Alpha Finance Homora V2 Audit

MAY 13, 2021 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

The Alpha Finance Lab team asked us to review and audit their Homora V2 smart contracts. We looked at the code and now publish our results.

Scope

We audited commit `5efa332f2ecf8e9705c326cffda5305bc6f752f7` of the AlphaFinanceLab/homora-v2 repository. Only the following files inside the `/contracts` folder were in scope:

```
├── Governable.sol
├── HomoraBank.sol
├── oracle
│   ├── AggregatorOracle.sol
│   ├── BalancerPairOracle.sol
│   ├── CoreOracle.sol
│   ├── CurveOracle.sol
│   ├── ProxyOracle.sol
│   ├── UniswapV2Oracle.sol
│   └── UsingBaseOracle.sol
├── spell
│   ├── BalancerSpellV1.sol
│   ├── BasicSpell.sol
│   ├── CurveSpellV1.sol
│   ├── SushiswapSpellV1.sol
│   ├── UniswapV2SpellV1.sol
│   └── WhitelistSpell.sol
└── wrapper
├── WERC20.sol
├── WLiquidityGauge.sol
├── WMasterChef.sol
└── WStakingRewards.sol
```

Overall Health

Overall, we found the project's codebase to be very readable, well organized, and sufficiently commented. The team was highly responsive and open to feedback and discussion. They have many publicly accessible docs for their code and protocol in general.

System Overview

Homora V2 allows users to leverage their assets to provide liquidity to several external protocols. Homora V2 allows for compounding leverage, where borrowed assets can be used as additional collateral to further increase leverage. It achieves this by keeping all leveraged assets protocol-controlled while still allowing users to provide liquidity to a variety of protocols.

Privileged Roles

At deployment, the protocol sets addresses for a `governor` for the various contracts in the system. The `governor` has the power to initiate a transfer of the `governor` role to a new address. Initially, the `governor` will be the Alpha Finance Labs team, but they have expressed their intent to transfer this role to the community in the future.

A number of actions in the protocol are callable only by the `governor`. These include:

- Controlling the various whitelists for tokens, users, and spells that the protocol depends on
- Setting the oracle addresses throughout the protocol
- Setting the fee `HomoraBank` charges users
- Resolving the `pendingReserve`, to finalize outstanding user fees within the system. See issue H01 for more information

about why this action is critical.

- Setting the protocol switch to allow calls from other contracts or only from Externally Owned Accounts
- Withdrawing the reserve funds from the bank (this includes the collected fees)
- Registering valid liquidity gauges for interactions with the Curve protocol

External dependencies and trust assumptions

The protocol relies on CREAM to fund loans for Homora V2 users. It also deposits user tokens to Uniswap, SushiSwap, Balancer, and Curve. We assumed that all of these external protocols work as intended.

During this audit, we also assumed that the `governor` and oracle price feeds are available, honest, and not compromised.

Findings

Here we present our findings.

Critical severity

None.

High severity

[H01] Users can force other users to pay their fees

When users `borrow` tokens using `HomoraBank`, the bank performs a borrow from `CREAM`. As `HomoraBank` takes on the debt from `CREAM`, it must account internally for each user's proportionate share of that debt. As `CREAM` adds interest to the borrowed amount, the `totalDebt` for the corresponding token increases, which in turn increases the amount owed by each `HomoraBank` user who has debt in that token.

In addition to paying interest, users also pay a fee to the governor of `HomoraBank`. This fee is a percentage of the interest earned by `CREAM`. However, while the `CREAM` interest is accrued automatically on many calls to the bank – so that the interest earned is always up to date at the time of a withdrawal – the same is not true of the fee.

The fee is calculated every time interest is accrued, in the `accrue` function. The fee amount is then merely recorded as a `pendingReserve` in the bank struct; it is not yet added to the `totalDebt` owed by users (which is required to effectively "charge" the fee). Only when the governor calls the function `resolveReserve`, does the `pendingReserve` "resolve" and the fee actually get added to the `totalDebt`.

This means that if a user repays their debt before the governor next calls `resolveReserve`, they will avoid paying fees on their debt. The economic incentive is then for users to front-run the governor's calls to `resolveReserve`. Importantly, although a user could avoid paying fees in this manner, the fees would still be accrued to the bank, and they would instead be charged to the other users who have debt associated with the same token.

In extreme cases, especially if `resolveReserve` were not called regularly by governance, malicious users could pass off huge amounts of fees to other users, which could lead to those other users getting liquidated as a result.

To avoid these issues and to align the code with the intent of the system, consider incrementing `totalDebt` every time fees are accrued to ensure that all fees are charged to users fairly according to their share of borrows.

Update: *Fixed in PR#94.*

Medium severity

[M01] Math not rounding in protocol favor

Throughout `HomoraBank`, the amount of debt owed by a position is calculated proportionally to the number of debt shares that position holds. The exact calculation performed is `debtOwed = totalDebt * positionsShare / totalShare`. However, this calculation truncates the debt owed in the user's favor – rounding down even when `x.99999` is owed.

This means that at any given time, a significant amount of debt can be unaccounted for. Say `n` users each owe `100.9` to the protocol, the sum of their `debtOwed` calculated with truncation is `0.9n` less than the `totalDebt` of the protocol. In an extreme situation, this could lead the protocol to become `0.9n` undercollateralized without detection. If users start paying off this truncated debt, then the extra amount of debt is passed on to other users.

In some situations, a truncation in the user's favor can have catastrophic consequences for a protocol. While we have not found that to be true of this situation, consider always rounding all divisions in the protocol's favor to help mitigate the risk of users abusing their advantage in the system.

Update: *Fixed in PR#95.*

[M02] Governor can prevent user repaying debt

When borrowing from the bank, the protocol requires that the token being borrowed is whitelisted, to ensure that only approved tokens can be used within the protocol. However, when repaying the bank, the same is required. This means that if governance concludes that a previously-whitelisted token is no longer safe, and they want to prevent users from borrowing that token, then they would have no option but to cause users with outstanding debt in that token to be unable to repay that debt.

The same issue is also true of spells that inherit from `WhitelistSpell` and require that a liquidity provider (LP) token is whitelisted both when depositing and withdrawing it. This is the case in `UniswapV2SpellV1`, `SushiswapSpellV1`, `CurveSpellV1`, and `BalancerSpellV1`.

Consider removing the requirement that repayments require a whitelisted token to enable users to repay debt on previously-whitelisted tokens, or to withdraw previously deposited LP tokens. Alternatively, if more granular control over all operations is preferred, then consider modifying the whitelist implementation to support permissions for the different processes in the system.

Update: *Partially fixed in PR#96. Alpha implemented a mechanism in the HomoraBank that allows them to pause borrows and repayments separately. However this fix is not on a per-token basis, and so if the team wants to remove just 1 token from the system, they will still be forced to either lock up all user funds in that token, or to pause borrows for all tokens. Additionally no fixes were made for the whitelisting in spells.*

Low severity

[L01] Spells make assumptions about underlying collateral

Throughout the spell contracts – namely `UniswapV2SpellV1`, `SushiswapSpellV1`, `BalancerSpellV1`, and `CurveSpellV1` – assumptions are made about a position's collateral token, and the spell's intended collateral token. We now consider the example of the `addLiquidityWStakingRewards` function of the `UniswapV2SpellV1`, however the same issue exists in different forms within all spells listed above.

The function fetches the position's underlying collateral information, requires that the collateral token wraps the desired liquidity provider (LP) token, and then assumes from then onwards that the collateral token *is* the `wstaking` token. Given

that `werc20`-wrapped LP tokens are a form of collateral within the protocol, the global `werc20` token would also pass the require statement despite not being the `wstaking` token.

While assuming that the collateral token is `wstaking` does not cause any vulnerabilities, it does lead to the call reverting. Unwrapping the actual collateral token instead would enable users to unwrap LP tokens from `werc20` or a different `wstaking` contract, and deposit again with rewards using the current `wstaking` contract.

If the intention with the require statement is to ensure that the existing collateral token *is* `wstaking`, then consider adding a more explicit `require` statement to check for this. Otherwise, consider unwrapping the position's actual collateral token to enable for greater functionality within the spell.

Update: *Fixed in PR#97.*

[L02] Constants not explicitly declared

Throughout the codebase there are many occurrences of literal values being used with unexplained meaning. This makes areas of the code more difficult to understand. Some examples include:

- The 'maximum integer' `uint(-1)` used throughout the codebase.
- The acceptable range of `1e18` to `1.5e18` used in `AggregatorOracle`.
- Many literal values used throughout `_optimalDepositA` to calculate optimal uniswap values.
- The literal `2**112` used throughout the contracts.
- The literal `10000` used throughout `ProxyOracle`.

Literal values in the codebase without an explanation as to their meaning make the code harder to read, understand, and maintain for developers, auditors and external contributors alike. Consider defining a `constant` variable for every literal value used and giving it a clear and self-explanatory name. Additionally, for complex values, inline comments explaining how they were calculated or why they were chosen are highly recommended.

Update: *Partially fixed in PR#98. Alpha added constant variables to address the second point above, but have decided to leave the others as they are.*

[L03] Events missing or not well indexed

There are instances in the codebase where storage is modified, but events are not emitted. For instance, all of the functions that modify the governor and pending governor in the `Governable` contract.

Where there are events, many of their definitions are lacking indexed parameters. Some examples include:

- The events in `ProxyOracle`
- The `SetRoute` event in `CoreOracle`

Because it may be of particular interests to outside observers, consider indexing the `token` argument in events, as is done in the `AggregatorOracle` contract.

As event emissions and properly indexed parameters assist off-chain observers watch, search, and filter on-chain activity, consider emitting events whenever on-chain storage modifications occur and consider ensuring that all events in the codebase have indexed parameters.

Update: *Fixed in PR#99.*

[L04] Before initialization, `EXECUTOR` returns the wrong value

Within the `HomoraBank` contract, the function `EXECUTOR` returns the address of the owner of the position currently being executed. If no position is currently being executed, the function reverts and does not return a value. However, before the contract is initialized, the check for whether a position is currently executing does not work, and the function incorrectly returns the `0` address instead of reverting.

Consider setting the initial value of `POSITION_ID` at the top of the contract, instead of in the `initialize` function, so that the check succeeds even before the contract is initialized.

Update: *Acknowledged.*

[L05] Incorrect interface definition

In the `CurveOracle` and `CurveSpellV1` contracts, external calls are made to Curve's on-chain registry. The registry function `get_n_coins` is used to obtain the number of underlying tokens, which various functions use to properly iterate relevant token arrays. However, the interface being used deviates from the implementation. Specifically, the interface has the return value of this call as a single `uint`, when in reality it is an array consisting of two `uint` values – the first representing the number of "wrapped tokens" and the second representing "*all* underlying tokens". In the case of metapools, these values may be different.

Consider updating the interface to accurately reflect the implementation in order to avoid potential confusion, avoid unexpected behavior, and clarify the intention of the code.

Update: *Fixed in PR#100.*

[L06] Not handling `approve` return value

As defined in the ERC20 Specification, the `approve` function returns a `bool` that signals the success of the call. However, throughout the codebase, the value returned from calls to `approve` is ignored. Examples of this are:

- Line 29 of `WStakingRewards`
- Line 63 of `WMasterChef`

In other places in the codebase, calls to `approve` are within a `require` statement which does handle the boolean return. Examples of this are:

- Lines 90-92 of `IbETHRouterV2` (this contract is out of scope).

To handle calls to `approve` safely, even when interacting with ERC20 implementations that, incorrectly, *do not* return a boolean, consider using the `safeApprove` function in OpenZeppelin's SafeERC20 contract for all approvals.

Update: *Fixed in PR#101.*

[L07] Misleading comments

The following comments are potentially misleading or errant:

- Throughout `AggregatorOracle.sol` there are comments that say `maxPriceDeviation` is measured in basis points. However, it is not. It is a number between `1e18` and `1.5e18`, where `1e18` is 0% price deviation accepted, and `1.5e18` is 50% price deviation accepted.
- Lines 127 and 128 of `ProxyOracle.sol` should each say `ERC20` not `ERC1155`.

Consider correcting these comments to improve the readability of the codebase and reduce unnecessary confusion.

Update: *Fixed in PR#102.*

[L08] Protocol does not fully support `ERC20` tokens with fees

`WERC20` is a wrapper for `ERC20` tokens. The `mint` function transfers in underlying `ERC20` tokens from a user and, in exchange, mints them some corresponding amount of `ERC1155` tokens. Because an `ERC20` may reduce the quantity of tokens transferred (by subtracting transfer fees, for instance) the quantity of tokens that are transferred are not necessarily the quantity of tokens that end up in a recipient's balance. The `mint` function accounts for this behavior, and only mints the quantity of `ERC1155` tokens corresponding to the actual balance increase of `ERC20` tokens.

This would suggest the protocol is prepared to support `ERC20` tokens with transfer fess. However, contracts that call `mint`, such as the `BasicSpell` contract function `doPutCollateral`, do not account for the potential decrease and will revert when the transferred amount and amount received are not identical.

The same sort of partial support is present in the `HomoraBank` contract, where the `doBorrow` function, for instance, even returns the result of such fee-aware computations. This return value is merely discarded however, when `BasicSpell.doBorrow` leads to a call of `HomoraBank.doBorrow`, but does not itself note the actual tokens sent – again resulting in potential reversions.

To clarify intent and to avoid unnecessary confusion caused by the partially implemented support for such tokens, consider adding inline documentation to explain where support is intentionally lacking. Alternatively, if the desire is to fully support such tokens, then consider further building out that support.

Update: *Fixed in PR#103. BalancerSpellV1 was updated to correctly handle ERC20s with fees. Additionally comments were added to `doBorrow` and `doTransmit` to explain that the `amount` parameter should not be used as the received amount. No changes were made to `doPutCollateral` and `doTakeCollateral`, Alpha stated: "The collateral tokens are LP tokens, which shouldn't have fees on transfer (and we won't support ones that have fees)."*

[L09] `addLiquidityWStakingRewards` doesn't verify underlying token

The `addLiquidityWStakingRewards` function of the `UniswapV2SpellV1` and `BalancerSpellV1` allows the user to provide the address of the `wstaking` contract they would like to use to wrap their liquidity provider (LP) tokens. However the function never checks that the `underlyingToken` of the `wstaking` contract provided is the appropriate LP token.

Fortunately, as the code exists now, it does ultimately revert when the function attempts to mint `wstaking` tokens without having the correct LP tokens to do so. However, this intent is not made clear, and a small change to the code could lead to an incorrect `wstaking` contract being used.

Consider adding a check that the `wstaking` contract provided by the user is for the correct LP token.

Update: *Partially fixed in PR#97. Checks were added to spells to ensure that the collateral and wstaking contracts are the same. However no checks are made to ensure that the `wstaking` provided to `addLiquidityWStakingRewards` has the correct LP token.*

[L10] The `pairs` mapping in `BalancerSpellV1` is permanently empty

Within the `BalancerSpellV1` contract, a mapping called `pairs` is defined, which maps from Balancer Pool addresses to the two underlying token addresses of that Balancer Pool. Within the function `getPair`, the underlying tokens for a given Pool are looked up in the mapping, and if the entry is empty, then the underlying tokens are fetched from the Pool itself. However, throughout the contract, nothing is ever added to the `pairs` mapping, meaning that it is guaranteed to be empty at all times.

Consider updating the underlying tokens for a Pool in the mapping after fetching them, so that each Pool is only queried

once. This will reduce gas costs for users and clarify the intention of the code.

Update: *Fixed in PR#104.*

[L11] Number of oracle sources can exceed reversion threshold

The `AggregatorOracle` facilitates the retrieval of token prices in terms of ETH from multiple sources, which it then aggregates. It then either reverts, or returns the mean or median retrieved price depending on some thresholds for deviation and the number of sources. If there are ever more than three sources for a given token, then the price retrieval function, `getETHPx`, will revert. This effectively sets an upper bound on the number of sources per token.

However, the governor can currently add an unlimited number of sources, because the functions for adding sources for tokens do not take the effective upper bound of three into consideration. Given that it would be detrimental to have in excess of three sources per token, consider limiting the number of sources that can be added.

Update: *Fixed in PR#105.*

[L12] `WMasterChef` can leave Sushi inaccessible

The `WMasterChef` function `emergencyBurn` burns some amount of the caller's `WMasterChef` tokens and then withdraws the corresponding amount of liquidity provider (LP) tokens from SushiSwap's `MasterChef` contract. The withdrawn LP tokens are then transferred to the caller.

This `withdraw` from Sushiswap's `MasterChef` also transfers SUSHI tokens to the `WMasterChef` contract – but they are not forwarded from there to the caller. Instead, `emergencyBurn` merely neglects to handle the SUSHI rewards. During and after this call to `emergencyBurn`, there exists no mechanism to withdraw any SUSHI tokens released as part of this call.

Fortunately, `emergencyBurn` is never actually called by any spell in the protocol. However given its potential to lock Sushi tokens with no recourse, consider removing the function altogether. Alternatively, consider adding functionality to deal with any SUSHI that would otherwise remain inaccessible because of this function's current implementation.

Update: *Fixed in PR#106.*

[L13] Upgradeable contract parents not reserving space

The `HomoraBank` contract is intended to be upgradeable via a proxy pattern. However, its parent contracts do not have any space reserved in case they need additional storage variables added at upgrade. While `HomoraBank` itself can have additional storage slots appended in case of an upgrade, its parent contracts currently offer no such flexibility.

To accommodate this possibility, it is common to have a fixed-length array of full 32-bit words in each parent contract. If a parent contract needs to have storage appended, the length of the array is decremented to make room for any newly-added storage slots. Such a layout helps to accommodate unforeseen future upgrades to parent contracts without compromising the storage layout or data integrity of the child contract or the proxy using the child contract's logic.

Consider reserving some storage in the parent contracts of any upgradeable contracts, and using community vetted tooling such as OpenZeppelin's Upgrade Plugins to assist with facilitating upgrades where possible.

Update: *Partially fixed in PR#107. Space was reserved in 2 parent contracts, but not all parent contracts.*

[L14] Sprawling whitelist architecture

The protocol relies extensively on whitelisting:

- The `HomoraBank` contract maintains three separate whitelist mappings; one for tokens, one for spells, and another for

users.

- The spells, upon which the system relies to interact with external protocols, inherit from `WhitelistSpell`, such that each spell maintains its own whitelist of supported liquidity pool tokens.
- The `ProxyOracle`, which the protocol places between itself and the ultimate sources of token price data, also maintains a whitelist of supported collateral tokens. It also maintains a mapping of oracle data, which acts as another quasi-whitelist for tokens.

With such a sprawling whitelist architecture, there is no single point of reference to check for protocol support of any given token. For instance, the `HomoraBank` whitelist could signal support for a token, but if the `ProxyOracle` does not *also* have support for it, then the token is not actually usable by the protocol. There is no logic to facilitate keeping these whitelists in sync with each other. Consequently, keeping the disparate whitelists in sync currently falls on the governor as a manual chore.

Consider adding logic to unify the whitelists where possible. For example, for `HomoraBank` to whitelist a token, perhaps it should check that the token is already supported by the Oracle. For a spell to add support for a token, perhaps it should check for support with `HomoraBank`. Logic to maintain updates and removals from whitelists could also make the architecture more maintainable.

Update: *Partially fixed in PR#108. The handling of the disparate whitelists is improved with some checks added when adding to a whitelist, but it is still possible for the whitelists to be out of sync. For instance, there are no checks in place for when whitelist entries are modified.*

Notes & Additional Information

[N01] Spells attempt to swap 0 tokens

In the various spells, including `BalancerSpellV1`, `SushiswapSpellV1`, and `UniswapV2SpellV1`, there is logic to initiate a swap of tokens that comprise a liquidity pool.

In each spell, a user can pass along some amounts of `tokenA` and `tokenB` that they need to repay the `HomoraBank`, as well as amounts they want to withdraw. The sum of these values is the "desired" amount of `tokenA` and `tokenB` that must be withdrawn from the protocol in question. Where sufficient `tokenA` has been withdrawn, but not enough `tokenB`, the protocol attempts to swap some `tokenA` for `tokenB` to obtain the desired amounts for both tokens. The same logic exists for the opposite situation.

The issue is that the conditional check that controls when a swap should occur includes the situation where there is no excess token to be swapped – as it uses a strict equality. In this situation an amount of `0` is passed to the swap function as the amount of tokens to sell, which will fail.

Consider making the conditionals that control the swap attempts strict inequalities to avoid wasting gas in some edge cases, and also to further align the code with intentions.

Update: *Fixed in PR#109.*

[N02] `CurveSpellV1` unnecessarily has a WERC20

Upon creation, `CurveSpellV1` is provided a `WERC20`, which is passed to `BasicSpell`'s constructor to initialize a `IWERC20` global variable. Within `BasicSpell` the global variable is used within two helper functions for child contracts to use. However, neither of these functions are ever used in `CurveSpellV1`. It also never accesses the variable directly. In fact, the `WERC20` provided is never utilized by the spell, it is only provided one because the inheritance of `BasicSpell` requires it.

Consider redesigning the inheritance model so that unnecessary contracts and variables are not inherited by contracts that

do not need them. This will simplify the codebase and increase readability.

Update: *The Alpha team decided not to fix this issue.*

[N03] Encoding pool information may lack space for proper mapping

Within the `WMasterChef` and `WLiquidityGauge` contracts, there are functions to encode various aspects of liquidity pool deposits inside a single `uint256` value. The functions achieve this by mapping aspects of the deposit – including the `pool id` – to a subset of bits inside the `uint256`. The limited number of bits available to perform these mappings effectively places upper bounds on the value of particular details that can be encoded.

For instance, in `WMasterChef` the relevant `pool id` is encoded with 16 bits and in `WLiquidityGauge` the `pool id` is encoded with only 8 bits. What this means in practice is that the largest `pool id` that can be encoded by these contracts is `65535` and `255`, respectively.

While this limitation exists locally, the liquidity pool providers themselves do not have the same upper bounds on the number of pools that can be created or the maximum `pool id`s that may result. Pools with `id`s exceeding local upper bounds would simply be incompatible with the local encoding scheme and would remain unavailable for use within HomoraBank V2.

While 16 bits is likely a reasonable upper bound for `pool id`s, 8 bits is significantly more restrictive. Where possible, consider using as many bits as possible to locally encode values that do not have similarly restrictive external upper bounds.

Update: *Fixed in PR#110.*

[N04] Gas inefficiencies

Throughout the codebase, there are several opportunities to improve gas efficiency. Below is a non-exhaustive list of such opportunities:

- Throughout the spells, calls are made to `bank.POSITION_ID`, just to pass the value into `bank.getPositionInfo`. To reduce these two external calls to just one, consider implementing a `getCurrentPositionInfo` function.
- The require on line 92 of the `AggregatorOracle` is redundant. A primary source cannot be set that does not meet this condition. Consider removing the redundant require.
- The `poke` modifier calls `accrue` before proceeding with the rest of the modified function code. The `accrue` function requires that `bank.isListed`, but this check is often replicated at the beginning of the `poke` modified functions. Consider removing the check that `bank.isListed` from functions that use the `poke` modifier.
- Line 89 of `CurveSpellV1` encodes the tokenID, just to decode it again, and then looks it up in the `gauges` mapping. However the `gauges` mapping is publicly accessible, so it would be cheaper to look up `gauges[pid][gid].impl.lp_token` directly. Consider removing the unnecessary encoding on this line.
- At the end of `BalancerSpellV1`'s `addLiquidityInternal` function, it looks up the current LP token balance. Immediately after calling `addLiquidityInternal`, `addLiquidityWERC20` also looks up the current LP token balance. Consider returning the LP token balance from `addLiquidityInternal` to avoid fetching it twice.
- In `UniswapV2SpellV1` and `SushiswapSpellV1`, `removeLiquidityInternal` and `addLiquidityInternal` call `getPair` to fetch the LP token for a given pair. However all functions that call these functions also execute a call to `getPair` previously. Consider passing the LP token address to `removeLiquidityInternal` and `addLiquidityInternal` so that a duplicate call to `getPair` is not necessary.
- The functions `mint` and `burn` in the `WStakingRewards` contract each perform actions on the `staking` contract before fetching the `rewardPerToken`. However, due to the fact that the actions performed on the `staking` contract have already called `rewardPerToken` themselves, `WStakingRewards` can make a cheaper call to the stored `rewardPerTokenStored` instead.

Update: *Partially fixed in PR#111. Alpha have chosen not to change the final point, stating "IStakingRewards do not necessarily have rewardPerTokenStored, so we'll keep the implementation as is."*

[N05] Inconsistent coding style

There are general inconsistencies and deviations from the Solidity Style Guide throughout the codebase. These may lead to misconceptions and confusion when reading the code. Below is a non-exhaustive list of inconsistent coding styles observed.

Typically `internal` function names should begin with a leading underscore, while `public` and `external` functions should not. This makes it clear to readers which functions are publicly accessible. However, throughout the codebase, some `internal` function names start with an underscore, while others do not. For example:

- `_setPrimarySources` is internal and has a leading underscore.
- `doTransmitETH` is internal and does not have a leading underscore.

Some parameters lead with an underscore, while some do not. For example:

- The `_pendingGovernor` parameter in the `setPendingGovernor` function.
- The `token` and `amount` parameters in the `doTransmit` function.

Some global values are defined in all capitals, however this style should be reserved for constants. This can lead users to believe that certain values cannot be changed, when in reality they can be. For example:

- These 4 variables in `HomoraBank` are mutable but defined in capitals.
- Function `EXECUTOR` is a function that returns a value frequently updated.

Some functions use named return values, while others don't. For example:

- `optimalDeposit` names return variables, and uses those variables.
- `_optimalDepositA` does not name the return variables.
- `encodeId` names the return variable, but never uses this name. This is a waste of gas.

Within global variables, some contract addresses are stored as their interface type, while others are stored as addresses and only cast to their interface when the variable is used. For example:

- `IBank bank` is passed to the constructor as an `IBank`, and stored as the same.
- `IWERC20 werc20` is passed to the constructor as an `address`, and cast to an `IWERC20` for storage.
- `address weth` is passed to the constructor as an address and stored as the same. However when it is used, it is cast to `IWETH`.

Consider enforcing a standard coding style, such as that provided by the Solidity Style Guide, to improve the project's overall legibility. Also consider using a linter like Solhint to define a style and analyze the codebase for style deviations.

Update: *Not fixed. The Alpha team state that "we'd like to keep as is, as it doesn't affect the core logic at all. In the future, we'll use solhint to help with the coding style."*

[N06] Lack of input validation

Although most of the functions throughout the codebase properly validate function inputs, there are some instances of functions that do not. One example is:

- `setOracle` accepts the zero address.

Consider implementing require statements where appropriate to validate all user-controlled input, including governance functions, to avoid the potential for erroneous values to result in unexpected behaviors or wasted gas.

Update: *Fixed in PR#113.*

[N07] Lack of explicit visibility in state variable

The `pairs` mapping is using the default visibility.

To favor readability, consider explicitly declaring the visibility of all state variables and constants.

Update: *Fixed in PR#114.*

[N08] `BasicSpell` not marked as abstract

In Solidity, the keyword `abstract` is used for contracts that are not functional contracts in their own right. Such contracts must be inherited to create functional contracts. The `BasicSpell` contract is comprised of a base set of functions, intended to be used by inheriting contracts to interact with the `HomoraBank`. Consider marking the `BasicSpell` as `abstract` to clearly signify that the contract is a base contract designed to aid contracts that inherit it.

Update: *Fixed in PR#115.*

[N09] Divisions performed mid calculation

Due to the fact that Solidity truncates when dividing, performing a division in the middle of a calculation can result in truncated amounts being amplified by future calculations. There are a few instance in the codebase where division is performed mid-calculation. For example:

- In the `addLiquidityInternal` function
- In the `convertForLiquidation` function, where division is performed at the end of each line, but each line is part of a larger calculation.
- In the `_optimalDepositA` function

Being mindful of overflows, consider changing the order of operations where possible such that truncating steps are performed last to minimize any unnecessary loss of precision.

Update: *Acknowledged. Alpha stated: "These are the intended orders since multiplications can overflow. Each number (after division) should maintain an extra precision of 1e18, which should be enough.*

[N10] Missing Natspec

While the majority of the codebase is well-commented, with Natspec used for most functions, there are some places that Natspec is missing or incomplete. Some examples of this are:

- `setWhitelistUsers` has no Natspec.
- `getPositionDebtShareOf` does not describe its parameters. The same is true of `getPositionDebts`.
- Natspec is missing throughout `BNum`. Though this contract is out of the scope of this audit, its functions are used in the scope of the audit.
- The `@return` tag to describe return parameters is not used except in `WMasterChef`.

Consider updating the above examples and checking that complete Natspec exists for all functions throughout the codebase.

Update: *Partially fixed in PR#116. Points 1 and 2 above were corrected, however the team chose not to correct points 3 and 4.*

[N11] Naming issues hinder understanding and clarity of the codebase

To favor explicitness and readability, several parts of the contracts may benefit from better naming. Our suggestions are to rename:

- `event SetPrimarySource` to `event SetPrimarySources`.
- `getPair` in `BalancerSpellV1`, `SushiswapSpellV1`, and `UniswapV2SpellV1` to `getAndApprovePair`.
- `px` to `price` or `ethPerX`.
- `collateralSize` to `collateralAmount`.
- `allBanks` to `tokensWithBank`.
- `allowContractStatus` to `allowContractCalls`.
- `balanceOfERC20` to `localBalanceOfERC20`.
- `struct Oracle` to `struct TokenFactors`.
- `stSushi` and `enSushi` to `startSushiRewards` and `endSushiRewards`, respectively.
- `stRewardPerToken` and `enRewardPerToken` to `startRewardPerToken` and `endRewardPerToken`, respectively.

Consider renaming these parts of the contracts to increase overall code clarity.

Update: *Partially fixed in PR#117. Some of the above recommendations were implemented.*

[N12] No SPDX Identifiers

Consider adding SPDX License Identifiers to the top of each Solidity file in your codebase. Declaring the license associated with your codebase in this manner can help mitigate licensing confusions, and thereby increase community involvement, contributions, and collaborations.

Update: *Fixed in PR#118.*

[N13] Not using `delete` to zero values

In the `unsetOracles` function within the `ProxyOracles` contract, when entries inside the `oracles` mapping are "unset", they are manually replaced with an empty Oracle struct.

To simplify the code and clarify intent, consider using `delete` instead.

Update: *Fixed in PR#119.*

[N14] Using `now` instead of `block.timestamp`

Throughout `SushiswapSpellV1` and `UniswapV2SpellV1`, `now` is used rather than `block.timestamp` to refer to the block time. This term can be misleading and is deprecated in more recent versions of Solidity.

Consider using `block.timestamp` for clarity and to facilitate future upgrades.

Update: *Fixed in PR#120.*

[N15] Using outdated OpenZeppelin contracts

The codebase relies on OpenZeppelin Contracts version 3.2.0, which is outdated. Considering the Solidity version in use throughout the codebase, the latest compatible version of OpenZeppelin Contracts is 3.4.0. Consider using the most recent version of OpenZeppelin Contracts where possible.

Update: *Fixed in PR#121*

[N16] Visibility too permissive for `ensureApprove`

The `ensureApprove` function accepts a `token` and a `spender` and grants approval for the `spender` for an unlimited quantity of `token`s that belong to the contract making the function call. This function is used extensively, and exclusively, throughout spells that inherit the `BasicSpell` contract. Given that these approvals will only be relevant to the system if they are made from within a spell, the `public` visibility of `ensureApprove` may be unnecessarily permissive.

Consider limiting function visibility where possible to improve the overall clarity and readability of the code.

Update: *Fixed in PR#122.*

[N17] Typos

The codebase contains the following typos:

- `Reserev` should be `Reserve`.
- `the spell approve` should be `the spell has approved`.
- `to the bank` should be `in the bank`.
- `if not exist` should be `if it does not exist`.
- `resolve` should be `resolved`.
- `basis point` should be `basis points`.
- `positon` should be `position`.
- `trigger` should be `triggering`.
- `bank not exists` should be `bank does not exist`. This typo is repeated 5 times in HomoraBank.sol.
- `liqudity` should be `liquidity`. This typo is also present on line 341.
- `pood` should be `pool`.
- `amont` should be `amount`.
- `The pool id that that you received LP token back` should be `The pool id that you will receive LP tokens back to`.

Consider correcting these typos to improve code readability.

Update: *Partially fixed in PR#123 – some of the above recommendations were implemented.*

[N18] Declare `uint` as `uint256`

Throughout the codebase, the datatype `uint` is used as an alias for the more explicit `uint256`. To favor explicitness, consider declaring all instances of `uint` as `uint256`.

Update: *Acknowledged.*

[N19] Uninformative revert messages in `require` statements

There are several instances in the codebase where `require` statements have ambiguous or imprecise error messages. As error messages are intended to notify users about failing conditions, they should provide enough information so that appropriate corrections can be made to interact with the system. Below is a non-exhaustive list of identified instances:

- Line 29 of `CoreOracle.sol` – consider `Price oracle failure`.
- Line 97 of `CurveSpellV1.sol` – consider `Bad pid or gid`.

Uninformative error messages greatly damage the overall user experience, thus lowering the system's quality. Consider not only fixing the specific instances mentioned above, but also reviewing the entire codebase to make sure every error message is informative and user-friendly.

Update: *Fixed in PR#124.*

[N20] Unnecessary imports and inheritance

The codebase contains a number of instances of unnecessary imports and unnecessary inheritance. Some examples are:

- `HomoraBank` imports and inherits `Initializable`. However, `Governable` already imports and inherits `Initializable`, so this can be removed from `HomoraBank`.
- `CurveSpellV1` imports `IWERC20` and does not use it. This import can be removed.
- `WERC20` imports `IERC20` and `SafeERC20`. However, the latter already imports the former, so `WERC20` need not import `IERC20`. The same is true in `WMasterChef`, `WStakingRewards`, and `WLiquidityGauge`.
- Though out of scope, we did notice that `ERC1155NaiveReceiver` imports OpenZeppelin's `ERC1155Receiver` and `IERC1155Receiver`. The former imports the latter, so explicitly importing the latter is unnecessary.

Consider removing unnecessary imports and inheritance to simplify the codebase.

Update: *Fixed in PR#125.*

Conclusions

0 critical and 1 high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.

ADD COMMENT

Name *

Email *

Website

Save my name, email, and website in this browser for the next time I comment.

Notify me of follow-up comments by email.

Notify me of new posts by email.

POST COMMENT