Protectorate Security Audit

Report Version 1.0

March 18, 2024

Conducted by **Hunter Security**:

George Hunter, Lead Security Researcher

Table of Contents

1 About Hunter Security

Hunter Security is team of independent smart contract security researchers. Having conducted over 50 security reviews and reported tens of live smart contract security vulnerabilities, our team always strives to deliver top-quality security services to DeFi protocols. For security review inquiries, you can reach out to us on Telegram or Twitter at *@georgehntr*.

2 Disclaimer

Audits are a time-, resource-, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

3.3 Actions required by severity level

- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Hunter Security was engaged by Protectorate to review the ZaarStaking and LuckyBuy smart contracts from March 11, 2024, to March 13, 2024.

Overview

Project Name	Protectorate
Repository	https://github.com/Protectorate-Protocol/protectorate-contracts
Commit hash	81eeca79f331be6ca64123067d8c39bb708fb7a9
Resolution	fc08fecc485daa81f14f87dd91b2b6c97c108ff4
Methods	Manual review

Timeline

-	March 11, 2024	Audit kick-off
v0.1	March 14, 2024	Preliminary report
v1.0	March 18, 2024	Mitigation review

Scope

src/Zaar.sol
src/ZaarStaking.sol
src/LuckyBuy.sol

Issues Found

High risk	0
Medium risk	1
Low risk	1
Informational	1

5 Consultants

George Hunter - a proficient and reputable independent smart contract security researcher with over 50 solo and team security engagements contributing to the security of numerous smart contract protocols in the past 2 years. Previously held roles include Lead Smart Contract Auditor at Paladin Blockchain Security and Smart Contract Engineer at Nexo. He has audited smart contracts for clients such as LayerZero, Euler, TraderJoe, Maverick, Ambire, and other leading protocols. George's extensive experience in traditional audits and meticulous attention to detail contribute to Hunter Security's reviews, ensuring comprehensive coverage and preventing vulnerabilities from slipping through.

6 System overview

The ZaarStaking is a standard staking rewards contract where the staking token is Zaar (a token users can migrate to from PRTC) and the reward token is WETH. Rewards are deposited and distributed via a simple function called *distribute(uint256 amount)*. Unlike the well-known Sushiswap's Masterchef and Synthetix's StakingRewards contracts, the ZaarStaking does not take into account the duration of a stake when computing a user's accumulated rewards. Instead, anyone can stake Zaar tokens at any time and get their proportional share of the rewards for the time they've staked based on how regular and with what amounts the *distribute* function has been called. Users have the option to deposit a certain amount of Zaar tokens using either the *Instant* or *90_days* mode. The *Instant* mode allows users to withdraw their stake at any point in time. The *90_days* mode restricts users to being able to withdraw their staked tokens only after a period of 90 days has passed since their last stake. The incentive for using the latter mode is that users staking using the *90_days* mode receive 3x the rewards they would receive using the *Instant* mode. Claiming of accumulated rewards happens either separately if the user invokes the *claimRewards()* public function or implicitly on every *stake* and *unstake*.

The LuckyBuy is a betting contract where users can place a single bet providing whatever amount of WETH tokens they would like (as long as there is enough balance in the contract for the potential payout) and get an almost-instant result of whether their bet was a winning or losing one. What users are betting for is how small the next retrieved number from Chainlink's VRF would be. Upon calling the *placeBet(uint256 betAmount, uint8 odds)* function, a Chainlink VRF request is sent. Once the LuckyBuy contract receives the generated random number, it would settle the bet by checking whether the 256 modulo of the received random number is smaller than or equal to the passed *odds* upon placing the bet. If that is the case, then the user will be paid their *potentialPayout* which was calculated upon placing the bet considering the likelihood of the positive outcome as well as applying a 7% fee. The owner of the contract has the ability to withdraw any WETH tokens from the contract except the ones that are already reserved as potential payouts to betters.

6.1 Codebase maturity

Code complexity - *Strong*

The contracts are minimalistic, well-designed, and straightforward, implementing no unnecessary complexity, inherited contracts, or internal function calls.

Security - *Strong*

No severe vulnerabilities were identified throughout this audit. The code is well-written and tested.

Decentralization - *Strong*
There are no centralized authorities that can affect users' funds. Both contracts are immutable, and the only admin role is the owner of the LuckyBuy contract, who is only able to withdraw WETH tokens they have deposited into the contract or made as profit from lost bets.

Testing suite - *Satisfactory*
There are numerous unit and integration tests ensuring the correct execution of the core functionality of the contracts. However, several assertions are missing.

Documentation - *Strong*
The provided whitepaper is very thorough and comprehensive, defining the purpose, formulas, and invariants of the protocol.

Best practices - *Strong*
The smart contracts adhere to all best practices and are easily readable and auditable.

6.2 Privileged actors

- LuckyBuy contract's owner - has the privilege to withdraw any WETH funds except those reserved for potential payouts.

6.3 Threat model

What in the contracts has value in the market?

- The staking (Zaar) and reward (WETH) tokens in the ZaarStaking contract and the WETH funds from placed bets and lost bets in the LuckyBuy contract.

What are the worst-case scenarios for both contracts?

- Withdrawing more (or other users') Zaar or WETH rewards than what the user has accumulated, eventually draining the contract.
- A staker being able to withdraw their Zaar tokens earlier than the determined deadline (i.e., the 90 days mode).
- Not being able to unstake/withdraw Zaar tokens or claim WETH rewards (Denial-of-Service).
- Predicting or manipulating the result (random number) when placing a bet.
- The owner withdrawing funds that are allocated towards the potential payouts pot.
- Not being able to settle a bet (Denial-of-Service).
- Incorrectly calculating an appropriate payout for a placed bet (e.g., more than 2x for a 50% odds bet).

What are the main entry points and non-permissioned functions?

- *ZaarStaking.stake(uint256 amount, StakingPeriod duration)* - Allows a holder of Zaar tokens to stake them for a certain period of time.
- _ZaarStaking.unstake(uint256 *amount)* - Called by Zaar stakers to withdraw their staked tokens. Executes a claim of WETH rewards automatically. Does not allow a staker to withdraw their tokens before the according deadline (i.e., the 90 days mode).

- *ZaarStaking.claimRewards()* - Allows a staker to claim their accumulated WETH rewards. Called either externally or internally in *stake*.
- *ZaarStaking.distribute(uint256 amount)* - Allows any address to deposit and distribute WETH rewards proportionally among all stakers.
- *LuckyBuy.placeBet(uint256 betAmount, uint8 odds)* - Allows users to place a bet using WETH where the potential payout is determined based on their fair chance to win considering the passed *odds* (minus a 7% fee). Requests a single random word from a VRF coordinator and saves the bet's data in storage using the Chainlink VRF *requestId* as a key.
- *LuckyBuy.rawFulfillRandomWords(uint256 requestId, uint256[] memory randomWords)* - Called by the Chainlink *vrfCoordinator* as a callback after randomness has been requested in *placeBet*. It calculates the 256 modulo of the received number and calls the internal *settleBet* function, which checks whether the bet is winning and sends the corresponding payout if so.

6.4 Observations

Several interesting design decisions were observed during the review of Protectorate's ZaarStaking and LuckyBuy contracts, some of which are listed below:

- Instead of managing multiple stake positions per address, if a user decides to stake more Zaar tokens, they would be added to their current stake, meaning that the 90-day period will be reset if that's what the user has been using so far.
- A user will receive a 3x multiplier if they have created their stake via the 90-day stake mode option.
- The reward distribution does not consider the time length of stakes but instead rewards stakers based on batches of reward distribution. In other words, there is no specific reward rate per second or day, but the stakers expect to receive regular reward deposits via the *distribute* function.
- The LuckyBuy contract does not allow a user to place a bet if there are not enough tokens in the contract to cover the potential payout which ensures there are always enough funds to pay out a winning bet.
- If a user passes an *odds* value within the 237:255 range, they will certainly lose money due to the applied 7% fee.
- A user can "bridge" PRTC token to ZAAR at a 1:1 ratio which would burn the PRTC tokens by sending them to the 0xdead address and mint the equivalent amount of ZAAR. However, there is no function to allow users to exchange the tokens in the opposite direction at the same ratio, meaning that this migration is irreversible.

6.5 Useful resources

The following resources were used throughout the audit to expedite the understanding phase of the codebase, verify the defined invariants and requirements, and check integrations with external protocols:

- *WhitePaper*
- *Chainlink VRF Research*

7 Findings

7.1 Medium

7.1.1 An adversary can sandwich rewards distribution

Severity: *Medium*

Context: ZaarStaking.sol#L173

Description: The rewards distribution in ZaarStaking occurs when any arbitrary user calls the *distribute* function, which splits the deposited amount proportionally among all stakers based on their staking points.

The issue with this implementation, compared to the standard Sushiswap's Masterchef and Synthetix's StakingRewards contracts, is that if there's a large deposit of rewards, anyone can front-run the distribution transaction, create a huge *Instant* stake, claim a portion of the rewards dedicated to actual stakers, and then immediately withdraw within the same block, effectively stealing from other stakers' shares.

Recommendation: Fixing this issue is non-trivial with the current implementation. Consider documenting this behavior so that rewards are deposited regularly in small chunks to avoid large deposits.

Resolution: Acknowledged. A comment was added as a warning about the above behavior and risk.

7.2 Low

7.2.1 Chainlink VRF potential issues and risks

Severity: *Low*

Context: LuckyBuy.sol#L71-L77

Description: As stated in Chainlink's documentation:

"If you use Subscription funding, make sure your subscription account always maintains at least the minimum subscription balance. Otherwise, randomness requests will be held up to 24 hours and then deleted."

A potential problem in the LuckyBuy contract is that a malicious user may place multiple 1-wei bets in order to drain the subscription balance. There is no incentive for anyone to execute this attack vector other than to block the contract's functionality. Consider whether introducing a minimum bet amount might help reduce the attack surface here.

Additionally, there are 2 other security concerns mentioned in Chainlink's documentation that may affect the LuckyBuy contract:

1. The *fulfillRandomWords()* function should never revert - "If your fulfillRandomWords() implementation reverts, the VRF service will not attempt to call it a second time. Make sure your contract logic does not revert". This is not a severe problem in the LuckyBuy contract simply because it reverts only in cases where we wouldn't like and expect the VRF service to attempt to call the function again.

2. The *requestConfirmations* parameter is set to 3, which represents the number of confirmed blocks the VRF service waits before writing a fulfillment to the chain in order to make potential rewrite attacks unprofitable. Blockchain reorganizations usually have a depth of 1 block on Ethereum Mainnet, so 3 would be appropriate. However, if the contract would be deployed on a different chain, consider adjusting this value appropriately.

Recommendation: Consider the above risks and mitigation suggestions.

Resolution: Acknowledged.

7.3 Informational

7.3.1 Typographical mistakes, code-style suggestions and non-critical issues

Severity: *Informational*

Context: src

Description: The contracts contain one or more typographical issues, code-style suggestions and non-critical issues. In an effort to keep the report size reasonable, we enumerate these below:

1. Consider whether it would be useful to allow withdrawing any tokens from the LuckyBuy contract, not just WETH (i.e., a recovery function for accidentally sent tokens).
2. The expression in *calculatePayout* could be simplified to *betAmount * 256 * 93 / odds / 100*. Additionally, to make it more readable, the literals can be made into constants like *MAX_ODDS_VALUE* (256), *BET_NET_PERCENTAGE* (93), and *PERCENTAGE_BASE* (100).
3. ZAAR's *bridge* and ZaarStaking's *balanceOf* methods are *public* but can be marked *external* as they are never called internally.
4. The *success* status check in *ZAAR.bridge* is not necessary as the *prtc.transferFrom* function never returns false.
5. Consider using the latest Solidity pragma version (currently 0.8.24) to utilize the latest optimizations and compiler bug fixes. Keep in mind to use the *paris* EVM version if the contract will be deployed on blockchains that do not support the PUSH0 opcode yet.
6. Consider marking the event parameter *period* in *UserStaked* as *indexed*.
7. Consider reusing *previewRewards* in *claimRewards* to avoid code duplication and make the contract more readable.
8. A typo in the NatSpec comment for *unstake* - should use *zaar*, not *prtc*.
9. Consider importing *IERC20* in Zaar.sol from the same file as *ERC20*.
10. Consider enforcing stakers to specify the correct stake duration mode in the stake function instead of silently overwriting it to the *90_day* mode if they already have a stake with the *90_day* mode but have passed the *Instant* mode.
11. Consider removing the first check in *unstake* and simply reverting or returning if the passed amount is 0. Given the current behavior, there may occur unexpected issues in external contracts integrating with ZaarStaking.

Recommendation: Consider fixing the above typographical issues, code-style suggestions and non-critical issues.

Resolution: Resolved.