Prepared for
Ray Huang
River

Prepared by
Sunwoo Hwang
Jaeeu Kim
Zellic

October 10, 2025

Smart Vault

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 River from September 28th to October 2nd, 2025. During this engagement, Zellic reviewed Smart Vault'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:

- Does the system handle differences in underlying token decimals correctly, or could this lead to unexpected behavior?
- Under normal operations, are there any cases where unexpected reverts could occur (e.g., due to overflow/underflow or faulty logic)?
- Are there any attack vectors that could drain user funds or compromise system liquidity?
- Can any user bypass collateralization requirements to mint excess satUSD?

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
- Strategy

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

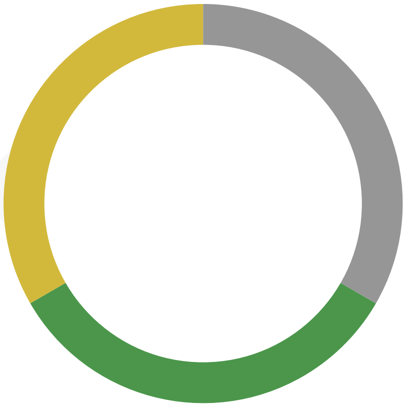
1.4. Results

During our assessment on the scoped Smart Vault contracts, we discovered nine findings. No critical issues were found. Three findings were of medium impact, three were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of River in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	0
🟨 Medium	3
🟩 Low	3
◼ Informational	3



2. Introduction

2.1. About Smart Vault

River contributed the following description of Smart Vault:

> River is building a chain-abstraction stablecoin system that enables cross-chain collateral, yield, and liquidity — all without bridging. Powered by the omni-CDP stablecoin satUSD, users can earn, leverage, and scale across different ecosystems natively. River features the first omni-CDP module that allows users to collateralize assets on Chain A and mint stablecoin satUSD on Chain B — without bridging the assets.

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:

Smart Vault Contracts

Type	Solidity
Platform	EVM-compatible
Target	satoshi-smart-vault
Repository	https://github.com/Satoshi-Protocol/satoshi-smart-vault
Version	`51cc1c0c3942b28ea1b269d2ba0bee8f26bec316`
Programs	`src/core/**.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.4 person-weeks. The assessment was conducted by two consultants over the course of 5 calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Sunwoo Hwang
Engineer
sunwoo@zellic.io ↗

Jaeeu Kim
Engineer
jaeeu@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

September 28, 2025	Kick-off call
September 28, 2025	Start of primary review period
October 2, 2025	End of primary review period

3. Detailed Findings

3.1. Precision loss in reward calculation

Target	SmartVault.sol		
Category	Code Maturity	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

User rewards are calculated using the following formula:

```
reward = (share * interval * rewardRate * REWARD_PER_TOKEN_PRECISION) /
    (totalShares * REWARD_PER_TOKEN_PRECISION)
where REWARD_PER_TOKEN_PRECISION = 1e18
```

In the on-chain contract, `rewardRate` is set to `0.05 * 1e18`. The calculation can suffer from precision loss when the interval between updates is too small. To avoid this loss, the interval should be at least `totalShares * 20 / share`. Otherwise, users may lose rewards due to frequent updates of `_lastUpdateTime`.

Impact

Frequent updates to `_lastUpdateTime` can result in precision loss during reward calculations, potentially leading to users receiving fewer rewards than they should. This is particularly impactful for users with smaller share amounts relative to the total shares.

Recommendations

We recommend implementing a minimum time interval between `_lastUpdateTime` updates to prevent excessive precision loss. Alternatively, we suggest redesigning the reward mechanism so that rewards are only collected at the end of the reward period.

Remediation

TBD

3.2. Missing reward-configuration check

Target	SmartVault.sol		
Category	Code Maturity	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

There are three fields in `RewardConfig`: `rewardRate`, `rewardStartTime`, and `rewardEndTime`. While members of `VaultConfig` are verified in `_checkVaultConfig`, an equivalent validation does not exist for `RewardConfig`. The configuration should be validated to ensure `rewardRate` does not exceed the precision basis and that the end time is not earlier than the start time.

```
function updateRewardConfig(IERC20 rewardToken,
    RewardConfig memory rewardConfig) external onlyManager {
    _updateAllLastRewardPerToken();
    _rewardConfigs[rewardToken] = rewardConfig;

    emit RewardConfigUpdated(rewardToken, rewardConfig);
}
```

Impact

It could lead to overpayment of rewards or potential sanity-check failures in the `RewardConfig`.

Recommendations

Add validation for `rewardRate` and `rewardEndTime` in the `_checkRewardConfig` function.

Remediation

TBD

3.3. Insufficient test coverage

Target	Project-wide		
Category	Protocol Risks	Severity	Medium
Likelihood	N/A	Impact	Medium

Description

When building a complex contract with multiple moving parts (state changes) and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios on every function that touches the contract's state.

Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all contracts, and all functions — not just surface-level functions. It is important to test the invariants required for ensuring security and also verify any mathematical properties from the project's specification.

Unit tests for each individual function are not fully implemented. The tests are written based on scenario-based flows, primarily focusing on setting `vault`, `deposit`, `withdraw`, and `claim` under general vault configurations. The testing did not include any scenarios involving `removeAllocation`.

Impact

Good test coverage has multiple effects:

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in the product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point may seem contradictory given the time investment to create and maintain tests. To expand upon it, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks other code if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence.

Tests have your back here. They are an excellent indicator that the existing functionality was most likely not broken by a change to the code. Without a comprehensive test suite, the above benefits are lost. This increases the likelihood of bugs and vulnerabilities going unnoticed until after deployment, which can be costly and damaging to the project's reputation.

Recommendations

We recommend building a rigorous test suite that includes all contracts to ensure that the system operates securely and as intended.

Remediation

TBD

3.4. Deposit-cap bypass

Target	SmartVault.sol		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `_beforeDeposit` function verifies if the depositor is whitelisted and if the deposit amount is within the deposit cap. However, there are two issues with the current implementation:

1. The `receiver` address is not checked against the whitelist. This allows a depositor to use multiple receiver accounts to bypass the `depositCapPerUser` limit.
2. Even if the `receiver` is whitelisted initially, the `_update`, `claim`, and `withdraw` functions do not verify whitelist status. This means a depositor can transfer their shares to any account to bypass the deposit cap after the initial deposit.

```
if (_isWhitelistMode() && !_isWhitelisted(depositor)) {
    revert NotWhitelisted(depositor);
}

// [...]

if (balanceOf(receiver) + scaledAmount > depositCapPerUser) {
    revert DepositCapPerUserExceeded(amount, _vaultConfig.depositCapPerUser);
}
```

Impact

The deposit-cap mechanism can be bypassed by using multiple accounts, allowing unauthorized users to claim rewards and withdraw funds from the vault.

Recommendations

We recommend the following.

1. Add whitelist verification for the `receiver` address in the `_beforeDeposit` function.

2. Implement whitelist checks in the `_update` function to ensure only whitelisted accounts can interact with the vault.

Remediation

TBD

3.5. User rewards can be lost

Target	SmartVault.sol		
Category	Code Maturity	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `updateRewardTokenList` function removes all existing reward tokens and adds new ones. The `claim` function iterates over the `_rewardTokenList` to get the reward amount for each token, so if a reward token is removed during the reward period, users' rewards will be lost.

```
function updateRewardTokenList(IERC20[] memory rewardTokenList)
    external onlyManager {
    _updateAllLastRewardPerToken();
    delete _rewardTokenList;
    for (uint256 i; i < rewardTokenList.length; i++) {
        IERC20 rewardToken = rewardTokenList[i];
        _checkIsNotZeroAddress(address(rewardToken));
        _rewardTokenList.push(rewardToken);
    }

    emit RewardTokenListUpdated(rewardTokenList);
}
```

Impact

If a reward token is removed during the reward period, users' rewards will be lost.

Recommendations

Add a check to ensure that a reward token is not removed during the reward period.

Remediation

TBD

3.6. Incorrect tracking of staking amount

Target	SmartVault.sol		
Category	Code Maturity	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The staking amount is tracked in the `_stakingAmount` state variable. However, it is not updated when the manager withdraws from the staking vault. Because the contract accounts for staking using a debt token, the actual staked amount can differ from the value stored in `_stakingAmount`.

```
function withdrawFromStakingVault(uint256 amount) external onlyManager {
    smartVaultManager.stakingVault().withdraw(amount,
    address(smartVaultManager), address(this));
}
```

Impact

While this is not a security issue, this can cause inconsistent accounting.

Recommendations

Update `_stakingAmount` whenever the manager withdraws from the staking vault to keep the recorded amount in sync with the actual staked balance.

Remediation

TBD

3.7. Gas optimization

Target	SmartVault.sol		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `executeStrategy` function first increases the strategy's allowance to `type(uint256).max` before calling the strategy contract's `execute` function. After execution, it queries the current allowance and then decreases it by the queried amount.

However, when the allowance is set to `type(uint256).max`, the `transferFrom` function does not decrease the allowance. So the external call to query `underlyingAsset.allowance` is unnecessary, as we know the allowance will still be `type(uint256).max`.

```
function executeStrategy(address _strategy, bytes calldata data)
    external onlyManager {
    _checkIsStrategy(_strategy);
    underlyingAsset.safeIncreaseAllowance(address(_strategy),
    type(uint256).max);
    IStrategy(_strategy).execute(data);
    uint256 allowance = underlyingAsset.allowance(address(this),
    address(_strategy));
    underlyingAsset.safeDecreaseAllowance(address(_strategy), allowance);

    emit StrategyExecuted(_strategy, data);
}
```

Impact

The unnecessary external call to query the allowance consumes additional gas, which could be optimized to reduce transaction costs.

Recommendations

Replace the allowance query and dynamic decrease with a direct decrease of `type(uint256).max`.

Remediation

TBD

3.8. Missing vault check

Target	SmartVaultManager.sol		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The SmartVaultManager contract has functions to manage each vault. So each function checks if the vault address is valid. However, there are some functions that do not check if the vault address is valid.

- `pauseSmartVault`
- `resumeSmartVault`
- `updateVaultConfig`
- `updateStakingVault`
- `setStrategy`
- `setWhitelistMode`
- `updateStakingFactor`
- `updateStakingEnabled`
- `updateRewardConfig`
- `addTokenToRewardList`
- `updateRewardTokenList`

Impact

While this is not a security issue, it could lead to unexpected behavior if the vault address is not valid.

Recommendations

Add a check that the vault address is valid in the functions that do not check if the vault address is valid.

Remediation

TBD

3.9. Reward-token duplication

Target	SmartVault.sol		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The reward-token list is managed with array, which allows duplicate tokens.

```
function addTokenToRewardList(IERC20 rewardToken) external onlyManager {
    _checkIsNotZeroAddress(address(rewardToken));
    _updateAllLastRewardPerToken();
    _rewardTokenList.push(rewardToken);

    emit RewardTokenAdded(rewardToken);
}
```

Impact

While this is not a security issue, it could increase gas costs when iterating over the reward-token list.

Recommendations

Check for duplicate tokens in the `addTokenToRewardList` function.

Remediation

TBD

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. Reentrancy guard modifier

The user functions `deposit`, `withdraw`, and `claim` do not use a reentrancy guard. Although the current implementation is not vulnerable due to the use of ERC-20 tokens, a trusted manager, whitelisted vault addresses, and a safe checks-effects-interactions (CEI) pattern, it is recommended to implement a reentrancy guard for secure protocol operation and future extensibility.

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: RewardVault.sol

Function: `addAllocated(Allocation[] allocations)`

This function is used to add allocated tokens to the reward vault. Only the owner can call this function.

Inputs

- `allocations`
  - **Control**: Fully controllable by the caller (contract owner).
  - **Constraints**: None.
  - **Impact**: Address of asset, address of recipient, and amount of tokens.

Branches and code coverage

Intended branches

- Add the amount to the allocation, for each allocation.
  - ☑ Test coverage
- Transfer the amount to the contract.
  - ☑ Test coverage
- Emit the `AddAllocation` event.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - □ Negative test

Function call analysis

- `SafeERC20.safeTransferFrom(IERC20(allocation.asset), this.owner(), address(this), allocation.amount)`
  - **What is controllable?** `asset` and `amount`.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates the transfer fails.

Function: `removeAllocated(Allocation[] allocations)`

This function is used to remove allocated tokens from the reward vault. Only the owner can call this function.

Inputs

- `allocations`
  - **Control**: Fully controllable by the caller (contract owner).
  - **Constraints**: None.
  - **Impact**: Address of asset, address of recipient, and amount of tokens.

Branches and code coverage

Intended branches

- Check if the balance is enough, for each allocation.
  - □ Test coverage
- Subtract the amount from the allocation.
  - □ Test coverage
- Transfer the amount to the owner.
  - □ Test coverage
- Emit the `RemoveAllocation` event.
  - □ Test coverage

Negative behavior

- Revert if the caller is not the contract owner.
  - □ Negative test
- Revert if the balance is not enough.

- ☐ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(allocation.asset), this.owner(), allocation.amount)`
  - **What is controllable?** `asset`, but allocated by the contract owner, recipient, and amount.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates the transfer fails.

Function: `transferAllocatedTokens(Allocation allocation)`

This function is used to transfer allocated tokens to the recipient. This function is expected to be called by the smart vault, but it could be called directly by the allocation owner.

Inputs

- `allocation`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: Amount is not zero.
  - **Impact**: Address of asset, address of recipient, and amount of tokens.

Branches and code coverage

Intended branches

- Check that the provided amount is not zero.
  - ☑ Test coverage
- Check caller's (`msg.sender`'s) allocation exists.
  - ☑ Test coverage
- Subtract the amount from the caller's allocation.
  - ☑ Test coverage
- Transfer the amount to the recipient.
  - ☑ Test coverage
- Emit the `TransferAllocatedTokens` event.

☑ Test coverage

Negative behavior

- Revert if the amount is zero.
  - □ Negative test
- Revert if the caller's allocation does not exist.
  - □ Negative test

Function call analysis

- `SafeERC20.safeTransfer(IERC20(allocation.asset), allocation.recipient, allocation.amount)`
  - **What is controllable?** `asset`, but allocated by the contract owner, recipient, and amount.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates the transfer fails.

5.2. Module: SmartVaultManager.sol

Function: `claim(ClaimParams claimParams)`

This function is used to claim rewards from a smart vault.

Inputs

- `claimParams`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: Address of the vault must be whitelisted by owner.
  - **Impact**: Addresses of the vault and receiver.

Branches and code coverage

Intended branches

- Check the vault address is valid by calling `_checkVaultIsValid`.
  - ☑ Test coverage
- Call `claim` on the vault.

- ☑ Test coverage
- Emit the `Claim` event.
  - ☑ Test coverage

Negative behavior

- Revert if the contract is paused.
  - □ Negative test
- Revert if the provided vault address is not valid.
  - □ Negative test

Function call analysis

- `vault.claim(msg.sender, receiver)`
  - **What is controllable?** `vault` but whitelisted by owner and receiver.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** The revert indicates a failure in the vault claim, and reentrancy is not an issue since there are no subsequent values affected by the CEI pattern.

Function: `depositERC20(DepositParams depositParams)`

This function is used to deposit ERC-20 tokens into a smart vault.

Inputs

- `depositParams`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: Address of the vault must be whitelisted by owner.
  - **Impact**: Address of the vault, amount to deposit, and the receiver address.

Branches and code coverage

Intended branches

- Check the vault address is valid by calling `_checkVaultIsValid`.
  - ☑ Test coverage
- Call `depositERC20` on the vault.

- ☑ Test coverage
- Emit the `Deposit` event.
  - ☑ Test coverage

Negative behavior

- Revert if the contract is paused.
  - ☐ Negative test
- Revert if the provided vault address is not valid.
  - ☐ Negative test

Function call analysis

- `vault.depositERC20(amount, msg.sender, receiver)`
  - **What is controllable?** `vault` but whitelisted by owner, amount, and receiver.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates a failure in the vault deposit, and reentrancy is not an issue since there are no subsequent values affected by the CEI pattern.

Function: `fetchPriceUnsafe(IERC20 _token)`

This function is used to fetch the price of a token from the oracle without checking the last update time. It is expected to be called from the smart vault to fetch the underlying asset price.

Inputs

- `_token`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: None.
  - **Impact**: Address of the token.

Branches and code coverage

Intended branches

- Call `fetchPriceUnsafe` on the price feed.
  - ☑ Test coverage

- Scale the price to 18 decimals.

  ☑ Test coverage

Function call analysis

- `oracle.priceFeed.fetchPriceUnsafe()`
  - **What is controllable?** `oracle.priceFeed` but whitelisted by the owner.
  - **If the return value is controllable, how is it used and how can it go wrong?** If the price feed fails to return the correct current market price, it would be critical to the vault's collateral calculation.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates a failure in the oracle system, which prevents the vault from calculating the collateral asset value and thus causes it to malfunction.

Function: `fetchPrice(IERC20 _token)`

This function is used to fetch the price of a token from the oracle. It is expected to be called from the smart vault to fetch the underlying asset price.

Inputs

- `_token`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: None.
  - **Impact**: Address of the token.

Branches and code coverage

Intended branches

- Call `fetchPrice` on the price feed.

  ☑ Test coverage
- Scale the price to 18 decimals.

  ☑ Test coverage

Function call analysis

- `oracle.priceFeed.fetchPrice()`

- **What is controllable?** `oracle.priceFeed` but whitelisted by the owner.
- **If the return value is controllable, how is it used and how can it go wrong?** If the price feed fails to return the correct current market price, it would be critical to the vault's collateral calculation.
- **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates a failure in the oracle system, which prevents the vault from calculating the collateral asset value and thus causes it to malfunction.

Function: `manageDebtAndStake(ISmartVault vault)`

This function is used to manage debt and stake for a smart vault.

Inputs

- `vault`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: Address of the vault must be whitelisted by the owner.
  - **Impact**: Address of the vault.

Branches and code coverage

Intended branches

- Check the vault address is valid by calling `_checkVaultIsValid`.
  - ☑ Test coverage
- Call `manageDebtAndStake` on the vault.
  - ☑ Test coverage
- Emit the `ManageDebtAndStake` event.
  - ☑ Test coverage

Negative behavior

- Revert if the contract is paused.
  - □ Negative test
- Revert if the provided vault address is not valid.
  - □ Negative test

Function call analysis

- `vault.manageDebtAndStake()`
  - **What is controllable?** `vault` but whitelisted by the owner.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** The revert indicates a failure in the vault's manage debt and stake, and reentrancy is not an issue since there are no subsequent values affected by the CEI pattern.

Function: `transferCallback(IERC20 token, address from, uint256 amount)`

This function is used to transfer ERC-20 tokens to smart vault. It is expected to be called from the smart vault during `_depositERC20`.

Inputs

- `token`
  - **Control**: Not controllable, declared `underlyingAsset`.
  - **Constraints**: None.
  - **Impact**: Address of the token.
- `from`
  - **Control**: Not controllable,
  - **Constraints**: Should be the address of the depositor (caller of `depositERC20`).
  - **Impact**: Address of the depositor.
- `amount`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: Should be approved by the depositor.
  - **Impact**: Amount of token.

Branches and code coverage

Intended branches

- Check if the caller is the smart vault.
  - ☑ Test coverage
- Call `safeTransferFrom` on the token.
  - ☑ Test coverage

- Check if the balance of the token is changed correctly.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the smart vault.
  - □ Negative test
- Revert if the balance of the token is changed unexpectedly.
  - □ Negative test

Function call analysis

- `token.balanceOf(msg.sender)`
  - **What is controllable?** `token`, but `underlyingAsset` is declared in the vault.
  - **If the return value is controllable, how is it used and how can it go wrong?** Even without an actual token transfer, the callback can still succeed.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates that the token address is not valid ERC-20.
- `SafeERC20.safeTransferFrom(token, from, msg.sender, amount)`
  - **What is controllable?** `token`, but `underlyingAsset` is declared in the vault.
  - **If the return value is controllable, how is it used and how can it go wrong?** Even without an actual token transfer, the callback can still succeed.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates that the token is not a valid ERC-20 or that the sender lacks sufficient balance or approval.
- `token.balanceOf(msg.sender)`
  - **What is controllable?** `token`, but `underlyingAsset` is declared in the vault.
  - **If the return value is controllable, how is it used and how can it go wrong?** Even without an actual token transfer, the callback can still succeed.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates that the token address is not valid ERC-20.

Function: `vaultBurnDebtTokenCallback(uint256 amount)`

This function is used to burn debt tokens for a smart vault. It is expected to be called from the smart vault during `_burnDebtToken`.

Inputs

- `amount`

- **Control**: Not controllable, calculated in the contract.
- **Constraints**: Amount is not zero (validated in the `_burnDebtToken`).
- **Impact**: Amount of debt token to burn.

Branches and code coverage

Intended branches

- Check if the caller is the smart vault.
  - ☑ Test coverage
- Call `burn` on the debt token.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the smart vault.
  - □ Negative test
- Revert if the debt-token burning fails.
  - □ Negative test

Function call analysis

- `this.debtToken.burn(msg.sender, amount)`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** It is safe because `_burnDebtToken`, which calls `vaultBurnDebtTokenCallback`, performs a balance check after burning.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates a failure in the burn process, and since the `debtToken` is expected to be an ERC-20 address, there is no reentrancy risk.

Function: `vaultMintDebtTokenCallback(uint256 amount)`

This function is used to mint debt tokens for a smart vault. It is expected to be called from the smart vault during `_mintDebtToken`.

Inputs

- `amount`
  - **Control**: Not controllable, calculated in the contract.

- **Constraints**: Amount is not zero (validated in the `_mintDebtToken`).
- **Impact**: Amount of debt token to mint.

Branches and code coverage

Intended branches

- Check if the caller is the smart vault.
  - ☑ Test coverage
- Call `mint` on the debt token.
  - ☑ Test coverage

Negative behavior

- Revert if the caller is not the smart vault.
  - □ Negative test
- Revert if the debt-token minting fails.
  - □ Negative test

Function call analysis

- `this.debtToken.mint(msg.sender, amount)`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** It is safe because `_mintDebtToken`, which calls `vaultMintDebtTokenCallback`, performs a balance check after minting.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates a failure in the mint process, and since the `debtToken` is expected to be an ERC-20 address, there is no reentrancy risk.

Function: `withdraw(WithdrawParams withdrawParams)`

This function is used to withdraw underlying asset from a smart vault.

Inputs

- `withdrawParams`
  - **Control**: Fully controllable by the caller.
  - **Constraints**: Address of the vault must be whitelisted by the owner.

- **Impact**: Address of the vault, amount to withdraw, and the receiver address.

Branches and code coverage

Intended branches

- Check the vault address is valid by calling `_checkVaultIsValid`.
  - ☑ Test coverage
- Call `withdraw` on the vault.
  - ☑ Test coverage
- Emit the `Withdraw` event.
  - ☑ Test coverage

Negative behavior

- Revert if the contract is paused.
  - □ Negative test
- Revert if the provided vault address is not valid.
  - □ Negative test

Function call analysis

- `vault.withdraw(amount, msg.sender, receiver)`
  - **What is controllable?** `vault` but whitelisted by owner, amount, and receiver.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** The revert indicates a failure in the vault withdrawal, and reentrancy is not an issue since there are no subsequent values affected by the CEI pattern.

5.3. Module: SmartVault.sol

Function: `claim(address owner, address receiver)`

This function is used to claim rewards from the smart vault. It is expected to be called from the smart vault manager.

Inputs

- `owner`

    - **Control**: Fully controllable by the caller (`smartVaultManager`).
    - **Constraints**: None.
    - **Impact**: Address of the share owner.
- `receiver`
    - **Control**: Fully controllable by the caller (`smartVaultManager`).
    - **Constraints**: None.
    - **Impact**: Address of the receiver.

Branches and code coverage

Intended branches

- Check if the current timestamp is within the claim time range.
    - ☑ Test coverage
- Update the reward for the owner.
    - ☑ Test coverage
- Update the pending rewards and transfer the rewards to the receiver, for each reward token, if the pending rewards is not zero.
    - ☑ Test coverage
- Call the `transferAllocatedTokens` function of the reward vault to transfer the rewards to the receiver.
    - ☑ Test coverage
- Emit the `Claimed` event.
    - ☑ Test coverage
- Invoke the `_manageDebtAndStake` function.
    - ☑ Test coverage
- Skip when staking is disabled.
    - ☑ Test coverage
- Calculate the underlying token value using the manager's `fetchPrice` function.
    - ☑ Test coverage
- Calculate the target staking amount using the staking factor.
    - ☑ Test coverage
    - If the target is greater than the minted, mint the debt token and adjust the staking amount.
        - ☑ Test coverage
    - If the target is less than the minted, burn the debt token and adjust the

staking amount.

☑ Test coverage

Negative behavior

- Revert if the caller is not the manager.
  - □ Negative test
- Revert if the contract is paused.
  - □ Negative test
- Revert if the amount is zero.
  - □ Negative test
- Revert if the pending rewards are not zero.
  - □ Negative test
- Revert if minting/burning the debt token fails.
  - □ Negative test

Function call analysis

- `this._claim(owner, receiver) -> this.rewardVault.transferAllocatedTokens(allocation)`
  - **What is controllable?** `allocation` is set by the owner and receiver.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates no allocation for the receiver in the reward vault or when the reward vault does not have a sufficient balance.
- Other external call flows are the same as the `depositERC20` function's `_manageDebtAndStake` flow of `SmartVault`.

Function: `depositERC20(uint256 amount, address depositor, address receiver)`

This function is used to deposit ERC-20 tokens to the smart vault. It is expected to be called from the smart vault manager.

Inputs

- `amount`

    - **Control**: Fully controllable by the caller (`smartVaultManager`).
    - **Constraints**: Amount is not zero and is within the deposit cap.
    - **Impact**: Amount of token.
- `depositor`
    - **Control**: Fully controllable by the caller (`smartVaultManager`).
    - **Constraints**: Should be whitelisted when whitelist mode is enabled.
    - **Impact**: Address of the depositor.
- `receiver`
    - **Control**: Fully controllable by the caller (`smartVaultManager`).
    - **Constraints**: None.
    - **Impact**: Address of the share receiver.

Branches and code coverage

Intended branches

- Invoke the `_beforeDeposit` function.
    - ☑ Test coverage
- Check if the current timestamp is within the deposit time range.
    - ☑ Test coverage
- Check if the depositor is whitelisted when whitelist mode is enabled.
    - ☑ Test coverage
- Check if the amount is within the deposit cap.
    - ☑ Test coverage
- Check if the amount is within the deposit cap per user.
    - ☑ Test coverage
- Update the reward for the receiver.
    - ☑ Test coverage
- Invoke the `_depositERC20` function.
    - ☑ Test coverage
- Call the `transferCallback` function of `smartVaultManager` to get the underlying asset amount.
    - ☑ Test coverage
- Update the total deposited underlying asset amount.
    - ☑ Test coverage
- Update the shares for the receiver.

  - ☑ Test coverage
- Emit the `Deposit` event.
  - ☑ Test coverage
- Invoke the `_manageDebtAndStake` function.
  - ☑ Test coverage
- Calculate the underlying token value using the manager's `fetchPrice` function.
  - ☑ Test coverage
- Calculate the target staking amount using the staking factor.
  - ☑ Test coverage
  - If the target is greater than the minted, mint the debt token and adjust the staking amount.
    - ☑ Test coverage
  - If the target is less than the minted, burn the debt token and adjust the staking amount.
    - ☑ Test coverage

Negative behavior

- Revert if the caller is not the manager.
  - □ Negative test
- Revert if the contract is paused.
  - □ Negative test
- Revert if the amount is zero.
  - □ Negative test
- Revert if the depositor is not whitelisted when whitelist mode is enabled.
  - □ Negative test
- Revert if the amount is within the deposit cap.
  - □ Negative test
- Revert if the amount is within the deposit cap per user.
  - □ Negative test
- Revert if minting/burning debt token fails.
  - □ Negative test

Function call analysis

- `this._depositERC20(amount, depositor, receiver) -> smartVaultManager.transferCallback(underlyingAsset, depositor, amount)`
    - **What is controllable?** `depositor` and `amount`.
    - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
    - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates the transfer failed.
- `this._manageDebtAndStake() -> this._calculateUnderlyingTokenValue() -> this._getAssetPrice() -> this.smartVaultManager.fetchPrice(this.underlyingAsset)`
    - **What is controllable?** Nothing.
    - **If the return value is controllable, how is it used and how can it go wrong?** If the price feed fails to return the correct current market price, it would be critical to the vault's collateral calculation.
    - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates a failure in the oracle system, which prevents the vault from calculating the collateral asset value and thus causes it to malfunction.
- `this._manageDebtAndStake() -> this._mintDebtToken(mintAmount) -> this.smartVaultManager.vaultMintDebtTokenCallback(amount)`
    - **What is controllable?** `amount`.
    - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
    - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates the debt-token minting failed.
- `this._manageDebtAndStake() -> this._mintDebtToken(mintAmount) -> this.smartVaultManager.debtToken().balanceOf(address(this))`
    - **What is controllable?** Nothing.
    - **If the return value is controllable, how is it used and how can it go wrong?** Even without an actual token minting, the callback can still succeed.
    - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates the debt token is not a valid ERC-20 token.
- `this._manageDebtAndStake() -> this._burnDebtToken(mintAmount) -> this.smartVaultManager.vaultBurnDebtTokenCallback(amount)`
    - **What is controllable?** `amount`.
    - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
    - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates the debt-token burning failed.
- `this._manageDebtAndStake() -> this._burnDebtToken(mintAmount) ->`

`this.smartVaultManager.debtToken().balanceOf(address(this))`

  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** Even without an actual token burning, the callback can still succeed.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates the debt token is not a valid ERC-20 token.

- `this._manageDebtAndStake() -> this._adjustStaking(target) -> this._stakeDebtToken(stakeAmount) -> IStakingVault(stakingVault).deposit(amount, address(this))`
  - **What is controllable?** `stakingVault` is set by `smartVaultManager` and `amount` but calculated by staking factor and the price of underlying asset.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates the staking failed.
- `this._manageDebtAndStake() -> this._adjustStaking(target) -> this._stakeDebtToken(stakeAmount) -> IStakingVault(smartVaultManager.stakingVault().withdraw(amount, address(this), address(this))`
  - **What is controllable?** `stakingVault` set by `smartVaultManager` and `amount` but calculated by staking factor and the price of underlying asset.
  - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates the unstaking failed.

Function: `manageDebtAndStake()`

This function is used to manage debt and stake of the smart vault. It is expected to be called from the smart vault manager.

Branches and code coverage

Intended branches

- Invoke the `_manageDebtAndStake` function.
  - ☑ Test coverage
- Skip when staking is disabled.
  - ☑ Test coverage
- Calculate the underlying token value using the manager's `fetchPrice` function.

- ☑ Test coverage
- Calculate the target staking amount using the staking factor.
    - ☑ Test coverage
    - If the target is greater than the minted, mint the debt token and adjust the staking amount.
        - ☑ Test coverage
    - If the target is less than the minted, burn the debt token and adjust the staking amount.
        - ☑ Test coverage

Negative behavior

- Revert if caller is not the manager.
    - □ Negative test
- Revert if the contract is paused.
    - □ Negative test
- Revert if minting/burning debt token fails.
    - □ Negative test

Function call analysis

- The external call flows are the same as the `depositERC20` function's `_manageDebtAndStake` flow of `SmartVault`.

Function: `withdraw(uint256 amount, address owner, address receiver)`

This function is used to withdraw underlying assets from the smart vault. It is expected to be called from the smart vault manager.

Inputs

- `amount`
    - **Control**: Fully controllable by the caller (`smartVaultManager`).
    - **Constraints**: Amount should be less than the shares of the owner.
    - **Impact**: Amount to withdraw.
- `owner`
    - **Control**: Fully controllable by the caller (`smartVaultManager`).
    - **Constraints**: None.

    - **Impact**: Address of the share owner.
- `receiver`
    - **Control**: Fully controllable by the caller (`smartVaultManager`).
    - **Constraints**: None.
    - **Impact**: Address of the receiver.

Branches and code coverage

Intended branches

- Invoke the `_beforeWithdraw` function.
    ☑ Test coverage
- Check if the current timestamp is within the withdraw time range.
    ☑ Test coverage
- Check if the amount is within the shares of the owner.
    ☑ Test coverage
- Invoke the `_withdraw` function.
    ☑ Test coverage
- Update the shares for the receiver.
    ☑ Test coverage
- Call the `safeTransfer` function of the token to the receiver.
    ☑ Test coverage
- Update the total deposited underlying asset amount.
    ☑ Test coverage
- Emit the `Withdraw` event.
    ☑ Test coverage
- Invoke the `_manageDebtAndStake` function.
    ☑ Test coverage
- Skip when staking is disabled.
    ☑ Test coverage
- Calculate the underlying token value using the manager's `fetchPrice` function.
    ☑ Test coverage
- Calculate the target staking amount using the staking factor.
    ☑ Test coverage
    - If the target is greater than the minted, mint the debt token and adjust the

staking amount.

    ☑ Test coverage

- If the target is less than the minted, burn the debt token and adjust the staking amount.

    ☑ Test coverage

Negative behavior

- Revert if the caller is not the manager.
    - □ Negative test
- Revert if the contract is paused.
    - □ Negative test
- Revert if the amount is zero.
    - □ Negative test
- Revert if the amount is greater than the shares of the owner.
    - □ Negative test
- Revert if minting/burning debt token fails.
    - □ Negative test

Function call analysis

- `this._withdraw(amount, owner, receiver) -> SafeERC20.safeTransfer(this.underlyingAsset, receiver, amount)`
    - **What is controllable?** `receiver` and `amount`.
    - **If the return value is controllable, how is it used and how can it go wrong?** The return value is not used.
    - **What happens if it reverts, reenters or does other unusual control flow?** A revert indicates the transfer failed or the contract does not have enough underlying asset.
- Other external call flows are the same as the `depositERC20` function's `_manageDebtAndStake` flow of `SmartVault`

6. Assessment Results

During our assessment on the scoped Smart Vault contracts, we discovered nine findings. No critical issues were found. Three findings were of medium impact, three were of low impact, and the remaining findings were 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.