decent yields from collected trading fees.

In the following, we show a specific `addLiquidity2()` routine that supports the liquidity addition for pools with two underlying tokens. Note this routine is marked as `payable`, indicating the acceptance of `ETH`. However, the internal logic does not transfer `ETH` to the corresponding `Curve` pool. There is also no call to convert `ETH` into `WETH`. As a result, the current implementation is unable to support `ETH`-related pools. Note that there are at least two `ETH`-related `Curve` pools: `seth` and `steth`.

```
/// @dev add liquidity for pools with 2 underlying tokens
function addLiquidity2(
  address lp,
  uint[2] calldata amtsUser,
  uint amtLPUser,
  uint[2] calldata amtsBorrow,
  uint amtLPBorrow,
  uint minLPMint,
  uint pid,
  uint gid
) external payable {
  address pool = getPool(lp);
  require(ulTokens[lp].length == 2, 'incorrect pool length');
  require(wgauge.getUnderlyingToken(wgauge.encodeId(pid, gid, 0)) == lp, 'incorrect
      underlying');
  address[] memory tokens = ulTokens[lp];

  // 0. Take out collateral
  uint positionId = bank.POSITION_ID();
  (, , uint collId, uint collSize) = bank.getPositionInfo(positionId);
  if (collSize > 0) {
    (uint decodedPid, uint decodedGid, ) = wgauge.decodeId(collId);
    require(decodedPid == pid && decodedGid == gid, 'incorrect coll id');
    bank.takeCollateral(address(wgauge), collId, collSize);
    wgauge.burn(collId, collSize);
  }
```

```

    // 1. Ensure approve 2 underlying tokens
    ensureApproveN(lp, 2);

    // 2. Get user input amounts
    for (uint i = 0; i < 2; i++) doTransmit(tokens[i], amtsUser[i]);
    doTransmit(lp, amtLPUser);

    // 3. Borrow specified amounts
    for (uint i = 0; i < 2; i++) doBorrow(tokens[i], amtsBorrow[i]);
    doBorrow(lp, amtLPBorrow);

    // 4. add liquidity
    uint[2] memory suppliedAmts;
    for (uint i = 0; i < 2; i++) {
      suppliedAmts[i] = IERC20(tokens[i]).balanceOf(address(this));
    }
    ICurvePool(pool).add_liquidity(suppliedAmts, minLPMint);

    // 5. Put collateral
    uint amount = IERC20(lp).balanceOf(address(this));
    ensureApprove(lp, address(wgauge));
    uint id = wgauge.mint(pid, gid, amount);
    bank.putCollateral(address(wgauge), id, amount);

    // 6. Refund
    for (uint i = 0; i < 2; i++) doRefund(tokens[i]);

    // 7. Refund crv
    doRefund(crv);
  }
```

Listing 3.14: CurveSpellV1::addLiquidity2()

In addition, the corresponding `removeLiquidity()` counterparts do not need to be `payable`.

Recommendation Revise the above liquidity addition and removal logic to reflect the intended purpose.

Status This issue has been fixed in the following PR: 69.

3.10 Proper Handling of Old Borrows in HomoraBank::setCToken()

- ID: PVE-010
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `HomoraBank`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

In Section 3.1, we study the `setCToken()` routine and report an issue in canceling previous spending allowance. In this section, we focus on the same routine and examine possible implications from this routine.

To elaborate, we shown below the routine's implementation. This routine allows for dynamic upgrade of a `cToken` contract address to a new one. Note `cTokens` are a back-end unit of account for the `Compound/CREAMv2` protocol: When a user supplies cryptocurrency to the protocol, `cTokens` are used to keep track of the funds that they have lent, as well as any interest earned.

```
/// @dev Upgrade cToken contract address to a new address. Must be used with care!
/// @param token The underlying token for the bank.
/// @param cToken The address of the cToken smart contract.
function setCToken(address token, address cToken) external onlyGov {
  Bank storage bank = banks[token];
  require(!cTokenInBank[cToken], 'cToken already exists');
  require(bank.isListed, 'bank not exists');
  cTokenInBank[bank.cToken] = false;
  cTokenInBank[cToken] = true;
  IERC20(bank.cToken).safeApprove(cToken, 0);
  IERC20(token).safeApprove(cToken, 0);
  IERC20(token).safeApprove(cToken, uint(-1));
  bank.cToken = cToken;
  emit SetCToken(token, cToken);
}
```

Listing 3.15: HomoraBank::setCToken()

```
struct Bank {
  bool isListed; // Whether this market exists.
  uint8 index; // Reverse look up index for this bank.
  address cToken; // The CToken to draw liquidity from.
  uint reserve; // The reserve portion allocated to Homora protocol.
  uint pendingReserve; // The pending reserve portion waiting to be resolve.
  uint totalDebt; // The last recorded total debt since last action.
  uint totalShare; // The total debt share count across all open positions.
```

```
}
```

Listing 3.16: The Bank Structure

When the `cToken` mapping is changed, the purpose is to redirect the drawing of liquidity from another pool. However, the associated meta-data or states, especially `totalDebt`, `reserve`, and `pendingReserve`, are not properly updated. With that, if a malicious actor simply calls `accrue()`, the current `totalDebt` is reset to 0! This may potentially make this contract stop working as `totalDebt` is used in both `borrow()` and `repay()` operations. Its denominator role leads to `divide-by-zero` error, reverting these `borrow()` and `repay()` operations.

```
/// @dev Trigger interest accrual for the given bank.
/// @param token The underlying token to trigger the interest accrual.
function accrue(address token) public override {
  Bank storage bank = banks[token];
  require(bank.isListed, 'bank not exists');
  uint totalDebt = bank.totalDebt;
  uint debt = ICErc20(bank.cToken).borrowBalanceCurrent(address(this));
  if (debt > totalDebt) {
    uint fee = debt.sub(totalDebt).mul(feeBps).div(10000);
    bank.totalDebt = debt;
    bank.pendingReserve = bank.pendingReserve.add(fee);
  } else if (totalDebt != debt) {
    // We should never reach here because CREAMv2 does not support *repayBorrowBehalf*
    // functionality. We set bank.totalDebt = debt nonetheless to ensure consistency. But do
    // note that if *repayBorrowBehalf* exists, an attacker can maliciously deflate debt
    // share value and potentially make this contract stop working due to math overflow.
    bank.totalDebt = debt;
  }
}
```

Listing 3.17: HomoraBank::accrue()

Recommendation Properly handle previous borrows when calling `setCToken` to update new `cToken`.

Status This issue has been fixed as the affected `setCToken()` routine has been removed in the following PR: 62.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the Alpha Homora V2 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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