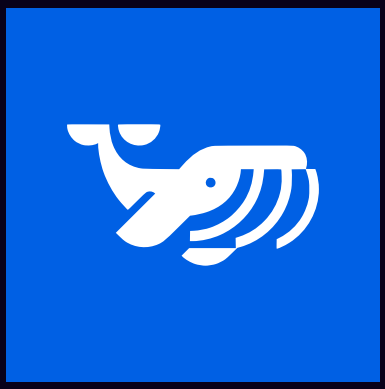

SMART CONTRACTS REVIEW

zokyo

February 17th 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING WHALES MONEY

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 3 High issues: 1 resolved and 2 acknowledged = - 6 points deducted
- 2 Medium issues: 2 acknowledged = - 4 points deducted
- 2 Low issues: 2 resolved = 0 points deducted
- 0 Informational issues: 0 points deducted

Thus, 100 - 6 - 4 = 90

TECHNICAL SUMMARY

This document outlines the overall security of the Whales Money smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Whales Money smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Whales Money team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Whales Money repository:
Repo: https://github.com/Nava-Labs/whales-money-contracts/commit/f4dc958f8276aef388f2a9930907a52470d161ba

Last commit - `0d00ca007a6db44e3a2493ddce7340793c76b6e4`

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./core/SPCTPool.sol
- ./core/wUSDFlat.sol
- ./core/RewardDistributor.sol
- ./core/oracle/swUSDwUSDExchangeRateChainlinkAdapter.sol
- ./core/oracle/SPCTPriceOracle.sol
- ./core/swUSDLayerZeroAdapter/WhalesMoneyLayerZeroAdapter.sol
- ./core/swUSDLayerZeroAdapter/swUSDOFTAdapter.sol
- ./core/swUSDLayerZeroAdapter/OFTExternal.sol
- ./core/Forwarder.sol
- ./core/wUSD.sol
- ./core/swUSD.sol
- ./core/child/ChildwUSD.sol
- ./core/child/ChildswUSD.sol
- ./interfaces/IERC20MintableBurnable.sol
- ./interfaces/MinimalAggregatorV3Interface.sol
- ./interfaces/IProxy.sol
- ./interfaces/IBridgeToken.sol
- ./interfaces/ISPCTPriceOracle.sol
- ./interfaces/IBaseTokenManager.sol
- ./interfaces/ISPCTPool.sol
- ./interfaces/IswUSD.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Whales Money smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.
02	Cross-comparison with other, similar smart contract/s by industry leaders.
03	Thorough manual review of the codebase line by line.

Executive Summary

The core functionality revolves around USDC-backed WUSD stablecoins, managed through a treasury-centric approach where USDC collateral is directly handled by a multisig treasury via deposit() and cdRedeem() functions. The system operates on a dual-token model: WUSD as the base stablecoin and sWUSD as its yield-bearing variant, with yield distribution managed through the addYield() function.

The protocol's cross-chain architecture is built on LayerZero's omnichain infrastructure, utilizing specialized adapters (swUSDOFTAdapter, OFTExternal) and child contracts for seamless operations across networks. The main components include a Forwarder contract (depositFor(), depositBySwap()) for multi-token deposits, the core WUSD and sWUSD contracts on Ethereum mainnet, and their corresponding child versions (ChildwUSD and ChildswUSD) on other chains. Security features are implemented through cooldown periods (CDPeriod), fee structures (mintFeeRate, redeemFeeRate) denominated in USDC, and a price oracle system (SPCTPriceOracle.getPrice()).

The protocol incorporates access control mechanisms through OpenZeppelin's AccessControl and Ownable patterns, with specialized roles like POOL_MANAGER_ROLE and YIELD_MANAGER_ROLE. DEX integration is achieved through Paraswap, enabling efficient token swaps via low-level calls in the Forwarder contract. The ERC4626 vault standard is implemented for the yield-bearing token (sWUSD), providing standardized vault functionality through deposit(), withdraw(), and redeem() functions. This comprehensive architecture ensures efficient cross-chain liquidity while maintaining security through controlled redemption processes (_userCD mapping), multi-signature governance, and standardized interfaces for future extensibility.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Whales Money team and the Whales Money team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	SPCT Tokens Not Minted In redeemBackSPCT	High	Resolved
2	setPrice() in SPCTPriceOracle lacks access control modifier	High	Acknowledged
3	redeem() from wUSD is likely to fail in most cases	High	Acknowledged
4	Collateral Rate Is Not Applied If Deviates From 1:1 Ratio	Medium	Acknowledged
5	Centralisation Risks	Medium	Acknowledged
6	Missing zero address checks in wUSDFlat's constructor	Low	Resolved
7	Multiple failing tests and assertions in wusd.t.sol	Low	Resolved

HIGH-1 | RESOLVED

SPCT Tokens Not Minted In redeemBackSPCT

The **redeemBackSPCT()** function burns **wUSD** tokens of the user but does not mint/transfer the corresponding **SPCT** tokens back to the user, therefore when a user wants to redeem his **wUSD** for **SPCT** , his **wUSD** would be burnt but he won't receive any **SPCT** back.

Recommendation:
Add IERC20(address(spct)).safeTransfer(msg.sender, _amount); after the burn operation.

HIGH-2 | ACKNOWLEDGED

setPrice() in SPCTPriceOracle lacks access control modifier

In contract SPCTPriceOracle.sol, the **setPrice()** function which is used to set the etherPrice variable, can be called by anyone as it lacks any access control modifier. This can result in any unauthorized actors from calling this function to setPrice.

```
function setPrice(uint256 newEtherPrice) external {
    etherPrice = newEtherPrice;
}
```

Recommendation:
It is advised to add the appropriate access control modifiers and disallow unauthorized users from calling this function.

Client's Comment: The function setPrice() will not be used in production, it is only for testing purposes.

redeem() from wUSD is likely to fail in most cases

In contract **wUSD.sol**, the **redeem()** is likely to fail in most cases. This is because the contract does not hold wUSD when **deposit()** is being called. This is because the deposited USDC tokens of users are directly transferred to the treasury (see line: 164 & 179):

Line: 164 _transferToTreasury(_amount);
Line: 179 _transferToTreasury(amountAfterFee);

Apparently, when **redeem()** function is called by the user, the following transfer takes place in the **redeem()** function, but is likely to fail as the contract does not hold any usdc tokens:

```
IERC20(usdc).safeTransfer(msg.sender, amountToWithdraw);

// transfer fee
IERC20(usdc).safeTransfer(feeRecipient, amountFeeToWithdraw);
```

This can be risky for users as they would not be able to redeem their tokens even after their staking time is complete. And they would need to rely on the project to manually transfer tokens to this contract. Instead of the system being trustless.

Recommendation:
It is advised to keep the user's tokens in the contract and avoid sending their tokens to the treasury, where the user's funds will be inaccessible to them. Ensuring that there are no cases when redeem() fails due to insufficient funds in the contract.

Client's comment: Yes, i acknowledge this issue, but we still want to send the USDC directly to our treasury, this is part of our business process, and there will be a mechanism to fix this issue

Collateral Rate Is Not Applied If Deviates From 1:1 Ratio

The **_checkCollateralRate()** function in **wUSD** contract currently performs a comparison between oracle price and collateral rate that always evaluates to true. This occurs because oracle.getPrice() / 1e18 >= collateralRate always returns true (1 >= 1)

This nullifies the entire purpose of having a collateral rate check, potentially allowing operations to proceed when they should be blocked due to insufficient collateral backing.

Recommendation:
Modify oracle to return real SPCT/USD price data.

Client's comment: I acknowledge this as well, but I prefer to leave it as is, following the exact implementation used by Anzen.

Centralisation Risks

a.) In the SPCTPool.sol contract, the **POOL_MANAGER_ROLE** can use the **redeemByFiat()** function to burn tokens from any account. A compromised or malicious **POOL_MANAGER_ROLE** can use this function to burn tokens from any account resulting in loss of spct tokens and value.
Client's Comment: Yes we will use multisig for POOL_MANAGER_ROLE

```
  function redeemByFiat(address _user, uint256 _amount) external whenNotPaused
onlyRole(POOL_MANAGER_ROLE) {
        require(_amount > 0, "REDEEM_AMOUNT_IS_ZERO");
        _burn(_user, _amount);
        emit Redeem(msg.sender, _amount);
    }
```

b.) In the wUSD.sol contract, if a user deposits usdc, but then gets blacklisted by the **POOL_MANAGER_ROLE**, then the user won't be able to redeem his deposited funds. This could be troublesome for valid depositors who are unable to withdraw if they get blacklisted in case the blacklist functionality is misused by a compromised or malicious **POOL_MANAGER_ROLE**.

Recommendation:
It is again advised to ensure that a multisig is used for **POOL_MANAGER_ROLE** of at least 3/5 configuration (or the admin of the contract assigned this role)

Client's comment: Yes of course we will use multisig for POOL_MANAGER_ROLE

Missing zero address checks in wUSDFlat's constructor

There are missing zero address checks in constructor parameters of the wUSDFlat constructor parameters. This is can lead to accidentally setting the **swusd** and **wusd** variables as zero address. Once set, these variables cannot be changed.

```
constructor(address _swusd, IERC20 _wusd) {
        swusd = _swusd;
        wusd = _wusd;
    }
```

The same goes for the constructor parameters **_usdc** & **_spct** of wUSD.sol and constructor parameter **_asset** of swUSD.sol, and **_usdc** & **_wusd** of Forwarder.sol

Recommendation:

It is advised to add appropriate zero address checks such as below

```
require( _swusd != address(0));
require( _wusd != address(0));
```

Multiple failing tests and assertions in wusd.t.sol

There are multiple tests failing in the test suite provided by the team. This can lead to serious issues if the assumptions made during testing and coding are different. Failed tests can lead to failed uncovering of hidden bugs and issues of much higher severities. Due to the time and scope of this audit, the team was able to only manual review during the allotted timeline. The failed tests and assertions can be seen as follows:
forge test --via-ir
[⠒] Compiling...
No files changed, compilation skipped

Ran 5 tests for test/wusd.t.sol:WUSDTest
[FAIL: assertion failed: 10000000000000000000 != 0]
testSpctDepositByFiatAndwUSDDepositBySPCT() (gas: 259843)
[PASS] testStakingSuccess() (gas: 550678)
[FAIL: assertion failed: 10000000000000000000 !~= 11000000000000000000 (max delta: 1, real delta: 10000000000000000000)] testStakingSuccessAndYield() (gas: 652835)
[FAIL: revert: ERC20: transfer to the zero address] testwUSDMintRedeemSuccess() (gas: 713301)
[FAIL: assertion failed: 0 != 10000000000000000] testwUSDMintRedeemWithwUSDFee() (gas: 366576)
Suite result: FAILED. 1 passed; 4 failed; 0 skipped; finished in 9.36s (12.37ms CPU time)

Ran 1 test suite in 9.37s (9.36s CPU time): 1 tests passed, 4 failed, 0 skipped (5 total tests)

Failing tests:
Encountered 4 failing tests in test/wusd.t.sol:WUSDTest
[FAIL: assertion failed: 10000000000000000000 != 0]
testSpctDepositByFiatAndwUSDDepositBySPCT() (gas: 259843)
[FAIL: assertion failed: 10000000000000000000 !~= 11000000000000000000 (max delta: 1, real delta: 10000000000000000000)] testStakingSuccessAndYield() (gas: 652835)

[FAIL: revert: ERC20: transfer to the zero address] testwUSDMintRedeemSuccess() (gas: 713301)
[FAIL: assertion failed: 0 != 10000000000000000] testwUSDMintRedeemWithwUSDFee() (gas: 366576)

Encountered a total of 4 failing tests, 1 tests succeeded

Recommendation:
It is advised to fix the failing tests and do thorough testing to ensure that the code is battle tested before going into production.

	./core/SPCTPool.sol **./core/wUSDFlat.sol** **./core/RewardDistributor.sol** **./core/oracle/swUSDwUSDExchangeRateChainlinkAdapter.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./core/oracle/SPCTPriceOracle.sol **./core/swUSDLayerZeroAdapter/WhalesMoneyLayerZeroAdapter.sol** **./core/swUSDLayerZeroAdapter/swUSDOFTAdapter.sol** **./core/swUSDLayerZeroAdapter/OFTExternal.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./core/Forwarder.sol ./core/wUSD.sol ./core/swUSD.sol ./core/child/ChildwUSD.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./core/child/ChildswUSD.sol ./interfaces/IERC20MintableBurnable.sol ./interfaces/MinimalAggregatorV3Interface.sol ./interfaces/IProxy.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./interfaces/IBridgeToken.sol **./interfaces/ISPCTPriceOracle.sol** **./interfaces/IBaseTokenManager.sol** **./interfaces/ISPCTPool.sol** **./interfaces/IswUSD.sol**
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Whales Money team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Whales Money team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo