Zellic

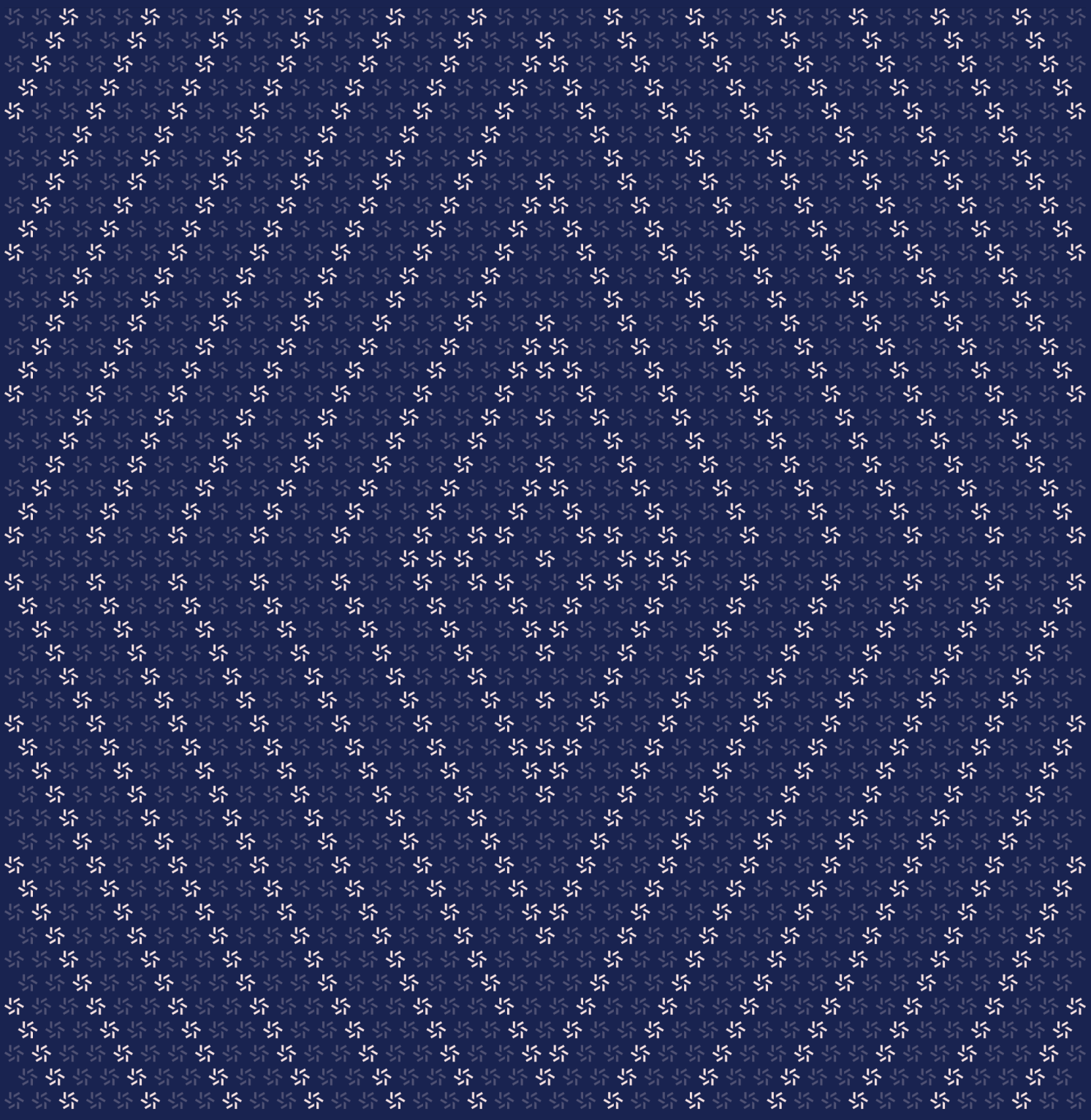
Prepared for
Flap Jackson
Paragons DAO

Prepared by
Junghoon Cho
Mohit Sharma
Zellic

July 12, 2024

PDT Staking V2

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Paragons DAO from July 10th to July 12th, 2024. During this engagement, Zellic reviewed PDT Staking V2's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is it possible to gain disproportionate rewards by manipulating reward pools?
- Are there issues in the protocol math or logic that lead to loss of funds?
- Does PDT Staking V2 have any centralization risks?
- Is PDT Staking V2 secure against common smart contract vulnerabilities?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Reward tokens
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped PDT Staking V2 contracts, we discovered three findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining finding was informational in nature.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	0
Informational	1

2. Introduction

2.1. About PDT Staking V2

Paragons DAO contributed the following description of PDT Staking V2:

> Paragons DAO is a web3 gaming community focused on enabling players and guilds to compete and maximize their rewards through financial tools, shareable assets, edutainment, and competitive opportunities. Paragons is reducing the financial barriers of web3 gaming, to create an ecosystem where everyone wins — players, protocols, and guilds alike.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3. Scope

The engagement involved a review of the following targets:

PDT Staking V2 Contracts

Type	Solidity
Platform	EVM-compatible
Target	pdt-staking
Repository	https://github.com/ParagonsDAO/pdt-staking/ ↗
Version	`5b60196206fc20ff7963a3f1a2466e18f60deac2`

Programs	`contract/StakedPDT.sol`
	`interfaces/IStakedPDT.sol`
	`interfaces/IForkedPDTStakingV2`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of four person-days. The assessment was conducted by two consultants over the course of three calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Junghoon Cho
Engineer
junghoon@zellic.io

Mohit Sharma
Engineer
mohit@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

July 10, 2024	Kick-off call
July 10, 2024	Start of primary review period
July 12, 2024	End of primary review period

3. Detailed Findings

3.1. Reward-token registration is irreversible

Target	StakedPDT.sol		
Category	Protocol Risks	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

The current version of the protocol allows adding a new reward-token address to the `rewardTokenList` using the `registerNewRewardToken` function. However, there is no function implemented to remove a reward token from this list, making the registration of reward tokens irreversible.

Impact

If an invalid address is accidentally registered, or if a registered reward token blacklists the StakedPDT contract address, the protocol becomes unusable.

For example, the following code from `distribute` iterates over `rewardTokenList` and calls `balanceOf` on every registered token.

```
for (uint256 itTokenIndex; itTokenIndex < _nTokenTypes; ) {
    address _token = _tokenList[itTokenIndex];
    uint256 _rewardBalance = IERC20(_token).balanceOf(address(this));
    uint256 _rewardsToDistribute = _rewardBalance - unclaimedRewards[_token];
```

If an invalid address is registered in the `rewardTokenList`, the `balanceOf` call on such address will fail, causing the `distribute` call to always revert. Since the `distribute` function must be called by the admin to start a new epoch, the protocol will be permanently halted and cannot be resolved without a protocol upgrade.

Recommendations

Add a function that can remove a specific reward token from the `rewardTokenList`.

Remediation

This issue has been acknowledged by Paragons DAO, and a fix was implemented in commit f5944102 ↗.

3.2. PDT can be set as a reward token and withdrawn by admin

Target	StakedPDT.sol		
Category	Protocol Risks	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The function `registerNewRewardToken` allows an admin address with the role `TOKEN_MANAGER` to call it.

```
function registerNewRewardToken(address newRewardToken)
    external onlyRole(TOKEN_MANAGER) {
    require(newRewardToken != address(0), "Invalid reward token");

    uint256 numOfRewardTokens = rewardTokenList.length;

    for (uint256 itTokenIndex = 0; itTokenIndex < numOfRewardTokens; ) {
        if (rewardTokenList[itTokenIndex] == newRewardToken) {
            revert DuplicatedRewardToken(newRewardToken);
        }

        unchecked {
            ++itTokenIndex;
        }
    }

    // If newRewardToken is not found in rewardTokenList, then add it
    rewardTokenList.push(newRewardToken);

    emit RegisterNewRewardToken(currentEpochId, newRewardToken);
}
```

The `registerNewRewardToken` function currently lacks a check to prevent the staked PDT token from being registered as a reward token.

Impact

Reward tokens owned by the StakedPDT contract can be withdrawn by the admin using `withdrawRewardTokens`. This leads to a centralization risk as it allows a compromised admin account to withdraw PDT tokens staked by the users.

Recommendations

Add the following check to `registerNewRewardToken`.

```
function registerNewRewardToken(address newRewardToken)
    external onlyRole(TOKEN_MANAGER) {
    require(newRewardToken != pdt, "Invalid reward token");
```

Remediation

This issue has been acknowledged by Paragons DAO, and a fix was implemented in commit afcf3d16↗.

3.3. Epoch length updating is underconstrained

Target	StakedPDT.sol		
Category	Protocol Risks	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The StakedPDT contract allows the admin to change the epoch length via the `updateEpochLength` function. The function adjusts the end time for the current epoch accordingly. However, the function only constrains the epoch length to be nonzero. As such, the admin can adjust the epoch length such that `epoch[currentEpochId].endTime` is in the past. This can lead to a scenario where `epoch[currentEpochId].endTime < contractLastInteraction`.

Impact

If epoch length is set incorrectly, making `epoch[currentEpochId].endTime < contractLastInteraction`, the `contractWeight` function would trigger an underflow and calls to `distribute` will crash, halting the protocol. This can, however, be fixed by the admin at runtime by simply calling `updateEpochLength` again.

Recommendations

Add the following check to `updateEpochLength`.

```
require(epoch[currentEpochId].startTime + newEpochLength >
    contractLastInteraction, "Invalid new epoch length");
```

Remediation

This issue has been acknowledged by Paragons DAO, and a fix was implemented in commit `f12d8412` ↗.

4. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

4.1. Module: StakedPDT.sol

Function: `claim(address to)`

This claims all pending rewards for `msg.sender`.

Inputs

- `to`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Recipient of the claimed rewards.

Branches and code coverage

Intended branches

- Reward details are updated after claim.
  - ☑ Checked
- Expired rewards are not claimed.
  - ☑ Checked

Negative behavior

- Revert if rewards for the current epoch have already been claimed.
  - ☑ Checked

Function: `distribute()`

This updates epoch details if the current epoch has ended.

Branches and code coverage

Intended branches

- Executed successfully if called by `EPOCH_MANAGER` after an epoch has ended.

☑ Checked

- Reward token is ignored for the following epoch if excess balance is not present.
  ☐ Unchecked

Negative behavior

- Revert if the current epoch has not ended.
  ☐ Unchecked
- Revert if not called by `EPOCH_MANAGER`.
  ☐ Unchecked
- Revert if no new rewards are added.
  ☑ Checked

Function: `registerNewRewardToken(address newRewardToken)`

This registers a new reward token.

Inputs

- `newRewardToken`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Cannot be a zero address.
  - **Impact**: Address of the token to be registered as a reward token.

Branches and code coverage

Intended branches

- New reward token is registered when called by `TOKEN_MANAGER`.
  ☑ Checked

Negative behavior

- Revert if not called by the token manager.
  ☑ Checked
- Revert if token is already registered.
  ☑ Checked

Function: `stake(address to, uint256 amount)`

This stakes PDT.

Inputs

- `to`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address that will receive credit for staking.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Cannot be zero.
  - **Impact**: The amount of PDT to stake.

Branches and code coverage

Intended branches

- StakeDetails updated after staking.
  - ☑ Checked
- Liquidity is maintained (i.e., sum of all user weights always equals the contract weight).
  - ☑ Checked

Negative behavior

- Revert if current epoch has ended.
  - ☑ Checked
- Revert if `msg.sender` does not have enough balance or did not set enough allowance for the contract.
  - ☑ Checked

Function: `unstake(address to, uint256 amount)`

This unstakes PDT.

Inputs

- `to`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: The address that will receive the unstaked PDT.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Cannot be zero or more than the amount staked by `msg.sender`.
  - **Impact**: The amount of PDT to unstake.

Branches and code coverage

Intended branches

- Liquidity is maintained (i.e., sum of all user weights always equals the contract weight).
  - ☑ Checked
- Amount of unstaked PDT is consistent with user weight.
  - ☑ Checked

Negative behavior

- Revert if current epoch has ended.
  - ☑ Checked
- Revert if unstake amount is more than the total amount staked by `msg.sender`.
  - ☐ Unchecked

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped PDT Staking V2 contracts, we discovered three findings. No critical issues were found. One finding was of high impact, one was of medium impact, and the remaining finding was informational in nature.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.