Smart Contract Security Assessment

June 06, 2023

Project Name:

Looter

Prepared by:

HYDN

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.

The HYDN delivery team has over 10 years combined experience in blockchain, having performed smart contract audits and built security systems for a large range of protocols and companies. HYDN have performed smart contract security services for the likes of SpookySwap, Swapsicle, Nau Finance, CrossWallet, Dancing Seahorse, Octane, Position Exchange, and more.

HYDN worked closely with Looter 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.

About Looter

Looter is a new generation DEX AMM on Arbitrum. Leveraging from novel Game Theory and lottery-like concepts, they aim to rapidly bootstrap network effects through a gamified strategy-oriented user experience.

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 Errors

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, or vulnerabilities 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 Looter 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

As part of this audit, HYDN was tasked with performing a Smart Contract Audit on the Looter smart contracts. HYDN found one Critical vulnerability which can result in loss of assets for Looter users and the protocol. Alongside this, HYDN found several further vulnerabilities ranging from High to Low impact.

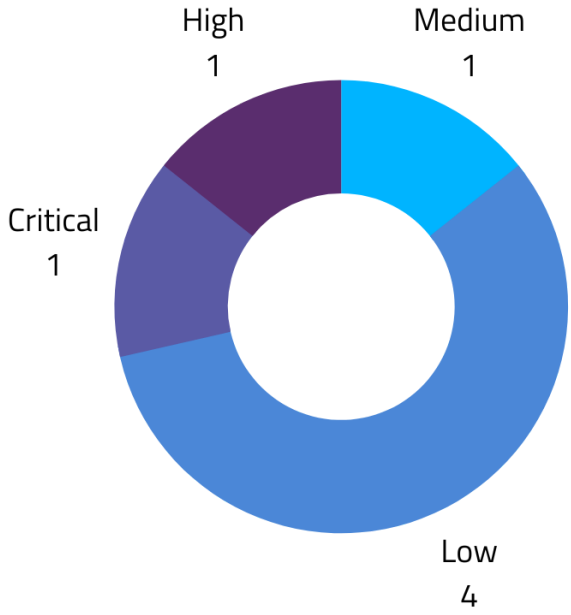
Project Summary

Platform	Arbitrum
Language	Solidity
Scope	https://github.com/0xK3K/vortex
Initial Commits	a659a03c56dc55d7f03b2e073e35b5b10b8b0e8a
Last Commits	d98ed278605c508732664aaa14db85ec0a7f5547

Audit Summary

Delivery Date	June 05 2023
Audit Method	Static Automated Analysis + Manual Review

Breakdown of Findings



ID	Title	Severity	Status
1	Chest Rollover Rate Open To Manipulation	Critical	Resolved
2	Weak Input Validation Triggers Chest Rollover	High	Resolved
3	Buy Back Rate Open To Manipulation	Medium	Resolved
4	Explicit Variables Visible	Low	Resolved
5	Simplify Raw Slot Getters	Low	Resolved
6	Shadowing Variable Declaration	Low	Acknowledged
7	Remove Unused Variables and Parameters	Low	Acknowledged

Detailed Findings

1 Chest Rollover Rate Open To Manipulation [Critical]

1.1 Description

Firstly, the Chest rollover function is predictable and can be accurately timed. This is due to the threshold formula, which can be activated by any minor addition or increase in position from the NFTPool. A malicious actor can exploit this by waiting until the last possible moment, then adding or creating the necessary position to trigger the closure/creation of a new chest.

As outlined in the comment on line 191 of Chest.sol, the proportion of USDC held by the old chest equates to 30% (subject to parameter settings). These funds are then transferred to Babburuzu.sol, which is responsible for selling it for Loot tokens and subsequently burning the received Loot.

A malicious actor could manipulate the Loot/USDC pool rate by swapping large quantities of USDC, then triggering the Chest rollover. To exploit this situation, they could inflate the Loot/USDC rate, thereby minimizing the amount of Loot burned. After the creation of the new chest, the actor could then sell back all the Loot acquired in the first step, resulting in a net profit in USDC.

Importantly, all these steps could be executed within a single transaction, further simplifying this exploit for a malicious actor.

```
183         // if threshold is reached, pull the winning number and create the next chest
184         if (thresholdReached()) {
185             // request winning ticket from factory
186             IChestFactory(factory).requestRandomNumber(chestLevel);
187 
188             uint256 buybackAndBurnAmount = IERC20(USDC).balanceOf(address(this)).mul(BUYBACK_AND_BURN_FEE).div(FEE_DENOMINATOR);
189             uint256 nextChestFees = IERC20(USDC).balanceOf(address(this)).mul(NEXT_CHEST_FEE).div(FEE_DENOMINATOR);
190 
191             // send 30% to Babburuzu for buyback and burn
192             address babburuzu = IChestFactory(factory).depositor();
193             IERC20(USDC).transfer(babburuzu, buybackAndBurnAmount);
194             IBabburuzu(babburuzu).buybackAndBurn();
```

Extract of Chest.sol

1.2 Impact

The intended process, which involves selling a portion of the chest at the market price to purchase and burn Loot, can be circumvented by a malicious actor for their own advantage. This exploitation would yield a net profit for the actor.

Moreover, it would result in a minimal or non-existent buyback and burn quantity of Loot, contrary to the original design of the mechanism.

1.3 Recommendations

Addressing this issue is not straightforward and necessitates a redesign of the process. If we wish to maintain the current approach, one possible solution could be to rely on an oracle, incorporating an acceptable slippage for the Loot/USDC rate. Given that Loot would only be exchangeable on this pool, establishing such a setup could be complex. Calculating a TWAP might be a viable approach, but it is not immune to rate manipulation. The challenge for a malicious actor would be greater, however, as they would need to manipulate the rate over a longer period, spanning multiple blocks.

An alternative approach could involve breaking the circuit and implementing an admin-restricted function to trigger the Chest rollover. This function would be invoked at unpredictable moments, thereby increasing the difficulty for any malicious actor trying to exploit the system.

1.4 Remediation

Automatic buy back has been removed.

2 Weak Input Validation Triggers Chest Rollover [High]

2.1 Description

Keeping scenario 1/ in mind, we can identify a straightforward flow that could result in similar negative impacts, and potentially even more harmful situations. When opening or increasing NftPosition, the dispatch function from Babburuzu.sol is called internally. As this dispatch function is not properly protected, a malicious actor could inject a 'fake' pair parameter (referenced in the extract below) and then call the chest deposit function.

An additional concern is the potential to inject a specially crafted contract as a pair, with the intent to drain any token or native asset held within Babburuzu.sol. While this is not the expected behavior, if any tokens were to become stuck in the contract, the attacker could force a swap. With prior rate manipulation, the malicious actor could effectively transfer any funds stored by the contract to themselves.

```
107     /**
108      * @dev Dispatch pool fees between devWallet, buyback and burn, and chest
109      */
110     function dispatch(address pair) override external {
111         address token0 = ILooterPair(pair).token0();
112         address token1 = ILooterPair(pair).token1();
113         uint256 initialToken0Amount = IERC20(token0).balanceOf(address(this));
114         uint256 initialToken1Amount = IERC20(token1).balanceOf(address(this));
115         uint256 initialUSDCBalance = IERC20(USDC).balanceOf(address(this));
116         uint256 initialETHAmount = address(this).balance;
117 
118         _removeLiquidity(ILooterPair(pair), token0, token1);
119         uint256 token0DispatchAmount = IERC20(token0).balanceOf(address(this)).sub(initialToken0Amount);
120         uint256 token1DispatchAmount = IERC20(token1).balanceOf(address(this)).sub(initialToken1Amount);
121 
122         // transfer 20% to dev wallet
123         if (token0 != WETH) {
124             uint256 devWalletAmount = token0DispatchAmount.mul(2).div(10);
125             IERC20(token0).transfer(devWallet, devWalletAmount);
126             token0DispatchAmount = token0DispatchAmount.sub(devWalletAmount);
127         } else {
128             token0DispatchAmount = address(this).balance.sub(initialETHAmount);
129             uint256 devWalletAmount = token0DispatchAmount.mul(2).div(10);
130             (bool success,) = devWallet.call{value: devWalletAmount}("");
131             require(success);
132             token0DispatchAmount = token0DispatchAmount.sub(devWalletAmount);
133         }
```

Extract of Babburuzu.sol

2.2 Impact

This scenario serves as a shortcut and additionally enables the malicious actor to drain any remaining funds in the contract for any reason. The ability to inject a pair and then trigger a liquidity removal followed by multiple swaps could potentially lead to unpredictable behaviors. For the sake of security, it is safer to avoid this scenario.

2.3 Recommendations

Implementing access control on the dispatch function within Babburuzu.sol could ensure that the caller is a NFTPool. In addition to or as an alternative, calling the getPair function from LooterFactory.sol could verify that this is a legitimate pair created by the factory. This is an underlying security measure, a shortcut, and a potentially effective solution. It would involve calling the nftFactory from within the dispatch function to ascertain that the current caller is indeed a NFTPool.

2.4 Remediation

Resolved.

3 Buy Back Rate Open To Manipulation [Medium]

3.1 Description

The transaction is similar to scenario 1/, but with a different outcome. The creation or increase of a position in the NFTPool could trigger the chest creation explained in scenario 1/, but also within Babburuzu.sol.

The dispatch function receives a portion of the liquidity provider token as fees, which are freshly pulled from the user to create or increase a position. This portion is then burned in exchange for the underlying assets. These assets are subsequently swapped for USDC, and the received USDC is deposited into the chest.

A malicious actor can exploit this mechanism by manipulating the rate of the pool asset/USDC pool prior to the position creation or update. This would ultimately reduce the amount of USDC deposited into the chest, allowing the actor to profit at the expense of other users.

```
146         // split remaining to chest and buyback and burn equally (40% each)
147         _sellTokenFor(token0, token0DispatchAmount.mul(5).div(10), USDC);
148         _sellTokenFor(token1, token1DispatchAmount.mul(5).div(10), USDC);
149         uint256 chestAmount = IERC20(USDC).balanceOf(address(this)).sub(initialUSDCBalance);
150         address chest = IChestFactory(chestFactory).currentChest();
151         IERC20(USDC).transfer(chest, chestAmount);
152         IChest(chest).deposit(chestAmount);
```

Extract of Babburuzu.sol

3.2 Impact

The malicious activity in this scenario is similar to that in scenario 1/, but it serves a different purpose. Here, the vulnerability is exploited to minimize the deposit amount into the chest, as opposed to scenario 1/.

This exploit is entirely feasible, but its real-world impact is likely to be less than it initially appears for several reasons. First, the fees taken from the user's liquidity provider (LP) are likely to represent a small portion of the total, which greatly reduces the economic viability of this exploit due to the cost of rate manipulation. The second reason is that there is no net profit: the malicious actor can recover (or at least reduce) the fees they are supposed to pay on their new or updated position, so it's not a net profit, but rather a reduction of cost. This exploit only affects the actor's own funds.

3.3 Recommendations

Similar to scenario 1/, there is no simple solution to address this issue. The potential solutions are also similar: one option could be to add a circuit breaker and perform the swap and chest deposit manually by an admin at unpredictable intervals. Another alternative could be to implement oracle or TWAP controls to ensure that swaps occur at an acceptable exchange rate.

3.4 Remediation

Automatic buyback have been removed.

4 Explicit Variables Visibility Issues [Low]

4.1 Description

Whilst omitting slot variable visibility is technically accepted by the compiler, it is not best practice and should be avoided.

```
17
18      uint256 constant UINT256_MAX = 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff;
19
```

Extract of Babburuzu.sol

```
26
27      address[] tokens;
28
29      // paths to sell assets via the router
30      mapping (address => mapping(address => address[])) paths;
31
32      bool initialized;
```

Extract of Babburuzu.sol

```
15
16      uint256 constant RECOVERY_TIME_PERIOD = 4 weeks;
17      uint256 constant BUYBACK_AND_BURN_FEE = 3000; // 30% goes to buyback and burn
18      uint256 constant NEXT_CHEST_FEE = 3000; // 30% goes to the next chest when threshold is reached
19      uint256 constant FEE_DENOMINATOR = 10000;
20
```

Extract of Chest.sol

```
19
20      mapping (uint256 => uint256) randomizerRequestIds; // maps request id to specific chest
21
```

Extract of ChestFactory.sol

4.2 Impact

Not explicitly labeling variables can create confusion and may lead to mistakes.

4.3 Recommendations

Explicitly label the visibility of functions and state variables. Functions can be specified as being external, public, internal or private. It is important to understand the differences between them, for example, external may be sufficient instead of public. For state variables, external is not possible.

Labelling the visibility explicitly will make it easier to catch incorrect assumptions about who can call the function or access the variable.

4.4 Remediation

Visibility is explicitly defined.

5 Simplify Raw Slot Getters [Low]

5.1 Description

Access to raw storage values can be directly achieved when visibility is set to public, utilizing a built-in getter without the need for a custom function.

```
131     /**
132      * @dev Returns LOOT's address
133      */
134     function loot() external view override returns (address) {
135         return address(_loot);
136     }
```

Extract of MasterLooter.sol

```
152     /**
153      * @dev Returns YieldBooster's address
154      */
155     function yieldBooster() external view override returns (address) {
156         return address(_yieldBooster);
157     }
```

Extract of MasterLooter.sol

```
130
131     function lpSupply() public view returns (uint256) {
132         return _lpSupply;
133     }
```

Extract of MasterLooter.sol

5.2 Impact

Simpler code is always better. By avoiding getter functions, both deployment costs and execution gas consumption can be reduced.

5.3 Recommendations

Change the visibility to public and access directly without a custom function.

5.4 Remediation

Getter has been removed.

6 Shadowing Variable Declaration [Low]

6.1 Description

Currently more than one variable is defined with the same name within the same accessible scope.

```
86  function buybackAndBurn() override public {
87      uint256 length = tokens.length;
88      for (uint256 k = 0; k < length; k++) {
89          address token = tokens[k];
90          uint256 balance = IERC20(token).balanceOf(address(this));
91          if (balance > 0) {
92              _sellTokenFor(token, balance, WETH);
93          }
94      }
95
96      uint256 balance = address(this).balance;
```

Extract of Babburuzu.sol

6.2 Impact

Shadowing declarations can lead to incorrect assumptions, potential misinterpretations, and code ambiguity.

It's important to avoid this practice to maintain clear and accurate code.

6.3 Recommendations

To avoid confusion, use distinct variable names or reassign values to the same slot. The latter approach may also optimize gas usage.

6.4 Remediation

Acknowledged, but unresolved.

7 Remove Unused Variables and Parameters [Low]

7.1 Description

Currently there are function parameters or local function variables which are never used.

```
171     * @dev Remove liquidity from pool prior to dispatch
172     */
173    function _removeLiquidity(ILooterPair pair, address token0, address token1) internal {
174        uint256 liquidity = pair.balanceOf(address(pair));
175        uint256 totalSupply = pair.totalSupply();
```

Extract of Babburuzu.sol

```
157     /**
158      * @dev Returns general "pool" info for this contract
159     */
160     function getPoolInfo() external view override returns (
161         address lpToken,
162         address loot,
163         address xLoot,
164         uint256 lastRewardBlock,
165         uint256 accRewardsPerShare,
166         uint256 lpSupply,
167         uint256 allocPoint
168     ) {
```

Extract of NFTPool.sol

7.2 Impact

This will lead to extra and unnecessary deployment and execution gas costs.

7.3 Recommendations

Remove unnecessary parameters and remove unused local variables.

7.4 Remediation

Acknowledged, but unresolved.