AirPuff

SMART CONTRACT AUDIT

zokyo

March 27th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passed a security audit.

ZOKYO AUDIT SCORING AIRPUFF

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.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 2 High issues: 2 verified = 2 points deducted for the concern regarding delegating security out of the contract
- 0 Medium issues: 0 points deducted
- 1 Low issues: 1 resolved = 0 points deducted
- 6 Informational issues: 2 resolved, 1 verified, 3 acknowledged = 2 points deducted for the best practices and styleguide violation

Thus, 100 - 2 = 96 points out of 100.

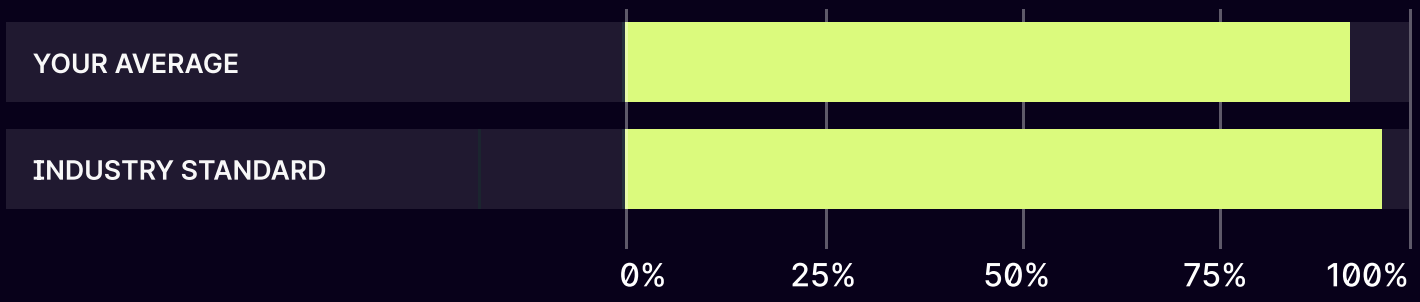
TECHNICAL SUMMARY

This document outlines the overall security of the AirPuff smart contracts evaluated by the Zokyo Security team.

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

Contract Status

Testable Code



Corresponds to the industry standards.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract 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 AirPuff team put in place a bug bounty program to encourage further active analysis of the smart contract.

Table of Contents

Auditing Strategy and Techniques Applied 6

Executive Summary 7

Protocol overview 8

Protocol Description 18

Roles and responsibilities 19

Settings 21

Deployment 22

Structure and Organization of Document 24

Complete Analysis 25

Code Coverage and Test Results for all files written by Zokyo Security 33

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the AirPuff repository:
https://github.com/Airpuff/AirPuff-Contracts/tree/main
Branch: main
Initial commit: e9084a8b0ae681a608aa893108b02c7febd6118a
Final commit: ab311c3a941e8a88e72fe92dd009d0717b003773

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 AirPuff smart contract. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Hardhat testing framework. 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.	03	Testing contract logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contracts by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo Security team has conducted a comprehensive AirPuff protocol audit covering all aspects, including lending, handling, and vault contracts, as part of the initial audit iteration.

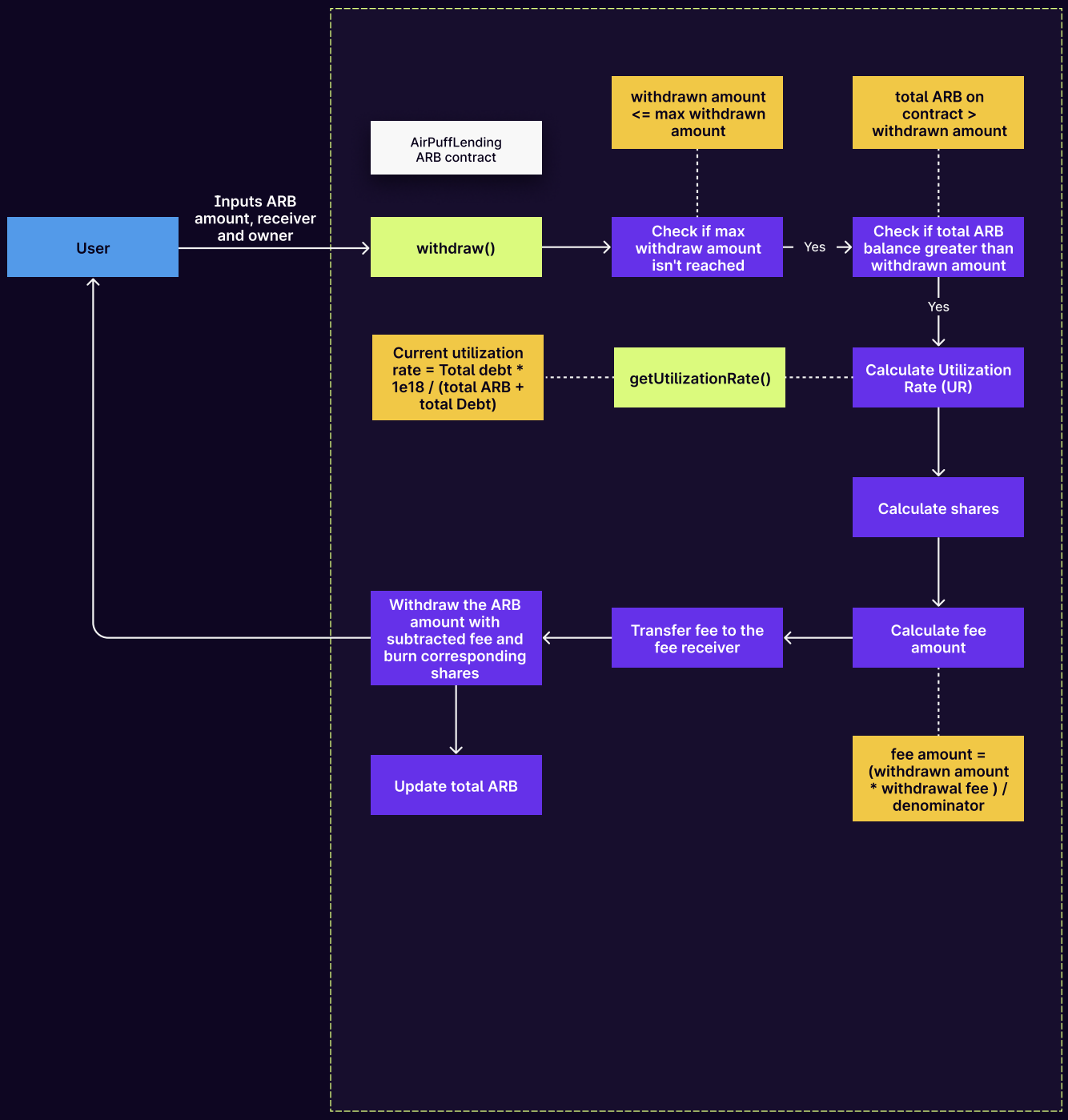
The audit's goal was to verify the correctness of vault implementation, the connection between contracts, all main functionality (lend, repay, open and close position), integration of Kyberswap, and to perform the check that the tokens on the contract were safe. The auditors reviewed the code line by line, compared it against several vulnerable areas and checklists of vulnerabilities (including the standard one), validated the business logic of the contracts, and ensured that best practices in terms of gas spending were applied. The setup and deployment scripts of the contracts were also carefully audited, although contracts were already deployed on the Arbitrum network.

During the manual portion of the audit, two high-risk issues were discovered regarding debt repayment manipulation and possible front-run attack. The AirPuff team verified both issues, explaining that the debt amount will always equal the borrowed amount due to protocol integration on top of the Kyberswap lending vaults. Although delegating some part of the security (e.g., front-run protection) to a proven 3rd party is acceptable, the security team highly recommends monitoring this functionality. Other low-level and informational severity issues were related to the withdrawal fee, which can be set to 100%, code styling, unused variables, lack of validation, lack of events emitting, custom errors, and unused lock time functionality. The Defactor team resolved some of these issues, though most of the issues were marked as acknowledged.

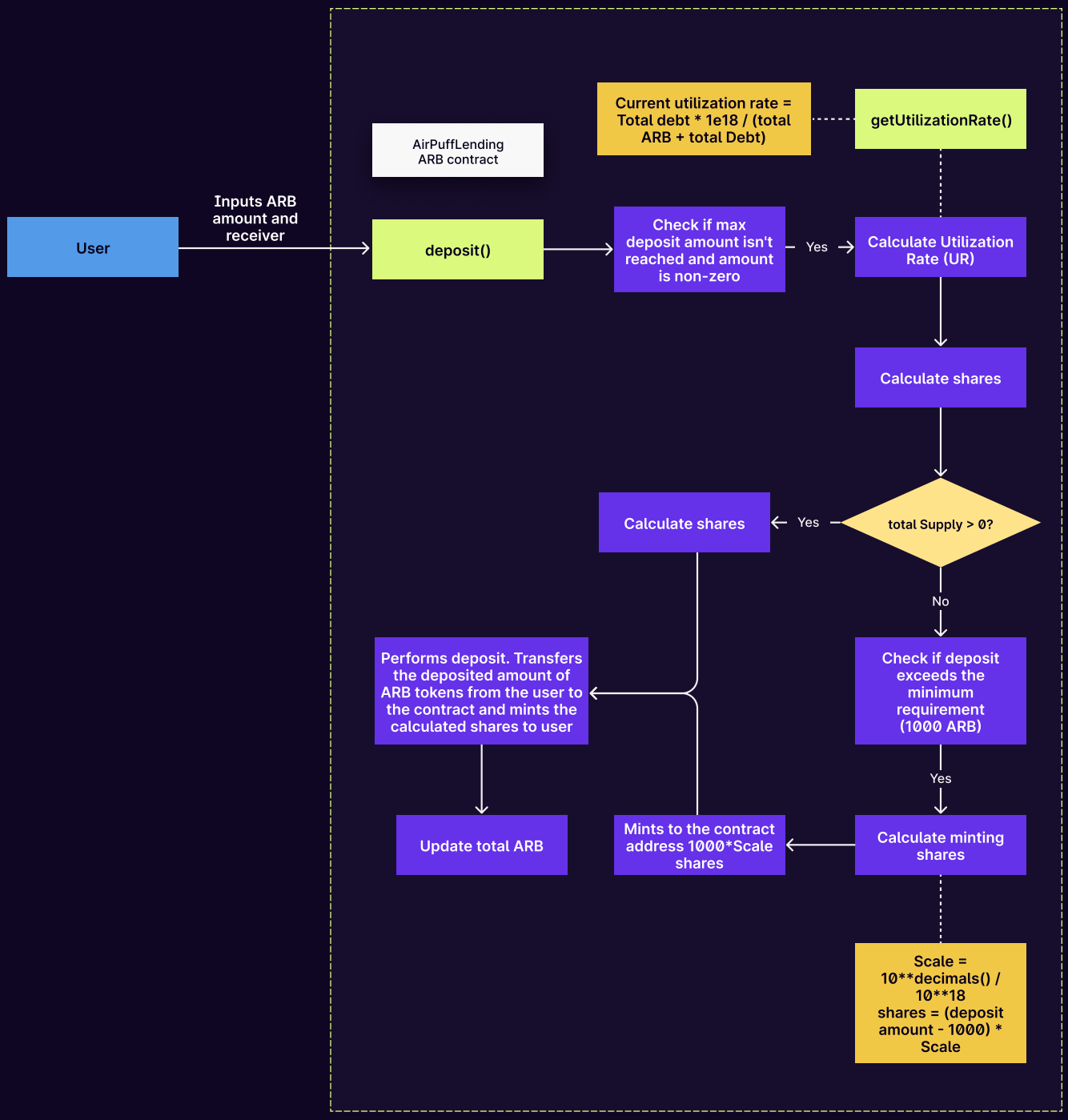
Another part of the audit process involved tests suite preparation. Zokyo Security prepared its own set of unit tests and additional scenarios to cover all main user flows and functionalities. The complete set of unit tests can be found in the Code Coverage and Test Result sections.

Overall, the security level of the contracts is high. The contract is well-written, with good informative code comments. Nevertheless, the contract fails to check against the backdoors, as the contract is upgradeable. Despite its common approach, it still creates a controllable backdoor, which should be noted in the funds-bearing contract. From other points of view the protocol functions as expected.

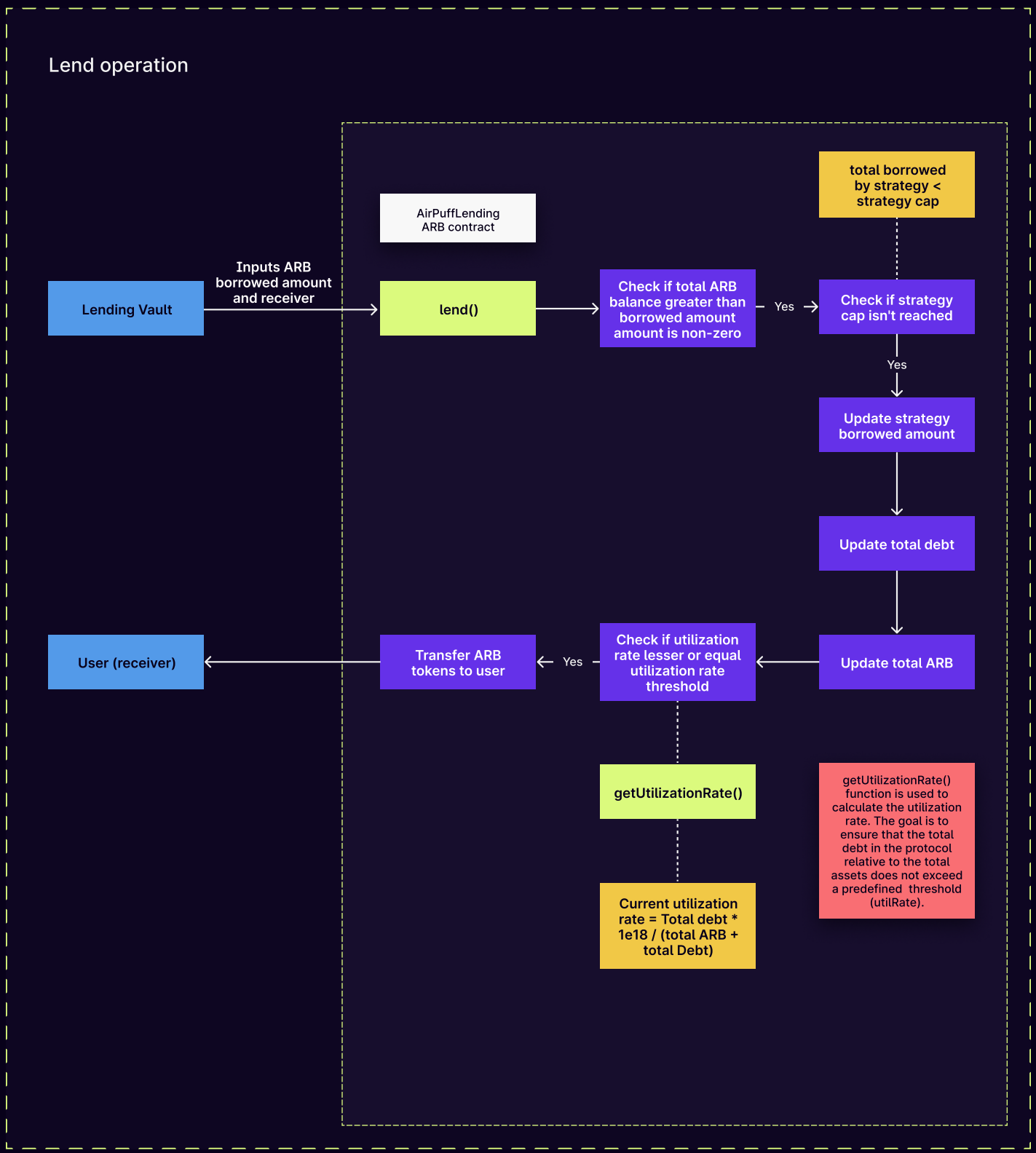
Withdraw operation

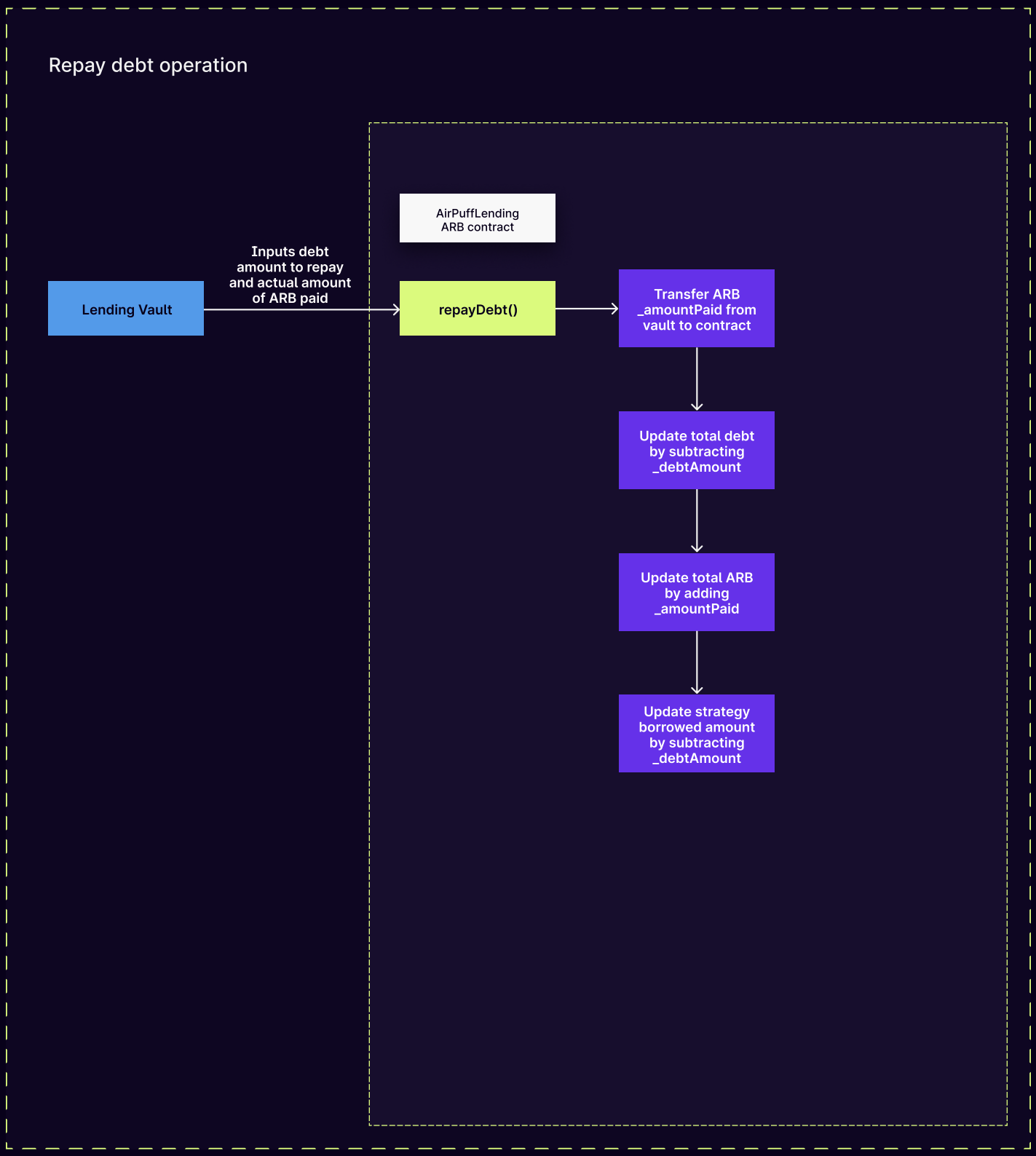


Deposit operation

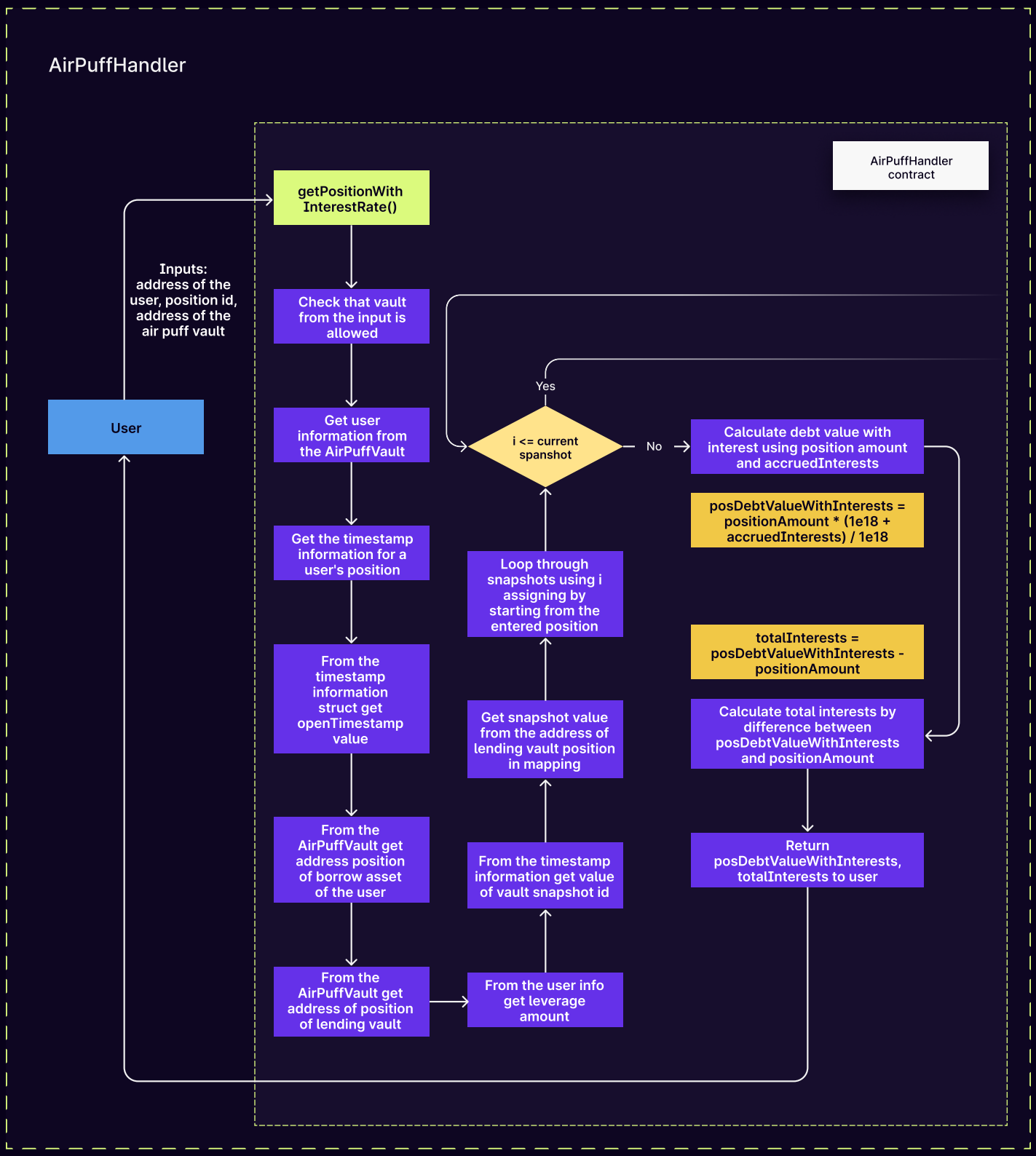


AirPuffLendingARB

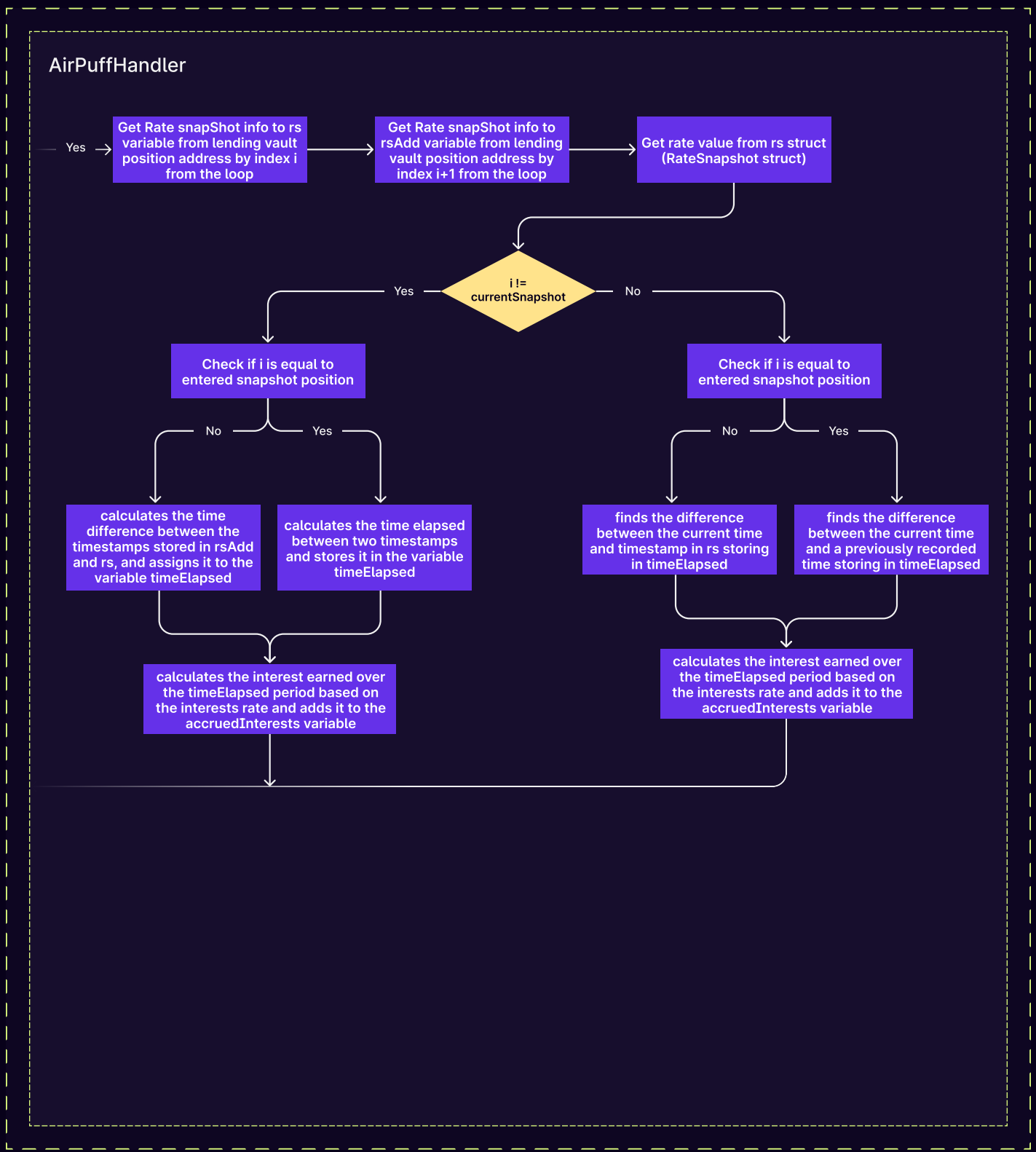



Repay debt operation
AirPuffLending ARB contract
Inputs debt amount to repay and actual amount of ARB paid
Lending Vault
repayDebt()
Transfer ARB _amountPaid from vault to contract
Update total debt by subtracting _debtAmount
Update total ARB by adding _amountPaid
Update strategy borrowed amount by subtracting _debtAmount

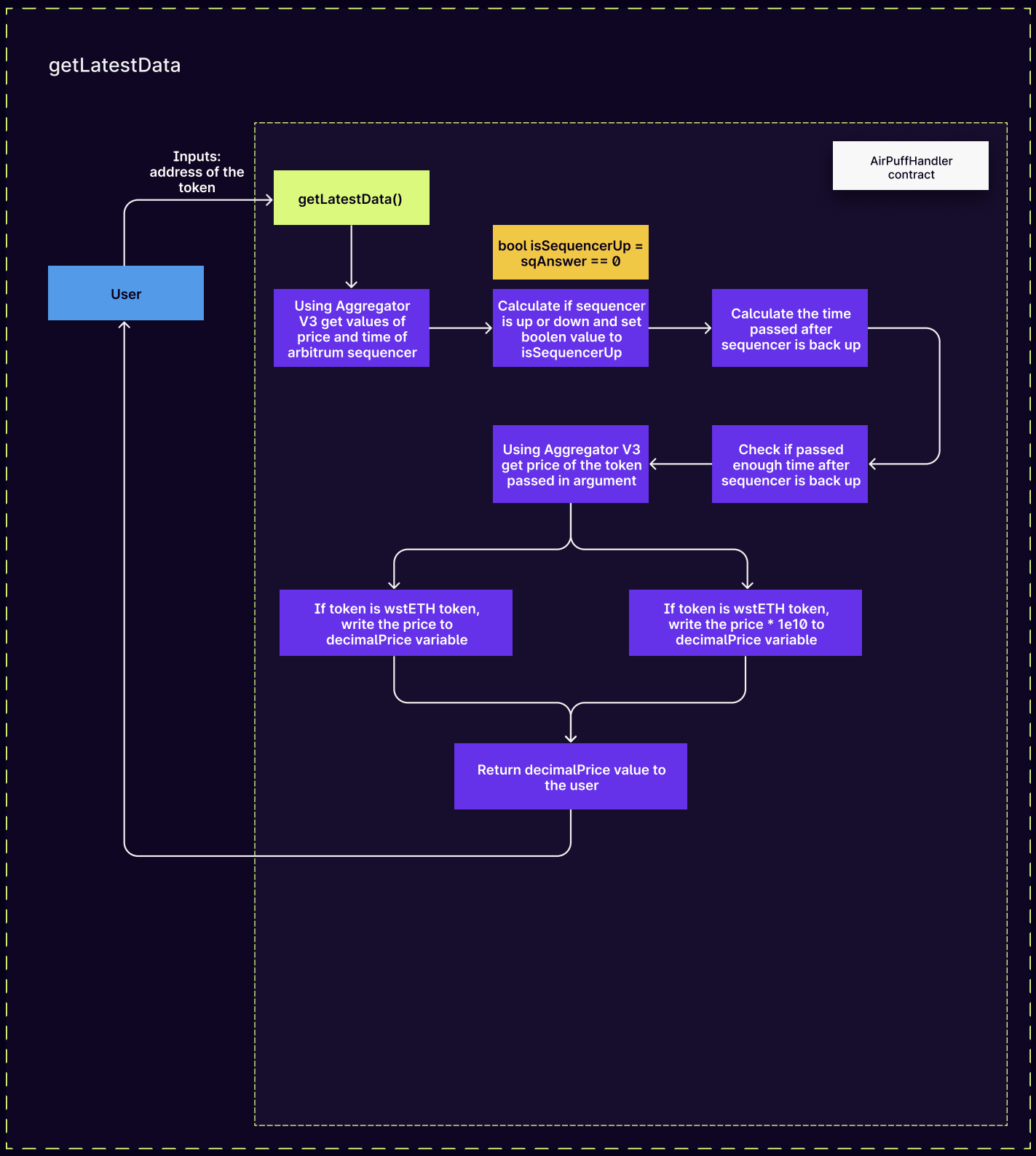
AirPuffHandler

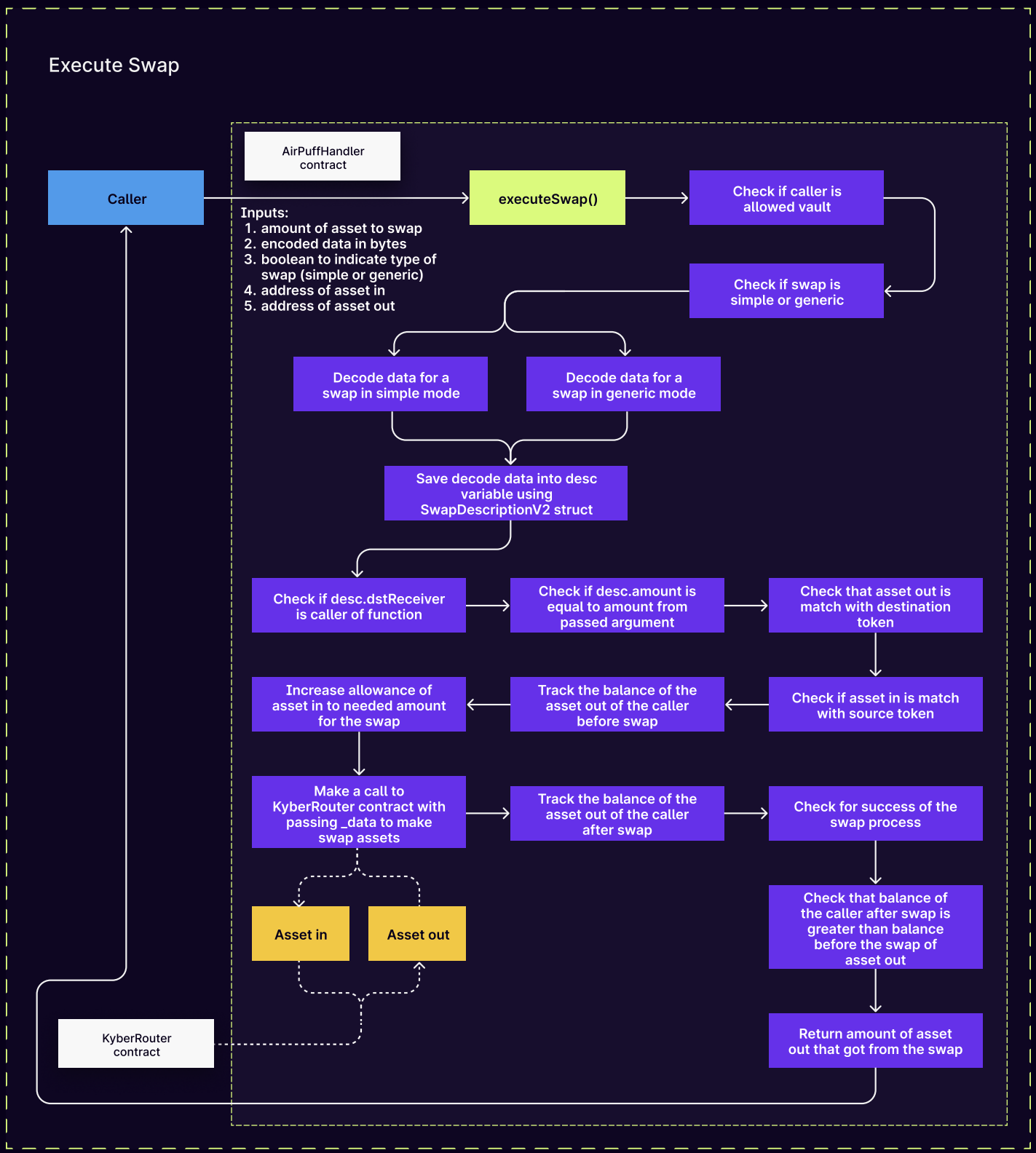


AirPuffHandler

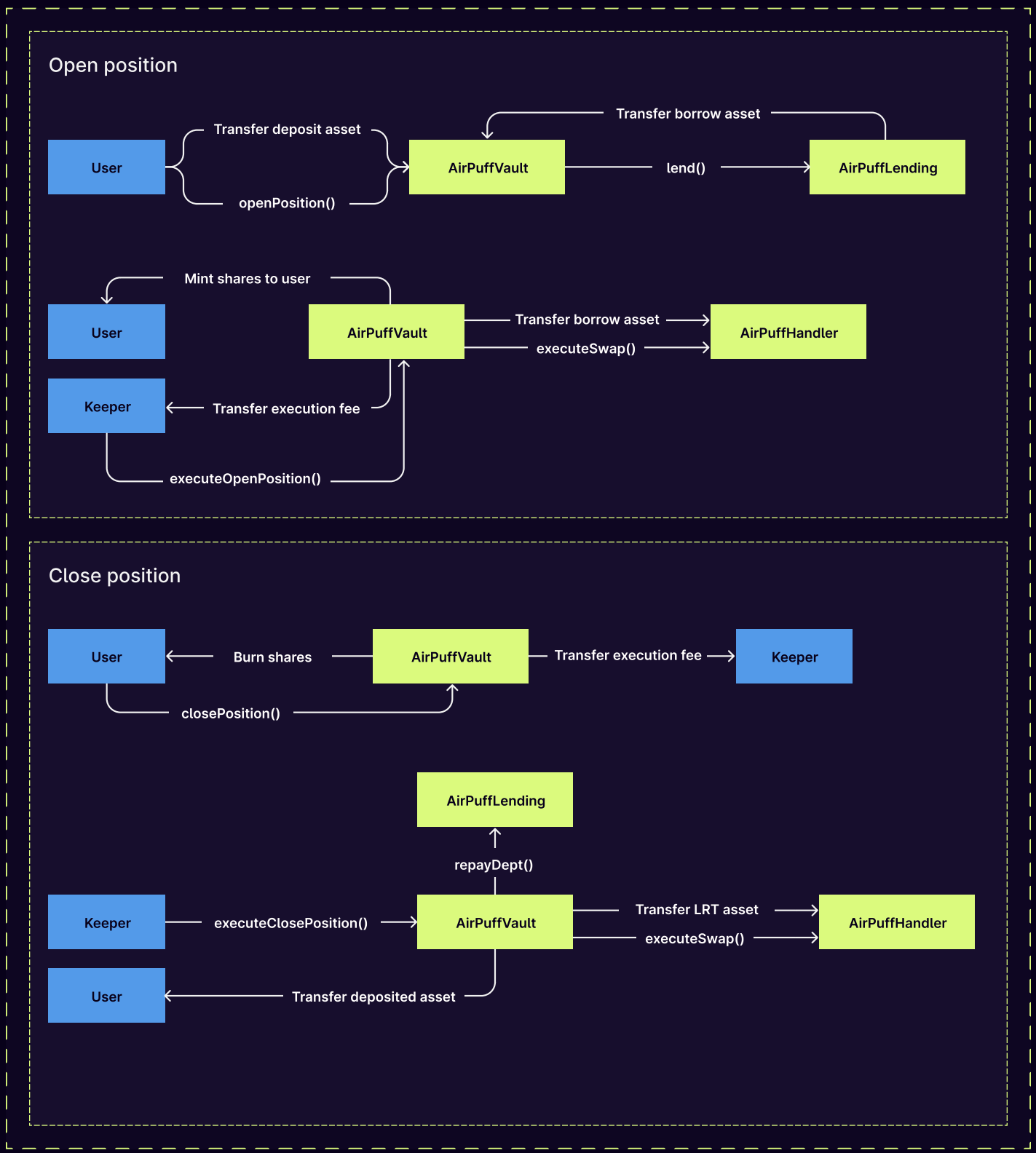


AirPuffHandler

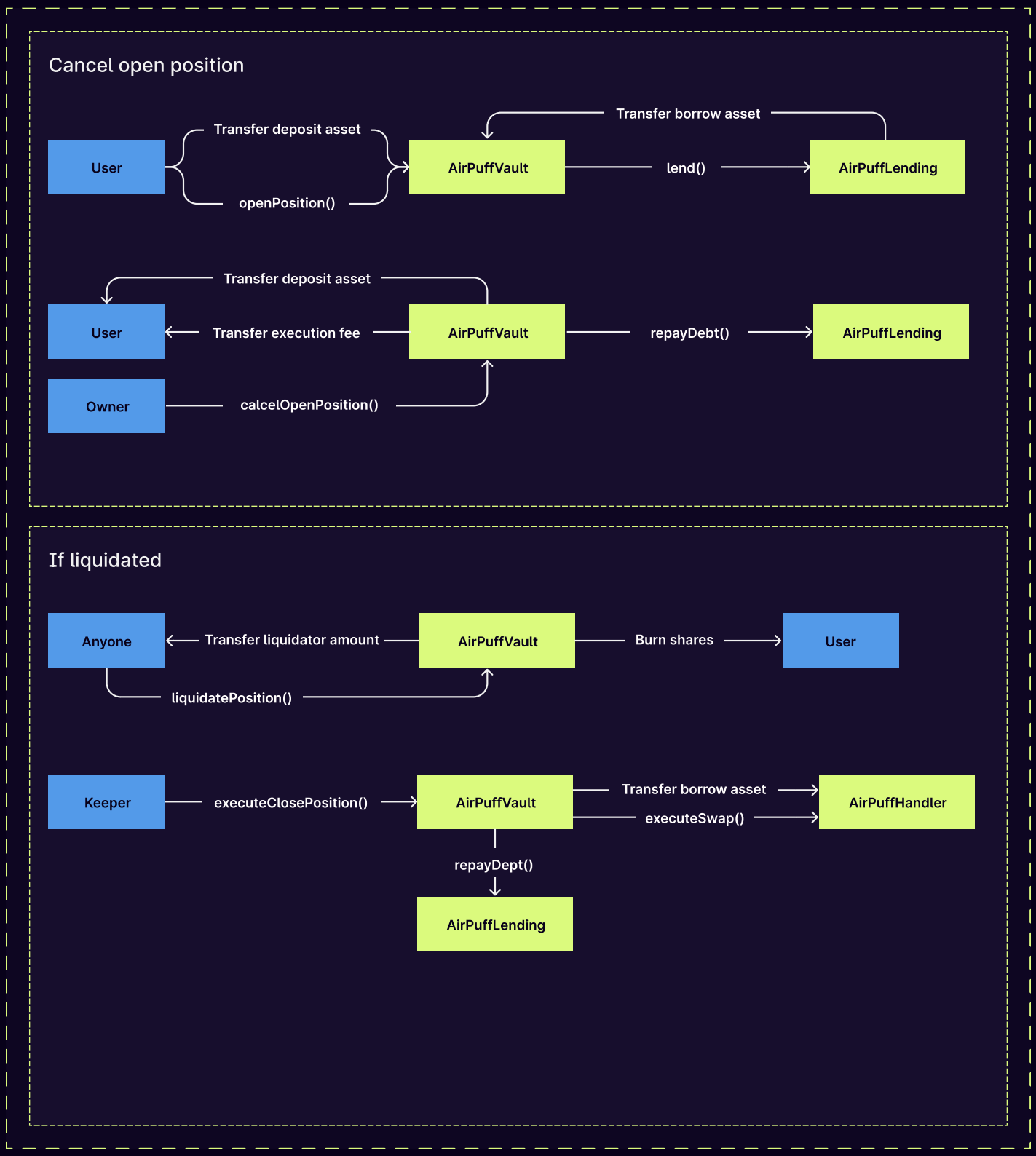



Execute Swap
AirPuffHandler contract
Caller
executeSwap()
Check if caller is allowed vault
Inputs:
1. amount of asset to swap
2. encoded data in bytes
3. boolean to indicate type of swap (simple or generic)
4. address of asset in
5. address of asset out
Check if swap is simple or generic
Decode data for a swap in simple mode
Decode data for a swap in generic mode
Save decode data into desc variable using SwapDescriptionV2 struct
Check if desc.dstReceiver is caller of function
Check if desc.amount is equal to amount from passed argument
Check that asset out is match with destination token
Increase allowance of asset in to needed amount for the swap
Track the balance of the asset out of the caller before swap
Check if asset in is match with source token
Make a call to KyberRouter contract with passing _data to make swap assets
Track the balance of the asset out of the caller after swap
Check for success of the swap process
Asset in
Asset out
Check that balance of the caller after swap is greater than balance before the swap of asset out
KyberRouter contract
Return amount of asset out that got from the swap

User flow


Open position
Transfer deposit asset
Transfer borrow asset
User
AirPuffVault
lend()
AirPuffLending
openPosition()
Mint shares to user
User
AirPuffVault
Transfer borrow asset
AirPuffHandler
executeSwap()
Keeper
Transfer execution fee
executeOpenPosition()
Close position
User
Burn shares
AirPuffVault
Transfer execution fee
Keeper
closePosition()
AirPuffLending
repayDept()
Keeper
executeClosePosition()
AirPuffVault
Transfer LRT asset
AirPuffHandler
executeSwap()
User
Transfer deposited asset

User flow


Cancel open position
Transfer borrow asset
Transfer deposit asset
User
AirPuffVault
lend()
AirPuffLending
openPosition()
Transfer deposit asset
User
Transfer execution fee
AirPuffVault
repayDebt()
AirPuffLending
Owner
calcelOpenPosition()
If liquidated
Anyone
Transfer liquidator amount
AirPuffVault
Burn shares
User
liquidatePosition()
Keeper
executeClosePosition()
AirPuffVault
Transfer borrow asset
AirPuffHandler
executeSwap()
repayDept()
AirPuffLending

PROTOCOL OVERVIEW

Description

AirPuffLendingARB:

The AirPuffLending smart contract is a specialized lending contract within the AirPuff ecosystem, designed to manage lending operations for a specific asset. This contract leverages OpenZeppelin's upgradeable contracts for ERC4626.

Key functionalities include:

- Asset Handling: The contract is primarily designed to manage the asset, supporting operations like deposits, withdrawals, lending, and debt repayment.
- Deposits: Users can deposit tokens into the contract, receiving shares that represent their portion of the vault. The contract calculates shares based on the current asset price within the vault.
- Withdrawals: Shareholders can withdraw their deposited tokens by redeeming their shares. Withdrawals are subject to fees and the current asset price.
- Utilization Rate Calculation: The contract calculates the utilization rate as the ratio of total debt to the sum of total tokens in the vault and total debt. This rate helps determine the lending and borrowing dynamics within the platform.
- Withdrawal Fees: The contract charges fees on withdrawals, defined as a percentage of the withdrawn amount. Fees are transferred to a specified fee receiver address.
- Protocol Fees Management: Owners can set and update protocol fee parameters, including the fee receiver address and the withdrawal fee percentage.

AirPuffHandler:

One of the key contracts in the AirPuff ecosystem is the contract to execute asset swaps, control and manage interest rates, receive asset prices, and manage allowed vaults.

Key functionalities include:

- Swap sessions: Gives you the ability to converte of one asset to another.
- Interest rate control: Is responsible for controlling the interest rate to adapt to market situations and creating snapshots of the current interest rate of all allowed vaults.
- Vault Management: Manage vaults, namely adding vaults to the list of allowed vaults.
- Oracles: Integrates Oracles from Chainlink to get the latest information on assets. Control which assets and Oracles are being used.
- Retrieve user information: retrieves user information from AirPuffVault, by specific ID position.

AirPuffVaultweETH:

The AirPuffVault smart contract enables users to open leveraged positions. The contract interacts with AirPuffLending to get lending tokens and with AirPuffHandler to perform swaps for the main vault token, which, in the case of the AirPuffVaultweETH contract, is weETH.

Key functionalities include:

- Positions: Gives users the ability to open or close leveraged positions.
- LRTAsset storage: Swaps deposited user tokens to assets stored on the contract.

Roles and Responsibilities

AirPuffLendingARB.sol:

1. Owner:
    - Can allowlist or blocklist lending strategies by specifying the strategy address and its status.
    - Set the utilization rate parameter, controlling how much of the protocol's assets can be borrowed.
    - Grants or revokes permission for a specific address to increment the total ARB balance of the contract.
    - Set the maximum borrow limit (cap) for a specific lending strategy.
    - Set the fee receiver address and the percentage of withdrawal fees charged on asset withdrawals.
    - Set the time duration for the timelock associated with user deposits and withdrawals.
    - Increase the total ARB balance of the contract.
2. Lending Vault:
    - Can lend ARB to the protocol, updating total debt and total ARB balances.
    - Can repay debt to the protocol, updating total debt and total ARB balances.
3. ARBGifter (Allowlisted address):
    - Allowlisted addresses can call the `increaseTotalARB` function to increase the total ARB balance of the contract.

AirPuffHandler.sol:

1. Owner:
   - Set the keeper's address.
   - Allows to update the addresses of swap routers and components responsible for swap execution.
   - Can allow or disallow vaults to interact with the handler.
   - Set Chainlink Oracles for specific tokens.
   - Set the interest rate parameters using ceiling/maximum interest slopes and base interest rate.
   - Has allowance to pause control functions.
2. Keeper:
   - Has permission to take a snapshot of the interest rates for all allowed lending vaults by giving the address of the AirPuffVault.

AirPuffVault.sol:

1. Owner:
   - Set leverage limits for a specific borrowed asset
   - Set the addresses for API3Oracle and AirPuffHandler
   - Set/delete the lending vault address for a specific asset
   - Whitelists a deposit/borrow assets
   - Update the protocol fee configuration
   - Cancel an open position request based on a pre-existing request ID
2. Keeper:
   - Execute open/closed position requests.

List of values assets

1. ERC20 ARB Token

The main asset used for lending, borrowing, depositing, and withdrawing within the protocol (AirPuffLendingARB).

2. Shares are minted to users during the ARB tokens deposit

Shares represent the position and allow to withdraw deposited ARB and accrued rewards. Shares act like standard ERC-20 tokens, meaning that they can be transferred, approved, traded, etc. (AirPuffLendingARB)

3. WETH, wstETH ERC20 tokens

Assets are set in the swap handler addresses and are involved in the swap execution process.

AIRPUFF SETTINGS

Settings

1. **AirPuffLendingARB.sol:**

- ARB Token Address (ARB): Represents the address of the ARB token, which is the primary asset used within the lending and borrowing protocol.
- AllowedStrategies (allowedStrategies): Is a mapping of addresses representing lending strategies that are responsible for lending ARB to the protocol and repaying debt. It is a key component in the protocol's liquidity management.
- Utilization Rate (utilRate): Is a parameter that indicates the proportion of assets in use within the protocol. The owner can set this rate to control the maximum allowed utilization, managing the risk of excessive leverage.
- Allowed To Gift (allowedToGift): Is related to addresses that are permitted to increase the total ARB balance of the contract.
- Strategy Caps (strategyCap): Defines the maximum borrowing limit (cap) for each allowed strategy. It prevents strategies from borrowing excessively, helping to control risk and maintain stability.
- Fee Receiver Address (feeReceiver): The address that receives withdrawal fees. Withdrawal fees are charged when users withdraw assets from the protocol.
- Withdrawal Fees Percentage (withdrawalFees): Defines the percentage of withdrawal fees charged when users withdraw assets. The value is in relation to the constant `DENOMINATOR`, where `withdrawalFees / DENOMINATOR` represents the actual percentage.
- Lock Time for Deposits and Withdrawals (lockTime): Determines the time duration for the timelock associated with user deposits and withdrawals. Users may be required to wait for a specified period before accessing their deposited assets or making new withdrawals.

2. **AirPuffHandler.sol:**

- Interest rates configuration (interestRateConfig): This is a structure that consists of base interest rate and slope parameters used to calculate rates under varying utilization conditions. This allows the platform to adapt to changing market dynamics.
- AirPuffVault (AirPuffVault): The address of the air puff vault contract.
- Keeper (keeper): The address of the keeper. The keeper has permission to call the function for taking interest snapshots.
- Swap handling addresses (swapHandlerAddresses): This is a variable with a type structure that defines critical components for swap routers and tokens that are involved in swaps. Available for editing only by the owner of the contract.

- Vault Allowlisting (allowedVaults): This mapping is responsible for managing permitted vaults, which can interact with the handler functions such as getting position value with interest rate, taking snapshots of interest rates, and swap execution.
- Chainlink Oracles (chainlinkOracle): This mapping is responsible for setting and updating the addresses of Chainlink oracles for tokens. It is also used to get the latest price data of the token.

3. **AirPuffVault.sol:**

- StrategyAddresses: The addresses of API3Oracle and AirPuffHandler.
- FeeConfiguration: The address of the air puff vault contract.
- Asset addresses: The addresses for LRTAsset wstETH and WETH.
- Whitelist deposit/borrow asset: The addresses of assets to be able to deposit or borrow.

Deployment script

AirPuffLendingARB:

The AirPuffLendingARB contract was already deployed on Arbitrum chain.

https://arbiscan.io/address/0xc5cf090c81e5e72d425750a4f9be16e0dadf0e66#code

The contract managed through TransparentUpgradeableProxy:

https://arbiscan.io/address/0×529f94bcd37896b6a38452497C62b2F0a8217517#code

There is a deployment script provided for AirPuffLendingARB. It uses TransparentUpgradeableProxy address to manage the AirPuffLendingARB. ARB address is set to the Arbitrum token address, AirPuffVault is used as the allowed address and ARB gifter is set to this address.

AirPuffHandler:

A deployment script is provided for AirPuffHandler with the deployment managing via OpenZeppelin proxy. _AirPuffVault address is set during the initialization. Initialization also includes the initial setting for interest rate configurations. The base interest rate is initialized with the hardcoded value 5e16, CEIL_SLOPE_1 is equal to 8e17, while CEIL_SLOPE_2 is equal to 1e18. MAX_INTEREST_SLOPE_1 is 1e17, and MAX_INTEREST_SLOPE_2 is 3e17. Deployment script also sets Chainlink oracles for assets: wEther, wStEth, usdc, usdt, usdce, and ARB. The script also includes setting the keeper address, allowlisting of the AirPuffVault, setting swap handler addresses, and a snapshot of the interest rate.

AIRPUFF DESCRIPTION

AirPuffVault:

Deployment script is located in migrations/1_deploy.js. AirPuffVault is deployed using hardhat deployProxy. The script initializes weETH, wstETH and WETH with:
0×35751007a407ca6FEFfE80b3cB397736D2cf4dbe,
0×5979D7b546E38E414F7E9822514be443A4800529,
0×82aF49447D8a07e3bd95BD0d56f35241523fBab1

Addresses are valid for the Arbitrum network. Values for MAX_BPS, DENOMINATOR and DECIMAL are hardcoded into the initialize() function as: 100_000, 1_000, 1e18. After the deployment, script sets up lending vaults for weth, wsteth, ARB, USDC, USDT and USDCe tokens.

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” depending on whether they have been fixed or addressed. 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.

HIGH-1 | VERIFIED

Debt repayment manipulation.

1. Based on the provided parameters, there is a scenario when the repaid debt is fully redeemed, but only a part of the general debt is paid. In the `repayDebt()` function, the caller can pass ANY `_debtAmount` parameter that does not correspond to the `_amountPaid` parameter. Here is the scenario: Contract lends 10,000 ARB to the user → The contract records a debt of 10,000 for the borrower → The borrower attempts to repay the debt by calling `repayDebt()` with: `_debtAmount` = 10,000 (indicating full repayment), `_amountPaid` = 100 (a low amount)

According to the current logic in the function:

- The `totalDebt` would be reduced by `_debtAmount` (10,000), marking the debt as fully repaid.
- However, the `totalARB` would only be increased by `_amountPaid` (100), a small fraction of the actual debt.

Thus, it creates a discrepancy. According to the' totalDebt' record, the borrower seems to have "repaid" the entire debt, but based on the change in' totalARB, ' they haven't actually provided enough ARB to cover it.

2. Further debt manipulation is possible, as the function can be called an unlimited number of times. The `totalDebt` could be fully reduced and even underflowed, by executing `repayDebt()` function with invalid `_debtAmount` value. It will cause significant changes at utilization rate and total assets calculations and will influence entire contract behavior.

Recommendation:

Remove the parameter `_debtAmount` and decrease the total debt in the amount actually repaid. OR verify that this is part of business logic. The lending vault will execute `repayDebt()` with the correct parameters, and further logic outside the contract will cover the unpaid debt.

Post audit.

The client verified that since the protocol is built on top of the lending vaults, the accounting is assumed to be properly integrated. _debtAmount will always equal the amount borrowed initially (the strategy interacting with the lending vault has a record of the values). The purpose of _amountPaid is to be able to pass in any amounts in case a late liquidation occurs, therefore impacting the vault share price. Nevertheless, auditors note that security checks are referred out of the contract, thus the team should still monitor the depending functionality.

Frontrun attack possible.

AirPuffHandler.executeSwap()
After executing the swap in the function, there are some checks. The first check is responsible for checking whether the swap failed, and the other check is responsible for checking whether the balance is increased. However, there is no check that after the swap, the amount of tokens is equal to or higher than the expected minimum amount. Such an approach can cause a front-running attack as the amount is not validated for a certain

Recommendation:
Add additional checks for the amount of swapped tokens to prevent possible attacks.

Post audit.
The client verified that since the protocol is built on top of the Kyberswap, the front-running is assumed to be handled on its side. Nevertheless, auditors note that while referring part of the security to the trusted protocol is okay, the team should still monitor the depending functionality.

LOW-1 | RESOLVED

Withdrawal fee can be set to 100%.

AirPuffLendingARB.sol, setProtocolFeesParams()
The contract calculates the withdrawal fee as a percentage `withdrawalFees` of the withdrawal amount `_assets`. The `withdrawalFees` variable is an integer that represents a basis point value. The DENOMINATOR constant is set to 10000, which essentially converts the basis points to a percentage (by dividing by 100). If the `withdrawalFees` is set to 10000 (the same value as `DENOMINATOR`), the entire withdrawal amount would be consumed as a fee.

Recommendation:
Consider adding validation for the withdrawal fee to be less than 100%.

Post audit.
Added validation for the withdrawal fee not exceeding 50%.

Code styling.

AirPuffLendingARB.sol

1. It's important to follow a single code style. Using `+=` and `-=` for simple variable updates is considered good practice in Solidity. In the code there are several places, where this is not observed:
   - repayDebt():
     totalDebt = totalDebt - _debtAmount;
   - lend():
     totalDebt = totalDebt + _borrowed;

Updating the provided code with `+=` will increase readability and overall quality of the contract.

2. There are several places with magic numbers in the contract that can be replaced with more descriptive constants.
   - deposit():
     require(_assets > 1000, "Not Enough Shares for first mint");
     uint256 SCALE = 10 ** decimals() / 10 ** 18;
     shares = (_assets - 1000) * SCALE;
     uint256 toAsset = _mint(address(this), 1000 * SCALE)

Updating the provided code with constant `MINIMUM_MINT_AMOUNT` instead of `1000` will increase readability and overall quality of the contract.

1. There are several places with unused code.
   - address public LendingVault;
   - function dummy()

Even if the code described above will be used in the future, it is still important to remove all the unused code before deployment.

Recommendation:
Consider updating code according to the Solidity best practices.

Post audit.
Unused code was removed. Though code style violation was acknowledged

INFORMATIONAL-2 | RESOLVED

Unused variables.

AirPuffHandler.AirPuffVault
This variable is set during the initialize function, but there is no function where this variable is used. Thus, there is no point in setting it in the initialize function.

Recommendation:
Remove unused variables from the function.

Post audit.
The initialization function in the contract AirPuffHandler was fixed, and _airPuffVault is no longer set. Thus, we recommend removing AirPuffVault variable from the contract as this variable is not used anywhere.

INFORMATIONAL-3	ACKNOWLEDGED

Lack of validation.

1. AirPuffLendingARB.sol, lend(), repayDebt(), increaseTotalARB()
There is no validation for non-zero values in the lend(), repayDebt(), increaseTotalARB() functions. Even if there is an access control system, this can be exploited by a malicious actor who can execute these functions in order to perform spam transactions or as a part of a complex reentrancy attack.
2. AirPuffHandler.setInterestRate()
All params are validated except baseInterestRate, which plays a crucial role in the system and should be checked to prevent causes.
3. AirPuffHandler.setSwapHandlerAddresses()
The function is responsible for setting crucial parameters in the swap process. Avoiding checks for zero addresses may increase the chances of human error mistakes.

Recommendation:
Consider adding validation for non-zero values. Consider adding validation to baseInterestRate in the setInterestRate function.

INFORMATIONAL-4	ACKNOWLEDGED

Custom errors should be used.

Starting from the 0.8.4 version of Solidity it is recommended to use custom errors instead of storing error message strings in storage and use “require” statements. Using custom errors is more efficient in terms of gas spending and increases code readability.

Recommendation:
Use custom errors.

INFORMATIONAL-5 | RESOLVED

Lack of event emitting.

AirPuffHandler.setSwapHandlerAddresses()
This function is responsible for setting crucial addresses that participate in swaps. Four params participate in the system settings, but only one emits an event, which may cause some issues.
AirPuffLendingARB.sol, increaseTotalARB
The `increaseTotalARB` function is responsible for increasing the total ARB balance, which is an important part of utilization rate calculations and all the main functionalities. No event is emitted, which may cause some issues.

Recommendation:
Add appropriate events

Post audit.
Events added to the contracts.

INFORMATIONAL-6 | VERIFIED

Unused lock time functionality.

AirPuffLendingARB.sol, lockTime, userTimelock
There is no corresponding functionality for the lock time and the user's timelock. Even if this code will be used in the future, it is still important to remove the unused code before deployment.

Recommendation:
Consider removing unused functionality.

Post audit.
The Customer verified that this functionality might be used in the future.

	Airpuff protocol (scoped contracts)
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	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

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting AirPuff in verifying the correctness of their contracts code, our team was responsible for writing integration tests using the Hardhat testing framework.

The tests were based on the functionality of the code, as well as a review of the AirPuff contracts requirements for details about issuance amounts and how the system handles these.

AirPuffHandler

initialization
- ✓ Should initialize properly
- ✓ Should revert if trying to initialize twice

Setters

Keeper
- ✓ Should allow to set keeper
- ✓ Should revert if keeper is zero address
- ✓ Should revert if caller is not owner

Swap Handler Addresses
- ✓ Should allow to set swap handler addresses
- ✓ Should revert if caller is not owner

Vaults management
- ✓ Should allow to set vaults
- ✓ Should revert if caller is not owner
- ✓ Should revert if vault is zero address

Chainlink oracles management
- ✓ Should allow to set chainlink oracles
- ✓ Should revert if caller is not owner
- ✓ Should revert if asset is zero address

Interest rate config
- ✓ Should allow to set interest rate config
- ✓ Should revert if caller is not owner
- ✓ Should revert if ceil slopes invalid
- ✓ Should revert if max interest slopes invalid

Price getter
✓ Should allow to get price from chainlink oracle (66ms)
User timestamo getter
✓ Should allow to get user timestamp info
Interest rate getter
✓ Should allow to get interest rate when utilRate greater than CEIL_SLOPE_1
✓ Should allow to get interest rate when utilRate less than CEIL_SLOPE_1 (83ms)
Interest snapshot
✓ Should allow to take interests snapshot
✓ Should revert if caller is not keeper
✓ Should revert if vault is not allowed
User info getter
✓ Should allow to get user info
Swap execution
✓ Should allow to make swap (1791ms)
✓ Should revert if caller is not allowed vault (522ms)
✓ Should revert if receiver is not allowed vault (783ms)
✓ Should revert if swaps amount is not match (1169ms)
✓ Should revert if tokenIn is not match (597ms)
✓ Should revert if tokenIn is not match (450ms)

AirPuffLendingARB
Initialization
✓ Initialize correctly
✓ Initialize twice fails
✓ Initialize with zero address fails
Balance of ARB
✓ Get ARB balance
Utilization rate
✓ Get utilization rate
AirPuff price
✓ Get AirPuff price when total assets is zero
✓ Get AirPuff price when total supply is not zero (50ms)
Total assets
✓ Get Total assets

```
# Set allowed strategy
  ✓ Set by non-owner fails
  ✓ Set zero address fails
  ✓ Set strategy
# Set utilization rate
  ✓ Set by non-owner fails
  ✓ Set invalid util rate fails
  ✓ Set utilization rate
# Set ARB gifter
  ✓ Set by non-owner fails
  ✓ Set zero address fails
  ✓ Set ARB gifter
# Set strategy cap
  ✓ Set by non-owner fails
  ✓ Set zero address fails
  ✓ Set zero cap fails
  ✓ Set strategy cap
# Set protocol fees params
  ✓ Set by non-owner fails
  ✓ Set zero address fails
  ✓ Set invalid withdrawal fees fails
  ✓ Set protocol fees params
# Set lock time
  ✓ Set by non-owner fails
  ✓ Set invalid lock time fails
  ✓ Set lock time
# Deposit
  ✓ Deposit zero value fails
  ✓ Deposit first time with not enough shares fails
  ✓ Deposit first time
  ✓ Deposit if total supply isn't zero (57ms)
# Lend
  ✓ Lend by not allowed strategy fails
  ✓ Borrow more than total ARB fails (40ms)
  ✓ Borrow more than borrow cap fails
  ✓ Leverage ratio too high fails (45ms)
  ✓ Lend (53ms)
```

Repay debt
✓ Repay by not allowed strategy fails
✓ Repay debt (72ms)
Withdraw
✓ Withdraw zero value fails
✓ Withdraw more than max value fails (54ms)
✓ Withdraw more than vault balance fails (73ms)
✓ Withdraw (70ms)
Mint
✓ Mint fails
Redeem
✓ Redeem fails
Increase total ARB
✓ Increase by not allowed ARB gifter address fails
✓ Increase total ARB
Scenarios
✓ Inflation attack (81ms)
✓ Rounding error (159ms)

AirPuffVault
Main
✓ Open position (1335ms)
Setters
✓ Set protocol fee
✓ Set borrowed asset leverage limit
✓ Set strategy addresses
✓ Set lending vault
✓ Whitelist deposit asset
✓ Whitelist borrow asset

We are grateful for the opportunity to work with the AirPuff 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 AirPuff team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo