Audit Report

PayRue Staking Contract

Summary and Scope

21	2	0	0
Notes	Warnings	Bugs	Vulnerabilities

This audit covers the **PayRue Staking Contract**. During the audit, attention was paid to design, overall code quality, best practices, business logic and security. Math is to be independently verified by the team.

Git HEAD **9f4684d45c9ec583568b77b88d1f910837e6d5e5** was audited.

This audit covers PayRueStaking.sol from PayRue, and Context.sol, IERC20.sol, Ownable.sol and ReentrancyGuard.sol from OpenZeppelin 4.1.0. No previous audit for these OpenZeppelin components was found.

Although this audit was originally conducted for Ethereum, this particular version has been available on the Polygon blockchain, an Ethereum compatible blockchain, at **0x0dc8c9726e7651afa4d7294fb2a3d7ee1436dd4a** since January 26th 2022.

Purpose

Purpose of the staking smart contract is to provide PayRue customers a way to earn passive income by locking their tokens out-of-circulation for a certain period of time.

Design

The staking system consists of one smart contract, which is well designed for general staking use, and therefore reusable. Although the contract is simple, it's more than the sum of its parts: smart contract design is hard, but the author has pulled it off successfully.

Token agnostic design

Any token conforming to EIP-20 (“ERC-20”) token standard can be used, either as the staking token, reward token, or both. This makes the contract extremely reusable for different kinds of staking-for-reward scenarios, and it provides a good starting point for developing future staking based applications.

Rewards are not reserved

Rewards are allocated on-the-go, and surplus tokens can be withdrawn by the contract operator at all times. While this is a clever way to make managing rewards straightforward, this is a double edged sword: in theory the contract operator could fool users with high excess rewards, and could withdraw the tokens by front-running once the user tries to withdraw the excess rewards (Warning #1). Excess rewards are also not reserved nor allocated to users, leading to a first-come-first-served situation.

Used tools and components

Solidity version 0.8.4 was used, which was neither recent, nor considered safe, this is considered a bad practice. However, Solidity compiler’s errata was reviewed, and no critical issues affecting this particular contract were found. Optimizer is used with 1000 rounds, this suits the project well, although is unconventional.

OpenZeppelin

OpenZeppelin, a high quality collection of reusable Solidity components is utilized for re-entrancy mitigations and access control, and is considered a best practice: OpenZeppelin is developed by industry leaders.

OpenZeppelin version 4.1.0 is used, which is outdated. This is considered a bad practice. However, although the 4.x series had multiple security issues, none seems to affect PayRue’s usage of the library at the time of writing.

Contracts

Whole staking system consists of one smart contract (PayRueStaking.sol), which consists of one file and is 476 lines long. This can be considered a best practice, since the business logic is simple enough for one file. Separating the code into multiple smaller files / contracts could be confusing.

Implementation

The code is exceptionally well designed and implemented: it's well thought out, and organized in a way that is easy to understand.

The Positive

- Admin can do rewarding for the user, this way the contract operator can pay the gas.
    - Improvement proposal: anyone could reward anyone (remove access control), since rewarding itself does not have any downsides.
- Token recovery mechanism is well implemented, and a clever way to handle surplus tokens.
- Storage is cleaned up, this is considered a best practice: it saves gas for the transaction cleaning the storage.
- Although NatSpec is not used, regular commenting utilized in the code is helpful.
- ReentrancyGuard is in place.
    - It's used overzealously, though (see Notes).
- General purpose: works on any EIP-20 token configuration (any EIP-20 token can be either reward token, staking token, or both).

The Neutral

- The access control could be more granular. With Role Based Access Control one could assign different accounts for paying the rewards for users, and another for more critical tasks. See OpenZeppelin Access Control documentation.

The Negative

- Excess funds not allocated per user: first come, first served.
- OpenZeppelin's "`Pausable`" contract should have been utilized instead of inventing one's own mechanism. Utilizing standard interfaces also helps with integration with other 3rd party code.
- Emergency withdrawal should be able to be used by anyone after being enabled by the owner: one can't expect the contract operator to be around forever... ...unlike the smart contract in question.
- NatSpec not used: even the source code might live "forever", having NatSpec will help automatic handling of the source code in the future.
- Functions not completely arranged by visibility, per Solidity Style Guideline.

- By using virtuals, inheritance is taken into account, however, consider using the OpenZeppelin's encapsulation pattern for safer inheritance in the future.
- No interface used for PayRueStaking. Use interfaces for robust integration with other smart contracts in the future.
- User facing functions are not defined `external`. Add `external` for semantic reasons.
- Named returns are used instead of `return`. Nowadays this is considered confusing, and hence is not used by OpenZeppelin. Use `return` for clarity.
- Tokens are not transferred consistently. Maybe use `SafeERC20` from OpenZeppelin?
- Why to have both, emergency withdrawal and pausing? From the user's point of view pausing is essentially a non-penalized emergency withdrawal that the contract operator can turn on and off at will (from the user's point of view both halt staking). Maybe this kind of action should be penalized, as is the case with emergency withdrawal: it can be used only once, so the contract operator is motivated to use it only in need.
    - Another non-penalized way for the contract operator to halt staking is to withdraw all surplus tokens. See the chapter "*Rewards are not reserved*" above for more information.
- `assert()` not used: symbolic execution / formal verification tools can't be used effectively.

Vulnerabilities (0)

Vulnerabilities are bugs which have serious consequences, such as loss of capital.

No vulnerabilities found.

Bugs (0)

Bugs are programming errors which are not serious enough to cause serious consequences, such as loss of capital, but can be problematic in other ways.

No bugs found.

Warnings (2)

Warnings are features which work in a way that could differ from its original purpose, but are not bugs. These might have unintended consequences.

File:Line number	Problem and solution
PayRueStaking.sol:249	By frontrunning, the contract operator can fool a user with high

	rewards by withdrawing the tokens before the user's transaction is mined on-chain.
PayRueStaking.sol:381	Gas usage considerations with loops: withdrawing large amounts from multiple smaller stakes could cost a lot of gas, become expensive and in extreme cases hit the block gas limit. This is not critical though: the user can always withdraw their tokens withdrawing small amounts at once. Discourage small stakes, either by utilizing the minimum staking amount, or by user interfaces.

Notes (21)

Notes are not problems per se, but instead points to pay attention to.

File:Line number	Problem and solution
PayRueStaking.sol:13	Lock Solidity version to prevent this contract being accidentally deployed on a Solidity for which this code wasn't designed for.
PayRueStaking.sol:51	Specify this in the constructor for future deployments.
PayRueStaking.sol:52	Specify this in the constructor for future deployments.
PayRueStaking.sol:56	Replace this with a function: it would not be gas effective, but would always return the actual result. The small gas penalty could be considered as a reasonable tradeoff for accuracy.
PayRueStaking.sol:56	Make this public for future on-chain integrations with other contracts.
PayRueStaking.sol:62	Rename to `pauseStaking`, for clarity.
PayRueStaking.sol:71	Use either interface or contract as the type instead of plain `address`.
PayRueStaking.sol:72	Use either interface or contract as the type instead of plain `address`.
PayRueStaking.sol:92	Separate `stake()` into public `stake()` and internal `_stake()`, so it could be used by derived contracts, just as `_rewardUser()` and `_unstakeUser()` (OpenZeppelin encapsulation design pattern).
PayRueStaking.sol:125	This could be an `assert`, since this should never happen.
PayRueStaking.sol:129	Unclear and misleading comment:

	`userData.storedRewardUpdatedOn` is instrumental for calculating the rewards.
PayRueStaking.sol:180	Practically `stakeable` is shadowed here: avoid shadowing, and use each variable for one purpose only, for clarity.
PayRueStaking.sol:194	Minimize (practically) dead code, and remove functions which are not actually needed, for simplicity.
PayRueStaking.sol:250	Use either interface or contract as the type instead of plain `address`.
PayRueStaking.sol:273	Unnecessary `nonReentrant`.
PayRueStaking.sol:275	Misleading error message.
PayRueStaking.sol:277	Emit an event for better and easier off-chain integration.
PayRueStaking.sol:285	Unnecessary `nonReentrant`.
PayRueStaking.sol:288	Emit an event for better and easier off-chain integration.
PayRueStaking.sol:294	Unnecessary `nonReentrant`.
PayRueStaking.sol:335	Emit an event for better and easier off-chain integration.

Remedies

Only minor problems were found, hence no action is advised: fixing minor issues might cause major issues (regressions).

Conclusion

✅ **No critical issues with the smart contract code found during the audit**. This audit was conducted in accordance with the auditor's best skills and knowledge, but audits are not definite proof of bug free code. Smart contracts are still experimental technology, and can harbor unexpected problems.

About the Author

Ville Sundell has been involved with smart contracts since Bitcoin's "Script", stumbled upon Ethereum in 2015 and published his first Solidity smart contract in 2016. Since his first audit in 2017, he has audited tokens listed on world's leading exchanges such as MATRYX, DAWN and GATENet. His previous Ethereum smart contract audit was of **GATENet Token**.