Smart Contract Security Assessment

February 28, 2024

Project Name:

Revert Finance - Vault

Prepared by:

HYDN

Executive Summary

Client Name: Revert Finance

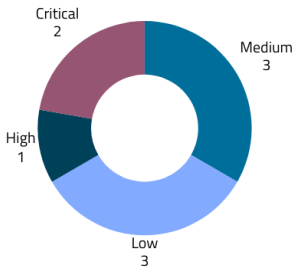
Language: Solidity

Timeline: Delivered 28th February 2024

Repository: https://github.com/revert-finance/lend

Method: Static Automated Analysis + Manual Review

Vulnerability Summary



9	8	1
Total Findings	Resolved	Acknowledged

Contents

Introduction

About HYDN

HYDN is an industry leader in blockchain security and smart contract audits. Founded by Warren Mercer, a world renowned cybersecurity and blockchain expert, who has previously held senior roles at NYSE, Cisco, and Alert Logic. Having been involved in cryptocurrency for over 10 years, Warren is dedicated to making the blockchain ecosystem as secure as it can be for everyone. Warren serves as the CEO for HYDN and heads up the delivery team to ensure that work is carried out to the highest standard.

HYDN worked closely with Revert Finance to consider their unique business needs, objectives, and concerns. Our mission is to ensure the blockchain supports secure business operations for all and he has built the team at HYDN from the ground up to meet this very personal objective.

To keep up to date with our latest news and announcements, check out our website https://hydnsec.com/ or follow @hydnsecurity on Twitter.

Methodology

When tasked with conducting a security assessment, HYDN works through multiple phases of security auditing to ensure smart contracts are audited thoroughly. To begin the process

automated tests are carried out, before HYDN then moves onto carrying out a detailed manual review of the smart contracts.

HYDN uses a variety of open-source tools and analyzers as and when they are required and alongside this, HYDN primarily focuses on the following classes of security and reliability issues:

Basic Coding Mistakes

One of the most common causes of critical vulnerabilities is basic coding mistakes. Countless projects have experienced hacks and exploits due to simple, surface level mistakes that could have been flagged and addressed by a simple code review. The HYDN automated audit process which includes model checkers, fuzzers, and theorem provers analyses the smart contract for many of these basic coding mistakes. Once the automated audit has taken place, HYDN then performs a manual review of the code to gain familiarity with the contracts.

Business Logic Risks

HYDN reviews the platform or projects design documents before analysing the code to ensure that the team has a deep understanding of the business logic and goals. Following this, HYDN reviews the smart contracts to ensure that the contract logic is in line with the expected functionality.

HYDN also analyses the code for inconsistencies, flaws, vulnerabilities, or non-technical business risks which could impact business logic such as tokenomics errors, arbitrage opportunities, and share pricing.

Complex Integration Risks

Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem.
We perform a meticulous review of all of the contract's possible external interactions, and summarise the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

Code Maturity

HYDN reviews the smart contracts for potential improvements in the codebase. These improvements ensure that the code follows industry best practices and guidelines, or code quality standards. Alongside this, HYDN makes suggestions for code optimization items such as gas optimization, upgradeability weaknesses, centralization risks, and more.

Impact Rankings

Once HYDN has completed the security assessment, each issue is assigned an impact rating. This impact rating is based upon both the severity and likelihood of the issue occurring. Each security assessment is different and the business needs and goals, such as project timelines,

and Revert Finance threat modelling. are taken into account when the impact rankings are created.

HYDN assigns the following impact rankings: Critical, High, Medium, Low (listed by severity).

Disclaimer

This HYDN security assessment does not provide any warranties on finding all possible issues within the scope and the evaluation results do not guarantee the absence of any issues. **HYDN cannot make any guarantees on any further code which is added or altered after the security assessment has taken place**. As a single security assessment can never be considered fully comprehensive, HYDN always recommends multiple independent assessments paired with a bug bounty program. This security assessment report should not be considered as financial or investment advice.

Summary of Findings

Executive Summary

From January 22nd 2024 through to February 12 2024, HYDN was tasked with performing a Smart Contract Audit on the Revert Finance Vault Smart Contracts.

During the audit process, HYDN first spent time to understand the business logic and goals of the client on calls and by studying documentation and whitepapers. Following this HYDN carried out a detailed manual analysis of the code.

During the analysis HYDN found 2 Critical Issues, 1 High Issue, and further Medium, Low, and Informational Issues. All of these issues are detailed in the Detailed Findings section of the report.

Project Summary

Delivery Date	February 28th 2024
Language	Solidity
Repository	https://github.com/revert-finance/lend
Scope	https://github.com/revert-finance/lend
Initial Commits	79b4744150faed4745934c0f8b6bcaf31c5d3892

Latest Commit	59cd24db0b5f0f482ea9ad57a8cc09eb2c6f11ed

Breakdown of Findings

ID	Title	Severity	Status
1	Auto Compound Holding Can Be Drained By Reentrancy Attack	**Critical**	Resolved
2	Signature Theft Front Run Can Result In Collateral Takeover	**Critical**	Resolved
3	AutoRange Config: Weak Owner Validation Allows Anyone To Update Config For Any Token	**High**	Resolved
4	Rate Manipulation Can Lead To Illiquid Position	**Medium**	Acknowledged
5	Auto Range Execution Returns Left To The Vault In Place of The User	**Medium**	Resolved
6	Non Standard Token Transfer Not Supported	**Medium**	Resolved
7	Gas Optimization - Simplify Conditions	**Low**	Resolved

8	Transformer Approval Persists After Position Transfer	**Low**	Resolved
9	Explicit Visibility	**Low**	Resolved

Detailed Findings

1 Auto Compound Holding Can Be Drained By Reentrancy Attack [Critical]

1.1 Description

The Auto compound transformer is intended to be called by the Vault (or Operator) to perform a collection of fees, optimally swap one side to the other, and increase the position size.

It intends to be called periodically, when the accumulated fees for a position reach a certain amount. The Pool rate is dynamic, meaning an optimized increase requires a specific ratio in both tokens. Users can swap one side to the other in order to reach the ratio.

Due to the volatile nature of the underlying pool, even with swap amount in one side or the other may remain and can't be added to the position, then auto compound maintains internal balance to allow position owners to withdraw it later, or it will be consumed in the next compounding cycle.

Revert also accumulates platform fees into the contract and can withdraw it at any time. Vault transforms user low level calls to the transformer and allows any arbitrary call. Legitimate users can call autoCompound (or any allowed transformer) with arbitrary data.

Malicious users can call autoCompound with a position he holds (not in Vault) and no validation is performed into auto compound directly. This allows malicious positions to be collected.

Between the storage read of the pending withdrawal balance on the set of the new balance, malicious users can withdraw, so the balance goes to zero and at the end of the compounding execution a new pending balance is set with the assumption that the previous pending balance hasn't changed.

```
111
112         // add previous balances from given tokens
113         state.amount0 = state.amount0 + positionBalances[params.tokenId][state.token0];
114         state.amount1 = state.amount1 + positionBalances[params.tokenId][state.token1];
```

Extract of AutoCompound.sol

```
170         // calculate remaining tokens for owner
171         _setBalance(params.tokenId, state.token0, state.amount0 - state.compounded0 - state.amount0Fees);
172         _setBalance(params.tokenId, state.token1, state.amount1 - state.compounded1 - state.amount1Fees);
```

Extract of AutoCompound.sol

1.2 Impact

This issue will allow a malicious user to withdraw more than they are supposed to and effectively drain any tokens held by the AutoCompound contract that is supposed to be held for other users and/or Revert fees accumulation.

1.3 Recommendations

Following the read check effect pattern the execution of the auto compounding can be fixed and stronger user validation can be added into the Vault to avoid this and also other possible side effects.

The straight response to this issue is to make AutoCompound execute() and withdrawLeftoverBalances() non re-entrant.

1.4 Remediation

Fixed. Re-entrancy is now prevented by Mutex Locker.

2 Signature Theft Front Run Can Result In Collateral Takeover [Critical]

2.1 Description

In order to deposit collateral positions, users can use signature based allowance and call directly the Vault contract with the signature into parameter.

Malicious attackers watching the pending transaction into the mempool can front-run the transaction, steal the victim signature, and deposit the position under his own ownership.

```
108     function executeWithPermit(uint256 tokenId, Instructions memory instructions, uint8 v, bytes32 r, bytes32 s) public returns (
109     {
110         nonfungiblePositionManager.permit(address(this), tokenId, instructions.deadline, v, r, s);
111         return execute(tokenId, instructions);
112
113         // NOTE: previous operator can not be reset as operator set by permit can not change operator - so this operator will sta
114     }
```

Extract of V3Utils.sol

```
329     function createWithPermit(uint256 tokenId, address owner, address recipient, uint256 deadline, uint8 v, bytes32 r, bytes32 s)
330         nonfungiblePositionManager.permit(address(this), tokenId, deadline, v, r, s);
331         nonfungiblePositionManager.safeTransferFrom(owner, address(this), tokenId, abi.encode(recipient));
332     }
```

Extract of V3Vault.sol

2.2 Impact

This issue will result in the loss of control of the position for the victim at the benefit of the attacker.

2.3 Recommendations

In both V3Vault and V3Utils the issue is similar, the signature permits approval to the contract, but the rest of the parameters are not validated, so an attacker can steal the signature and change the other parameters (instruction or recipient).

One strict approach will be to ensure that the caller of contract is the beneficiary- for V3Utils it may be complex due to the versatility of the possible action.

Stronger user validation with signature of the owner for all the parameters will ensure a chain of trust and block the theft of the signature, whilst maintaining user experience by having full signature based transaction.

2.4 Remediation

Fixed. Permit transfer now ensure caller is the owner.

3 AutoRange Config: Weak Owner Validation Allows Anyone To Update Config For Any Token [High]

3.1 Description

Auto range transformer intends to move a position from one range to any other range. The owner of the position can set a configuration to the scale of the range to maintain around the current price over time, the fees plan, or swap slippage tolerance etc.

The Vault transform function allows calling the transformer with arbitrary data and due to weak validation of configToken() function any vault user can modify any position configuration.

The ConfigToken() intends to allow the user to configure the position that he holds, or a position that is deposited into the vault. The situation where the vault itself calls the function allows it to pass bypass the access control.

```
218     function _validateOwner(uint tokenId, address vault) internal returns (address owner) {
219         owner = nonfungiblePositionManager.ownerOf(tokenId);
220         if (vault != address(0)) {
221             if (!vaults[vault]) {
222                 revert Unauthorized();
223             }
224             owner = IVault(vault).ownerOf(tokenId);
225         } else {
226             owner = nonfungiblePositionManager.ownerOf(tokenId);
227         }
228 
229         if (owner != msg.sender) {
230             revert Unauthorized();
231         }
232     }
```

Extract of AutoRange.sol

3.2 Impact

This configuration is very sensitive in situations where the majority of the pool liquidity is under Revert Vault or autoRange control. This may completely change the liquidity distribution, allowing rate manipulation in front of a swap, etc.

If particularly well timed, with a sandwich attack this will result in loss in a position value.

3.3 Recommendations

The input validation of configToken is too permissive, it needs to be strictly defined.
Forbidding the vault to call it directly will block malicious users from bypassing the control.

3.4 Remediation

Fixed. Vault is not allowed anymore to call configToken(), _validateOwner() to ensure the caller is not a vault.

4 Rate Manipulation Can Lead To Illiquid Position [Medium]

4.1 Description

The Vault requires both tokens of the collateral being used to be supported. An attacker can craft a pool with both supported tokens and specific tier fee on UniswapV3, where the pool does not exist or where liquidity is extremely low.

The Attacker can deposit into Vault a position, then Borrow and with a tiny swap move the price out the Oracle Deviation Tolerance.

This will cause the position to become unhealthy, meaning that liquidation can't occur because the current price of the collateral pool is out of the tolerable range.

```
620    function liquidate(LiquidateParams calldata params) external override returns (uint amount0, uint amount1) {
621
622        /*...*/
623        if (transformedTokenId > 0) {...}
626
627        LiquidateState memory state;
628
629        (state.newDebtExchangeRateX96, state.newLendExchangeRateX96) = _updateGlobalInterest();
630
631        uint debtShares = loans[params.tokenId].debtShares;
632        if (debtShares != params.debtShares) {
633            revert DebtChanged();
634        }
635
636        state.debt = _convertToAssets(debtShares, state.newDebtExchangeRateX96, Math.Rounding.Up);
637
638        (state.isHealthy, state.fullValue, state.collateralValue, state.feeValue) = _checkLoanIsHealthy(params.tokenId, state.debt);
639        if (state.isHealthy) {...}
```

Extract of V3Vault.sol

4.2 Impact

Unhealthy positions impact the whole protocol as lenders take on mutualized risks. Low liquidity positions where the current rate is manipulated on purpose will make the liquidators job more complex as upfront liquidation will be required- they will first need to align in range the low liquidity pool with the oracle target price.

4.3 Recommendations

This deviation protection is needed otherwise malicious users will be able to manipulate the exchange rate of the targeted position to make it unhealthy and force liquidation.

Whitelisting supported tokens is a first step, but it allows an exotic mix between the token and fees percentage where the pool does not exist or liquidity is extremely low, meaning it can be manipulated at no cost, and reversely corrected by liquidator.

Considering whitelisted pools with only high liquidity will solve this issue, but then these pools must be monitored to make sure liquidity remains in the same order of magnitude as the reference TWAP pool.

4.4 Remediation

Revert acknowledges this point and by design this extra complexity during liquidation is acceptable. Revert has an automated liquidation bot to handle this edge case.

5 Auto Range Execution Returns Left To The Vault In Place of The User [Medium]

5.1 Description

When the auto range transformer performs an update of the range and prices have moved since the last update of position, the ratio between both tokens may not fit perfectly into the new position, even after swap.

Left amounts in both tokens are returned to the caller- in case this position is under Revert Vault control, this amount will be sent to the Vault, because the Vault is the owner of the position.

```
214             // send leftover to owner
215             if (state.amount0 - state.amountAdded0 > 0) {
216                 _transferToken(state.owner, IERC20(state.token0), state.amount0 - state.amountAdded0, true);
217             }
218             if (state.amount1 - state.amountAdded1 > 0) {
219                 _transferToken(state.owner, IERC20(state.token1), state.amount1 - state.amountAdded1, true);
220             }
```

Extract of AutoRange.sol

5.2 Impact

As the left tokens would be transferred to the Vault after the range moved, it will be lost by the user (position) in profit for the Vault. The Vault admin does still have the ability to withdraw it so it is not lost.

5.3 Recommendations

In a case where the owner of the position is the Vault, the token can be returned to vault.ownerOf(): the lend position owner in place of the liquidity position holder.

Alternatively, on the model of the auto compounding, auto range can keep this left amount and maintain a virtual balance on the behalf of the user that would be consumed during the next range move or withdrawn later by the end owner.

5.4 Remediation

Fixed. AutoRange now returns left over to the real owner in case the position is held by the vault.

6 Non-Standard Token Transfer Not Supported [Medium]

6.1 Description

Unlike other contracts, the LeverageTransformer.sol does not use the safe transfer utility library in order to support non-standard token transfers.

```
83          // send leftover tokens
84          if (amount0 > added0) {
85              IERC20(token0).transfer(params.recipient, amount0 - added0);
86          }
87          if (amount1 > added1) {
88              IERC20(token1).transfer(params.recipient, amount1 - added1);
89          }
90          if (token != token0 && token != token1 && amount > 0) {
91              IERC20(token).transfer(params.recipient, amount);
92          }
```

Extract of LeverageTransformer.sol

```
156
157         // send leftover tokens
158         if (amount0 > 0 && token != token0) {
159             IERC20(token0).transfer(params.recipient, amount0);
160         }
161         if (amount1 > 0 && token != token1) {
162             IERC20(token1).transfer(params.recipient, amount1);
163         }
164     }
```

Extract of LeverageTransformer.sol

6.2 Impact

When a user tries to leverage up, to increase his position, if any dust remains it should be transferred back to the recipient specified in the parameters- the transaction will fail with non-standard tokens.

6.3 Recommendations

HYDN recommends to use the safe ERC20 safe transfer library, in the same way as the other contracts.

6.4 Remediation

Fixed. Leverage transformer uses erc20Safe library to ensure compatibility with non-standard erc20.

7 Gas Optimization - Simplify Conditions [Low]

7.1 Description

Some conditions can be simplified:

1) **Line 219 is unnecessary due to the next if/else statement that override the value**

```
218     function _validateOwner(uint tokenId, address vault) internal returns (address owner) {
219         owner = nonfungiblePositionManager.ownerOf(tokenId);
220         if (vault != address(0)) {
221             if (!vaults[vault]) {
222                 revert Unauthorized();
223             }
224             owner = IVault(vault).ownerOf(tokenId);
225         } else {
226             owner = nonfungiblePositionManager.ownerOf(tokenId);
227         }
228 
229         if (owner != msg.sender) {
230             revert Unauthorized();
231         }
232     }
```

Extract of Automator.sol

2) **Line 244 and Line 247 into if and else are identical, the only difference is the event, the balance update can be moved to the top of the if else statement.**

```
241     function _setBalance(uint tokenId, address token, uint256 amount) internal {
242         uint currentBalance = positionBalances[tokenId][token];
243         if (amount > currentBalance) {
244             positionBalances[tokenId][token] = amount;
245             emit BalanceAdded(tokenId, token, amount - currentBalance);
246         } else if (amount < currentBalance) {
247             positionBalances[tokenId][token] = amount;
248             emit BalanceRemoved(tokenId, token, currentBalance - amount);
249         }
250     }
```

Extract of AutoCompound.sol

7.2 Impact

Leaving these will increase the gas cost at contract deployment time and at execution time.

7.3 Remediation

1. Fixed by remediation of /4.
2. Fixed Conditional logic simplified.

8 Transformer Approval Persists After Position Transfer [Low]

8.1 Description

Vault approveTransform() allows the borrower to approve any address as transform executor, on his behalf. As this position is an ERC721, the following may occur:

- User1 may mint it
- Deposit into vault
- approveTransform() for himself
- Redeem the position collateral
- Then transfer to User2.
- The new owner believes he is in full control of the position but if he decides to deposit into the Vault User1 is still able to execute transformers he approved before.

```
384     function approveTransform(uint tokenId, address target, bool isActive) external override {
385         if (loans[tokenId].owner != msg.sender) {
386             revert Unauthorized();
387         }
388         transformApprovals[tokenId][target] = isActive;
389 
390         emit ApprovedTransform(tokenId, msg.sender, target, isActive);
391     }
```

Extract of V3Vault.sol

```
411         // only the owner of the loan, the vault itself or any approved caller can call this
412         if (loanOwner != msg.sender && address(this) != msg.sender && !transformApprovals[tokenId][msg.sender]) {
413             revert Unauthorized();
414         }
```

Extract of V3Vault.sol

8.2 Impact

The likelihood of this scenario is low, because from a realistic point of view users do not trade LP NFT positions. This may mislead new position owners as the approval is persistent.
A third-party protocol may build on top of the Revert vault and be exposed to this scenario.

Malicious users can then execute transformer which may lead to loss for the new owner.

8.3 Recommendations

When a position is closed at Redeem, Replaced, or Liquidate, so into _cleanupLoan() transformApprovals must be cleaned. The current data structure is a nested mapping which will be difficult to iterate over items to remove all of them.

8.4 Remediation

Fixed. Transformer approval is not by /owner, by /token, by /transformer.

9 Explicit Visibility [Low]

9.1 Description

To avoid any confusion, always be explicit, and to keep all the codebase uniform, it's recommended to always define visibility.

```
136     uint transformedTokenId = 0; // transient (when available in dencun)
137
138     mapping(address => bool) transformerAllowList; // contracts allowed to t
139     mapping(uint => mapping(address => bool)) transformApprovals; // owners
```

Extract of LeverageTransformer.sol

9.2 Remediation

Fixed. Visibility explicit.