Prepared for
Ben Shyong
Anzen Group Ltd.

Prepared by
Jinseo Kim
Seunghyeon Kim
Zellic

May 21, 2024

Anzen Finance
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 Anzen Group Ltd. from May 13th to May 15th, 2024. During this engagement, Zellic reviewed Anzen Finance'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 there a bug resulting in unrecoverable funds?
- Are there calculation errors allowing attackers to drain funds or withdraw disproportionate funds at the expense of others?
- Is there any possibility of flash-loan attacks?

1.3. Non-goals and Limitations

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

- 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.

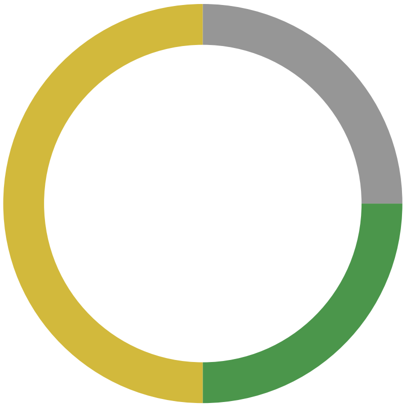
1.4. Results

During our assessment on the scoped Anzen Finance contracts, we discovered four findings. No critical issues were found. Two findings were of medium impact, one was of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Anzen Group Ltd.'s benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	2
🟩 Low	1
◼ Informational	1



2. Introduction

2.1. About Anzen Finance

Anzen Group Ltd. contributed the following description of Anzen Finance:

> USDz is the stablecoin created by Anzen. It is a non-rebasing, permissionless ERC20 token. USDz is always backed 1:1 by SPCT, which represents RWA as real collateral backing USDz.
>
> Through USDz, users can gain exposure to uncorrelated RWA yields within DeFi. These yields are consistent relative to crypto token prices and as such USDz is an attractive building block within DeFi that all users can build on top of.

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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Anzen Finance Contracts

Repository	https://github.com/Anzen-Finance/protocol-v2 ↗
Version	protocol-v2: `0e247bda782c22596b7a3486dde0a7729206b61f`
Programs	• Canto_childUSDz • childUSDz • SPCTPool • SPCTPriceOracle • sUSDz • USDz • USDzFlat • USDzPriceOracle • vault
Type	Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of four and a half person-days. The assessment was conducted 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:

Jinseo Kim
Engineer
jinseo@zellic.io ↗

Seunghyeon Kim
Engineer
seunghyeon@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

May 13, 2024	Kick-off call
May 13, 2024	Start of primary review period
May 15, 2024	End of primary review period

3. Detailed Findings

3.1. Wrong fee mechanism in `redeemBackSPCT`

Target	USDz		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `redeemBackSPCT` function in the USDz contract redeems USDz tokens to the corresponding SPCT tokens, taking a redemption fee:

```
function redeemBackSPCT(uint256 _amount)
    external whenNotPaused checkCollateralRate {
    require(_amount > 0, "REDEEM_AMOUNT_IS_ZERO");
    require(!_blacklist[msg.sender], "SENDER_IN_BLACKLIST");

    // calculate fee with SPCT
    if (redeemFeeRate == 0) {
        _burnUSDz(msg.sender, _amount);
    } else {
        uint256 feeAmount = _amount.mul(redeemFeeRate).div(FEE_COEFFICIENT);
        uint256 amountAfterFee = _amount.sub(feeAmount);

        _burnUSDz(msg.sender, amountAfterFee);

        if (feeAmount != 0) {
            _transfer(msg.sender, treasury, feeAmount);
        }
    }

    IERC20(address(spct)).safeTransfer(msg.sender, _amount);

    emit Redeem(msg.sender, _amount);
}
```

This function internally transfers the fee to the treasury and burns the rest. However, SPCT tokens of the `_amount`, which is the amount that fee is not deducted from, is transferred to the caller.

Impact

A user would be able to redeem their USDz tokens into SPCT tokens without paying the expected fee. This can lead to a mismatch between the total backed SPCT tokens and the minted USDz tokens, breaking the assumption that each USDz token is fully backed by an equivalent amount of SPCT tokens.

Recommendations

If fees are enabled, consider transferring SPCT tokens of the fee-deducted amount using `amountAfterFee`.

Remediation

This issue has been acknowledged by Anzen Group Ltd., and a fix was implemented in commit `8c70d55a` ↗.

3.2. Transfer event is emitted twice for minting or burning USDz

Target	USDz		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The USDz contract has internal helper functions for minting and burning USDz tokens, `_mintUSDz` and `_burnUSDz`:

```
function _mintUSDz(address _receiver, uint256 _amount) internal {
    _mint(_receiver, _amount);
    totalPooledSPCT = totalPooledSPCT.add(_amount);
    emit Mint(msg.sender, _amount);
    emit Transfer(address(0), _receiver, _amount);
}

function _burnUSDz(address _account, uint256 _amount) internal {
    _burn(_account, _amount);

    totalPooledSPCT = totalPooledSPCT.sub(_amount);
    emit Burn(msg.sender, _amount);
    emit Transfer(_account, address(0), _amount);
}
```

These functions call the `_mint` or `_burn` functions in the OpenZeppelin ERC20 implementation, record the pooled amount of SPCT tokens, and emit the events `Burn` and `Transfer`.

However, the `_mint` and `_burn` functions also emit the `Transfer` event via calls to the `_update` function:

```
function _mint(address account, uint256 value) internal {
    // ...
    _update(address(0), account, value);
}

function _burn(address account, uint256 value) internal {
    // ...
    _update(account, address(0), value);
}
```

```
function _update(address from, address to, uint256 value) internal virtual {
    // ...
    emit Transfer(from, to, value);
}
```

As a result, the identical `Transfer` event would be emitted twice when USDz tokens are minted or burnt.

Impact

Although this issue will not affect the business logic of this contract, it is possible that an off-chain infrastructure depending on this event fails to properly track the minting and burning of USDz tokens.

Recommendations

Consider removing the code that emits the `Transfer` event, as it is already being emitted.

Remediation

This issue has been acknowledged by Anzen Group Ltd., and a fix was implemented in commit `66273edb` ↗.

3.3. Protection logic in `rescueERC20` can be bypassed

Target	sUSDz		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The sUSDz contract provides the staking feature for USDz holders. Specifically, a user can stake their USDz tokens into the sUSDz contract in order to receive the yield that is deposited by the yield manager. The yield manager deposits the full amount of yield when they start the new vesting period.

The sUSDz contract has the `rescueERC20` function, which allows the pool manager to withdraw tokens that may be mistakenly transferred to this contract. To prevent users assets from being withdrawn, there is logic that ensures the deposited asset is not withdrawn through this function:

```
function rescueERC20(IERC20 token, address to, uint256 amount)
    external onlyRole(POOL_MANAGER_ROLE) {
    // If is USDz, check pooled amount first.
    if (address(token) == asset()) {
        require(amount <= token.balanceOf(address(this)).sub(totalAssets()),
    "USDZ_RESCUE_AMOUNT_EXCEED_DEBIT");
    }
    token.safeTransfer(to, amount);
}
```

However, it should be noted that the return value of the `totalAssets` function does not include the unvested amount:

```
function totalAssets() public view override returns (uint256) {
    uint256 unvested = getUnvestedAmount();
    return pooledUSDz.sub(unvested);
}
```

Impact

The pool manager can withdraw the part of yield that is not vested to users through the `rescueERC20` function, despite of the protective logic referenced earlier.

Recommendations

Consider modifying the protection logic of the `rescueERC20` function to ensure that USDz tokens of the amount `pooledUSDz` is left in the contract.

Remediation

This issue has been acknowledged by Anzen Group Ltd., and a fix was implemented in commit `e64b88de` ↗.

3.4. Partially implemented two-step ownership transfer

Target	Canto_childUSDz		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The Canto_childUSDz contract has the variable needed to implement the two-step ownership transfer:

```
// Make owner transfer 2 step.
address private _pendingOwner;
```

However, this variable is unused in this contract, and the functions in the inherited Ownable contract are not overridden.

Impact

This finding documents that the two-step ownership-transfer mechanism is unimplemented in this contract, unlike other contracts, which properly implemented this.

Recommendations

Consider implementing the remaining part of two-step ownership-transfer mechanism.

Remediation

This issue has been acknowledged by Anzen Group Ltd., who notified us that they plan to deploy the childUSDz contract on Canto with the additional line of code that registers the contract for CSR, instead of the Canto_childUSDz contract.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. SPCTPriceOracle does not represent the price of SPCT

During our assessment, we noted that the USDz contract disables the deposit and redemption features when the return value of `SPCTPriceOracle.getPrice()` is lower than 1 USD.

Our concern was that the fair market price of 1 SPCT should be lower than 1 USD, as 1 SPCT corresponds to 1 USD and SPCT also has risks, including but not limited to the possibility of bugs in SPCTPool, the possibility of the failure to repay, and fee for redemption.

Anzen Group Ltd. stated that they plan to integrate the oracle, which returns the backed USD asset per 1 SPCT to SPCTPriceOracle, not the price of 1 SPCT.

4.2. Centralization risk

The project intends to invest the deposited USDC tokens into real-world assets. To facilitate this, the pool manager of SPCTPool can use the `execute` function to transfer USDC tokens from the contract. It is crucial to manage these assets safely and ensure timely and adequate repayment.

The pool manager also has the authority to add or remove users from the whitelist or blacklist. Anzen Group Ltd. has informed us that this feature is required to comply with legal regulations.

We believe that access to privileged accounts should be carefully managed and controlled. Anzen Group Ltd. states that privileged roles will be controlled by a multi-signature wallet.

5. 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.

5.1. Module: SPCTPool.sol

Function: `deposit(uint256 _amount)`

This function is for depositing the USDC to this contract.

Inputs

- `_amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The value must be bigger than zero.
  - **Impact**: Amount to deposit.

Branches and code coverage

Intended branches

- Transfer the exact amount of USDC from the caller to this contract.
  ☑ Test coverage
- The function must mint the exact reduced amount of the SPCT token if any fee exists.
  ☑ Test coverage

Negative behavior

- Revert if the amount was zero.
  ☑ Negative test
- Revert if the caller does not have enough USDC to deposit.
  ☑ Negative test

Function: `redeem(uint256 _amount)`

This function is for redeeming.

Inputs

- `_amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must be bigger than zero.
  - **Impact**: Amount to redeem.

Branches and code coverage

Intended branches

- Transfer the exact amount of USDC from this contract to the caller.
  - ☑ Test coverage
- The function must burn the exact reduced amount of the SPCT token if any fee exists.
  - ☑ Test coverage

Negative behavior

- Revert if the amount was zero.
  - ☑ Negative test

Function: `rescueERC20(IERC20 token, address to, uint256 amount)`

This function is for rescuing the given token.

Inputs

- `token`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Token to rescue.
- `to`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Address to transfer to.
- `amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: None.
  - **Impact**: Amount to transfer.

5.2. Module: USDz.sol

Function: `depositBySPCT(uint256 _amount)`

This function is for depositing SPCT.

Inputs

- `_amount`
    - **Control**: Completely controlled by the caller.
    - **Constraints**: The caller must have enough SPCT to transfer.
    - **Impact**: Amount to deposit.

Branches and code coverage

Intended branches

- Transfer the SPCT to this contract.
    - ☑ Test coverage
- Mint the proper amount of USDz, which is calculated with the fee rates.
    - ☑ Test coverage

Negative behavior

- Revert when the contract is paused.
    - ☑ Negative test
- Revert when the given amount is zero.
    - ☑ Negative test
- Revert when the caller is blacklisted.
    - ☑ Negative test

Function: `deposit(uint256 _amount)`

This function is for depositing USDC.

Inputs

- `_amount`
    - **Control**: Completely controlled by the caller.
    - **Constraints**: The caller must have enough USDC to transfer.
    - **Impact**: Amount to deposit.

Branches and code coverage

Intended branches

- Transfer the USDC to this contract.
  - ☑ Test coverage
- Mint the proper amount of USDz, which is calculated with the fee rates.
  - ☑ Test coverage

Negative behavior

- Revert when the contract is paused.
  - ☑ Negative test
- Revert when the given amount is zero.
  - ☑ Negative test
- Revert when the caller is blacklisted.
  - ☑ Negative test

Function: `redeemBackSPCT(uint256 _amount)`

This function is for redeeming.

Inputs

- `_amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The caller must have enough USDz to burn.
  - **Impact**: Amount to redeem.

Branches and code coverage

Intended branches

- Burn the given amount of the USDz.
  - ☑ Test coverage
- Transfer the proper amount of the SPCT.
  - □ Test coverage

Negative behavior

- Revert when the given amount is zero.
  - ☑ Negative test
- Revert when the caller is blacklisted.
  - ☑ Negative test

Function: `redeem(uint256 _amount)`

This function is for redeeming.

Inputs

- `_amount`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The caller must have enough USDz to burn.
  - **Impact**: Amount to redeem.

Branches and code coverage

Intended branches

- Burn the given amount of the USDz.
  - ☑ Test coverage
- Transfer the proper amount of the USDC.
  - ☑ Test coverage

Negative behavior

- Revert when the reserved USD is smaller than the given amount.
  - ☑ Negative test
- Revert when the given amount is zero.
  - ☑ Negative test
- Revert when the caller is blacklisted.
  - ☑ Negative test

5.3. Module: sUSDz.sol

Function: `CDAssets(uint256 _assets)`

This function is for CDing the assets.

Inputs

- `_assets`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must not be bigger than the value of `maxWithdraw` of the caller.
  - **Impact**: Amount to CD.

Function: `CDShares(uint256 _shares)`

This function is for CDing the shares.

Inputs

- `_shares`
    - **Control**: Completely controlled by the caller.
    - **Constraints**: Must not be bigger than the value of `maxRedeem` of the caller.
    - **Impact**: Amount to CD.

Function: `addYield(uint256 _amount)`

This function adds the asset to this contract. The asset is intended to be USDz.

Inputs

- `_amount`
    - **Control**: Completely controlled by the caller.
    - **Constraints**: Must be bigger than zero.
    - **Impact**: Amount to yield.

Branches and code coverage

Intended branches

- Update the vesting amount as the given amount.
    - ☑ Test coverage
- The given amount must be added to the `pooledUSDz`.
    - ☑ Test coverage
- Transfer the given amount of the asset token to this contract.
    - ☑ Test coverage

Negative behavior

- Revert when the amount is zero.
    - ☑ Negative test
- Revert when the `totalSupply()` is not bigger than one.
    - ☑ Negative test

Function: `redeem(uint256 shares, address receiver, address _owner)`

This function is for redeeming the token by burning the share.

Inputs

- `shares`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The `maxRedeem` of the given `_owner` must not be bigger than the given `shares` input.
  - **Impact**: Amount of shares to redeem.
- `receiver`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must have enough shares to redeem.
  - **Impact**: Address to transfer the token.
- `_owner`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The `maxRedeem` of the given `_owner` must not be bigger than the given `shares` input.
  - **Impact**: Address of the share owner.

Function: `withdraw(uint256 assets, address receiver, address _owner)`

This function is for withdrawing the token by burning the share.

Inputs

- `assets`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The `maxWithdraw` of the given `_owner` must not be bigger than the given `assets` input.
  - **Impact**: Amount to withdraw the token.
- `receiver`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: Must have enough share to withdraw.
  - **Impact**: Address to transfer the withdrawn token.
- `_owner`
  - **Control**: Completely controlled by the caller.
  - **Constraints**: The `maxWithdraw` of the given `_owner` must not be bigger than the given `assets` input.
  - **Impact**: Address of the share owner.

5.4. Module: vault.sol

Function: `deposit(uint256 _amount)`

This function is for depositing the USDC.

Inputs

- `_amount`
    - **Control**: Completely controlled by the caller.
    - **Constraints**: The caller must have enough USDC to transfer.
    - **Impact**: Amount to deposit.

Branches and code coverage

Intended branches

- Transfer the USDC to this contract.
    - ☑ Test coverage
- Transfer the USDz to the caller.
    - ☑ Test coverage

Negative behavior

- Revert when the contract is paused.
    - ☑ Negative test
- Revert when the given amount is zero.
    - ☑ Negative test
- Revert when the calculated amount is smaller than the reserve.
    - ☑ Negative test

6. Assessment Results

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

During our assessment on the scoped Anzen Finance contracts, we discovered four findings. No critical issues were found. Two findings were of medium impact, one was of low impact, and the remaining finding was informational in nature.

6.1. 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.