// HALBORN

Liquis - EVM Contracts

Smart Contract Security Assessment

Prepared by: **Halborn**

Date of Engagement: **July 19th, 2023 - August 23rd, 2023**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	08/18/2023	Ferran Celades
0.2	Draft Review	08/18/2023	Gabi Urrutia
1.0	Remediation Plan	08/18/2023	Ferran Celades
1.1	Remediation Plan Review	08/18/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Ferran Celades	Halborn	Ferran.Celades@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Liquis engaged Halborn to conduct a security assessment on their smart contracts beginning on July 19th, 2023 and ending on August 23rd, 2023. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided four weeks for the engagement and assigned a full-time security engineer to assessment the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some security risks that were mostly addressed by the Liquis team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the bridge code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture and purpose
- Smart contract manual code review and walk-through
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes
- Manual testing by custom scripts
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Local deployment (Hardhat, Foundry)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

The security assessment was scoped to the following smart contracts:

- contracts/core/LiqLocker.sol
- contracts/core/LiqMinter.sol
- contracts/core/Liq.sol
- contracts/rewards/ExtraRewardsDistributor.sol
- contracts/rewards/AuraPenaltyForwarder.sol
- contracts/rewards/PrelaunchRewardsPool.sol
- contracts/rewards/LiqVestedEscrow.sol
- contracts/rewards/LiqMerkleDrop.sol
- contracts/_mocks/uniswap/MockUniswapV2Pair.sol
- contracts/_mocks/uniswap/MockUniswapV2Router02.sol
- contracts/_mocks/MockCrvDepositor.sol
- contracts/_mocks/compounder/MockStrategy.sol
- contracts/_mocks/MockBalInvestor.sol
- contracts/_mocks/MockAuraMath.sol
- contracts/_mocks/curve/MockCurveGauge.sol
- contracts/_mocks/curve/MockERC20.sol
- contracts/_mocks/curve/MockVoting.sol
- contracts/_mocks/curve/MockCurveVoteEscrow.sol
- contracts/_mocks/curve/MockCurveMinter.sol
- contracts/_mocks/curve/MockGaugeController.sol
- contracts/_mocks/curve/MockWalletChecker.sol
- contracts/_mocks/MockFeeTokenVerifier.sol
- contracts/_mocks/balancer/MockBalancerPoolToken.sol
- contracts/_mocks/balancer/MockFeeDistro.sol
- contracts/_mocks/balancer/MockBalancerVault.sol
- contracts/_mocks/balancer/MockRewardPool.sol
- contracts/_mocks/balancer/MockLiquidityGaugeFactory.sol
- contracts/_mocks/balancer/MockBalancerHelpers.sol
- contracts/_mocks/MockVoteStorage.sol
- contracts/_mocks/MockLiqLocker.sol
- contracts/_mocks/IERC20Extra.sol
- contracts/utils/AuraMath.sol
- contracts/utils/Math.sol

- contracts/utils/Permission.sol
- contracts/peripheral/LiquisClaimZap.sol
- contracts/peripheral/ZapInEth.sol
- contracts/peripheral/BalInvestor.sol
- contracts/peripheral/PooledOptionsExerciser.sol
- contracts/peripheral/LiquisViewHelpers.sol

- contracts/peripheral/PoolMigrator.sol
- contracts/peripheral/FlashOptionsExerciser.sol
- contracts/peripheral/GaugeMigrator.sol
- contracts/peripheral/ClaimFeesHelper.sol

- contracts/peripheral/BalLiquidityProvider.sol
- contracts/peripheral/BoosterHelper.sol
- contracts/peripheral/LitDepositorHelper.sol
- contracts/migration/TempBooster.sol
- contracts/interfaces/IVirtualRewards.sol
- contracts/interfaces/IExtraRewardsDistributor.sol
- contracts/interfaces/ICrvDepositor.sol
- contracts/interfaces/IRewardPool4626.sol
- contracts/interfaces/ILitDepositorHelper.sol
- contracts/interfaces/IBasicRewards.sol
- contracts/interfaces/IBooster.sol
- contracts/interfaces/ILiqLocker.sol
- contracts/interfaces/IRewardStaking.sol
- contracts/interfaces/IStrategy.sol
- contracts/interfaces/IGenericVault.sol
- contracts/interfaces/balancer/BalancerV2.sol
- contracts/interfaces/balancer/IBalancerCore.sol
- contracts/interfaces/balancer/IRewardHandler.sol
- contracts/interfaces/balancer/IBalGaugeController.sol
- contracts/interfaces/balancer/IBalPtDeposit.sol
- contracts/interfaces/IVoterProxy.sol
- contracts/interfaces/IChef.sol
- contracts/interfaces/IERC4626.sol
- contracts/interfaces/IBaseRewardPool.sol
- contracts/interfaces/ICrvVoteEscrow.sol

- contracts/interfaces/bunni/IVotingEscrow.sol
- contracts/interfaces/bunni/IFeeDistributor.sol
- contracts/interfaces/bunni/BunniToken.sol

Modified contracts from convex fork under:

- convex-platform/contracts/*

Commit: 0281f72ad4bb0490564cd37e0d278f359b01a7e5

OUT-OF-SCOPE:
Other smart contracts in the repository, external libraries and economical attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	1	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) CONTROLLED PARAMETER CAN LEAD TO INVALID REWARD CALCULATION	Medium (6.2)	SOLVED - 08/18/2023
(HAL-02) EXERCISE DOES INCREASE THE EPOCH ALWAYS	Low (3.1)	SOLVED - 08/18/2023
(HAL-03) REWARD UPDATE SHOULD NOT HAPPEN DURING VESTING	Informational (0.6)	SOLVED - 08/18/2023

FINDINGS & TECH DETAILS

3.1 (HAL-01) CONTROLLED PARAMETER CAN LEAD TO INVALID REWARD CALCULATION - MEDIUM (6.2)

Description:

Some functions under FlashOptionsExerciser.sol such as claimAndExercise, claimAndLock, withdrawAndLock and earned do use an address array as a parameter named _rewardPools. The array values are used as a IBaseRewardPool contract to fetch the earned amount. However, there is no validation on the parameters or white listing. This causes the earned returned value to be manipulated in those causing the exercising of the options to contain a higher amount. The _exerciseOptions function will initiate a flash loan, which will cause the executeOperation function to be triggered on return. At the end, the flashLoanSimple will be executing the executeOperation function, which does trigger the exercise on the olit token. The exercise function will do a transfer(address(0), amount); from the sender, in this case the FlashOptionsExerciser. The user data, corresponding to the manipulated amount will be used as the olitAmount on the olit exercise.

Moreover, this means that if olit tokens are present on this contract, anyone could potentially perform a claim and exercise without even owning any reward on the pools.

BVSS:

AO:A/AC:L/AX:L/C:N/I:C/A:N/D:N/Y:N/R:P/S:C (6.2)

Recommendation:

The olit exercise function will verify that the amount of tokens is valid before exercising. However, as the olit token was not part of the scope of the audit. Full assessment was not possible. However, by itself, the parameter allows full manipulation and the code should be modified to

verify the addresses of the pools

Remediation plan:

SOLVED: The code was changed to use pool IDs instead of addresses. A registry is kept up with the pools and verified against it. The contract is named Booster.

3.2 (HAL-02) EXERCISE DOES INCREASE THE EPOCH ALWAYS - LOW (3.1)

Description:

The `PooledOptionsExerciser` contract does allow calling `exercise` by anyone. This call does increment the `totalWithdrawable` for the current epoch and does increment the epoch. It will then transfer the `amountIn` of `olit` to the contract and `amountOut` of `lit` to the caller. However, if no amounts are transferred, for example having the `amountOut` being zero, the epoch will still increment, making it hard to track the epochs.

BVSS:

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

Recommendation:

It is recommended to check if the `amounts` are not zero. If they are, no epoch increment should happen.

Remediation plan:

SOLVED: The code is now checking if the `amountOut == 0` and returning.

3.3 (HAL-03) REWARD UPDATE SHOULD NOT HAPPEN DURING VESTING - INFORMATIONAL (0.6)

Description:

The notifyRewardAmount function under PrelaunchRewardsPool can be called after the START_VESTING_DATE is reached. This can cause issues with the current vesting rewardRate and reward balances

BVSS:

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:C (0.6)

Recommendation:

It is recommended to add a onlyBeforeDate check with START_VESTING_DATE on the notifyRewardAmount function.

Remediation plan:

SOLVED: The code was changed and a modifier checking the START_VESTING_DATE added.

MANUAL REVIEW

4.1 PrelaunchRewardsPool.sol

- onlyAuthorized does check for the msg.sender to be the owner, which by default is the contract deployer.
- onlyAfterDate does check if block.timestamp is greater than the parameter date.
- onlyBeforeDate does check if block.timestamp is less than the parameter date.
- stakeLit does transfer lit tokens from the sender to the PrelaunchRewardsPool contract. It then uses the LitDepositorHelper to call convertLitToBpt which internally does call _investBalToPool on the abstract contract BalInvestor. This latter does transfer Lit tokens from the caller, which will be PrelaunchRewardsPool to the BALANCER_VAULT pool id of BAL_ETH_POOL_ID. Once joining the pool BPT (BALANCER_POOL_TOKEN) tokens are transferred to the LitDepositorHelper which, if the balance is more than 0 are transferred to PrelaunchRewardsPool. Finally, the code does call _processStake with the BPT amount which does increase the total supply and the balances for the caller.
- _processStake does call using a modifier updateReward for the receiver. The updateReward does call rewardPerToken **which will return 0 if totalSupply is 0. (rewardPerTokenStored will be 0 initially)**. The lastUpdateTime value will be set to either the block.timestamp or periodFinish, the smaller value, this means that past periodFinish no new rewards are produced. Finally, if the account is different from 0 (which will be used to just update reward) the earned function is called, and the value stored under rewards. rewardPerTokenStored will be stored under userRewardPerTokenPaid for the same account.
- updateReward modifier does obtain the rewardPerToken which is based on the lastUpdateTime. If no new reward is updated with update, the old rewardPerTokenStored is returned. Thereafter, lastUpdateTime is updated with Math.min(block.timestamp, periodFinish). If the update is for an account, based on the new rewardPerToken the earnings are calculated and stored under rewards for the account. The last rate for the account is stored under userRewardPerTokenPaid.
- stake does transfer the specified amount of stakingToken from the caller to the PrelaunchRewardsPool contract. It then calls the

_processStake function with the amount and the sender. This will update the totalSupply and balances[msg.sender]
- stakeAll will do the same as stake but with the full balance of sender.
- stakeFor does allow specifying who you are staking to. However, the stakingToken tokens are transferred from the callers balance.
- **The notifyRewardAmount does state in the description that no "pull" method is present. However, a safeTransferFrom is performed before the value updates.**
- The setOwner, setCrvDepositor , setVoterProxy, setRewardToken, recoverRenouncedLiq and recoverERC20 do use onlyAuthorized.
- convert does use the crvDepositor to exchange BPT tokens for Liq tokens, it sets the balance to 0 on the reward pool, subtracts the total supply and sets the isVestingUser.
- The notifyRewardAmount was allowed to be called after the vesting period, which could cause issues with the reward rate. A new onlyBeforeDate(START_VESTING_DATE) modifier was added.
- convert does call updateReward.
- The following idea was tested and verified: Calling claim when rewards[msg.sender] == 0 will cause an underflow and deadlock if claimed[msg.sender] != 0.

4.2 Liq.sol

No issues found, direct fork of Aura.

4.3 LiqMinter.sol

The contract does protect the Liq token minterMint function with a timestamp of 3 years in the future.

No issues found, direct fork of Aura.

4.4 LiqLocker.sol

- The constructor does set the initial epoch based on `rewardsDuration`. Some `0` checks could be implemented.
- `notBlacklisted` does verify if both of the address arguments are not blacklisted. The second argument is only checked if different from the first argument.
- `modifyBlacklist` does allow the owner to change the `blacklist` flag for a given address. Only contract addresses are supported.
- `shutdown` allows the owner to set the `isShutdown` flag.
- Both `recoverERC20` and `addReward` do restrict `stakingToken` as the token address.
- `setApprovals` does set the approval for the same token and address twice, one reseting to `0` and the other to `type(uint256).max`.
- `lock` does internally call `_lock`, the `_lock` function does verify that the sender and `_account` are not blacklisted. It also checks amount and shutdown flag. It uses the `_amount` and adds its value to the balance tracker `locked` value, adding also to total supply.
  - It then checks for the user locks array if any previous lock is present and already unlocked by time. If no lock is present or already unlocked, it pushes the new amount with the unlock time being the end of epoch.
  - If there is already a previous lock for this epoch, it adds the amount to it.
  - It then checks the delegatee for the caller account and increments the `delegateeUnlocks` and adds a `_checkpointDelegate`. **This should be double-checked to make sure that the delegatee cannot use the locked tokens as there is no unlock time verification like done when pushing `LockedBalance`.**
  - Finally, it updates the epoch total supply.
- `_checkpointDelegate` does accept an address and the addition/deduction values. Votes are using `to224` instead of `to112` as the `LockedBalance`.

- delegate does verify that the address is not 0 and that the given delegatee is not the previous one. **There is no way to reset or remove delegation**. It will iterate over all pending userLocks for the upcoming epoch, remove them from the oldDelegatee and transfer them to newDelegatee. A checkpoint with deduction is stored for the old delegatee and with additions to the new delegatee.
- _checkpointsLookup performs a binary search on the epoch and returns the DelegateeCheckpoint.
- findEpochId does return the epoch id since the first epochs created since contract creation. **It will underflow if the _time is prior contract creation. However, the compiler version will catch the bug.**
- totalSupplyAtEpoch does add all previous epoch supplies.
- getReward does allow fetching the reward for any address, stacking is only allowed if the address is the sender.
    - **The _skipIdx version does not check for cvxCrv and stake automatically.**
- getRewardFor does verify that the caller has permissions to grab rewards for the parameter account. It then transfers all rewards to the account but olit tokens, which are transferred to the caller. From the smart contract context, the olit tokens are transferred to authorized OptionsExerciser contracts.
- queueNewRewards does only allow to be called by the approved distributors.
- _checkpointEpoch does verify if epoch times have passed though by using the current block.timestamp. If an epoch has passed, they are pushed with **supply of 0** to the epoch array.

4.5 LiqVestedEscrow.sol

Direct fork/renamed from Aura.

- fund will store for recipients the given amount of parameters. The total amount is transferred from the sender.
- claim does directly allow locking your reward tokens via the LiqLocker.
- cancel does set the totalLocked after the safeTransfer which could lead to a reentrancy if rewardToken is not safe/trusted.

4.6 ExtraRewardsDistributor.sol

Direct fork/renamed from Aura.

4.7 FlashOptionsExerciser.sol

- Constructor does approve oLIT token to spend all weth on this contract. It also approves balVault and litDepositorHelper to spend all LIT tokens from this contract.
- setOwner does check for previous ownership acl.
- exerciseAndLock does call _exerciseOptions which will skip flash loan if amount is zero. **However, in case that anyone does transfer LIT tokens to this contract, _convertLitToLiqLit will be called and tokens deposited even if the amount specified on the parameter was zero as the balance of the contract is used for the deposit.**

Issues:

_exerciseOptions should not rely on the _olitAmount of the parameter, since _rewardPools could be faked and the returned values wrongly used. executeOperation instead of decoding the params it should be using the balance of olit.

At the end, the flashLoanSimple will be executing the executeOperation function, which does trigger the exercise on the olit token. The exercise function will do a transfer(address(0), amount); from the sender, in this case the FlashOptionsExerciser. This means that if olit tokens are present

on this contract, anyone could potentially perform a claim an exercise without even owning any reward on the pools.

Same approach is followed on claimAndQueue in contracts/peripheral/PooledOptionsExerciser.sol

4.8 PooledOptionsExerciser.sol

- The queue function will store into a mapping of the user/epoch the amount of olit queued and also on the totalQueued. However, there is no check if the epoch should be updated or not. Probably assuming epoch will be manually updated on another function.
- The claimAndQueue function will do the same as queue but claiming funds from the provided reward pools as parameter. The balance used to queue is the diff between the before claim and after claim balance.
- The unqueue function, does allow removing from the last epoch and transfers it to the owner.
- The _exerciseAmounts function does add to the olitOracle multiplier the fee amount based on the basis value.
- The withdrawAndQueue and claimAndQueue does allow specifying the reward pool address.

Issues:

Code duplication:

All functions do use the following snippet, it could be extracted into an internal function:

Listing 1

```
1         queued[msg.sender][epoch] -= amount;
2         totalQueued[epoch] -= amount;
```

4.9 PoolMigrator.sol

Direct fork/renamed from Aura.

4.10 BalLiquidityProvider.sol

Direct fork/renamed from Aura.

4.11 BoosterHelper.sol

Direct fork/renamed from Aura.

4.12 ClaimFeesHelper.sol

Direct fork/renamed from Aura.

4.13 GaugeMigrator.sol

Direct fork/renamed from Aura.

4.14 BalInvestor.sol

Abstract contract implementing ways to approve and join the `BAL_ETH_POOL_ID`. It does correctly compute the `_getMinOut` function, by weighting both `amount` and oracle prices with `1e18`. The returned value is not weighted.

4.15 LitDepositorHelper.sol

Does make use of BalInvestor to deposit and convert LIT, WETH or ETH to LIT/WETH and sends to the user BPT tokens from the joining of the pool.

- The _depositFor function does check the asset deposited. It can be LIT, WETH or ETH. In the case of ETH the balance is first deposited on the WETH contract and wrapped. This is required as the _investSingleToPool does only accept two assets, LIT and WETH.

4.16 BaseRewardPool.sol (convex)

Minor formatting changes from the convex fork. Added the getRewardFor function:
- The function calls updateReward for the account parameter and then sends oLIT to OptionsExerciser

4.17 BaseRewardPool4626.sol

Forked from Aura with changes. Implements the transfer function:
- Allows tokenized pool deposits to be transferred

4.18 Permission.sol (convex)

New contract added with whitelisting functionality on a caller. It does allow calling modifyPermission to set a parameter caller as allowed on behalf of msg.sender actions such as getRewardFor on BaseRewardPool.

THANK YOU FOR CHOOSING

// HALBORN