Security Audit Report

The f(x) Protocol Updates:

The fxSAVE and Stability Pool USDC Yield Strategy

SECBIT

March 17, 2025

1. Introduction

The AladdinDAO is a decentralized network to shift crypto investments from venture capitalists to the wisdom of crowds through collective value discovery. To enhance user returns, the f(x) protocol has introduced the following two updates:

- The protocol has introduced the `fxSAVE` module, allowing users to earn compounded returns by depositing `fxBASE` or `fxBASE-gauge` tokens.
- The protocol can further deposit idle USDC tokens into Stability Pool V2.0 into Aave to earn interest. All generated interest will be fully distributed to holders of Stability Pool tokens.

SECBIT Labs conducted an audit from February 6 to March 17, 2025, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that fxSAVE and stability pool USDC yield strategy module contracts have no critical security risks. The SECBIT team has some tips on logical implementation, potential risks, and code revising (see part 4 for details).

Type	Description	Level	Status
Design & Implementation	4.3.1 Funds mistakenly deposited into the vault contract cannot be withdrawn.	Info	Discussed
Design & Implementation	4.3.2 Discussion on FXN Token yield handling in the `harvest()` function.	Info	Fixed
Design & Implementation	4.3.3 In extreme cases, users' asset redemption operations may fail.	Info	Discussed
Design & Implementation	4.3.4 Theoretically, there is a risk of sandwich attacks being used to steal contract rewards.	Info	Discussed
Design & Implementation	4.3.5 The `execute()` function in the contract is not actually used.	Info	Discussed
Design & Implementation	4.3.6 Users may incur a loss of principal when redeeming fxABSE tokens.	Info	Discussed
Design & Implementation	4.3.7 Attackers may steal the protocol's earnings generated on the Aave protocol.	Info	Fixed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about fxSAVE and stability pool USDC yield strategy module are shown below:

- Smart contract code
  - initial review commit
    - fxSAVE: *0ffa96f*
    - stability pool USDC yield strategy: *1b46767*
  - final review commit
    - fxSAVE: *e1a06b7*
    - stability pool USDC yield strategy: *6498785*

2.2 Contract List

The following content shows the contracts included in the fxSAVE and stability pool USDC yield strategy module, which the SECBIT team audits:

Name	Lines	Description
ConcentratorBase.sol	72	Abstract contract providing an interface for updating the protocol's core parameters.
SavingFxUSD.sol	202	Users can deposit `fxBASE` or `fxBASE-gauge` tokens into this contract to earn compounded returns.
RewardHarvester.sol	26	The earnings from the `fxSAVE` protocol will be transferred to this contract. This contract will convert the received rewards into corresponding assets and reinvest them into the `fxSAVE` contract for compounding.
FxUSDBasePoolV2Facet.sol	43	Auxiliary contract facilitating users to deposit assets into the `fxBASE` gauge.
SavingFxUSDFacet.sol	45	Auxiliary contract facilitating users to directly deposit or redeem `fxBASE` tokens in the `fxSAVE` contract.
AaveV3Strategy.sol	60	Provides an interface to the Aave protocol for the f(x) protocol to help it deposit USDC tokens into the Aave protocol.
StrategyBase.sol	26	Abstract contracts, base strategy contracts.
AssetManagement.sol	48	Asset Management Contract, the Administrator can deposit USDC tokens in Aave Agreement through the interface.

3. Contract Analysis

This part describes code assessment details, including "role classification" and "functional analysis".

3.1 Role Classification

Two key roles in the fxSAVE and stability pool USDC yield strategy modules are the Governance Account and Common Account.

- Governance Account
    - Description

Contract Administrator

- Authority
    - Update fees ratio

    - Transfer ownership
    - Convert contract yields for reinvestment
    - Deposit USDC tokens into the Aave protocol
- Method of Authorization

The contract administrator is the contract's creator or authorized by transferring the governance account.

- Common Account
    - Description

Users participate in the FX protocol

- Authority
    - Deposit `fxBASE` or `fxBASE-gauge` tokens into the `fxSAVE` contract to earn returns
- Method of Authorization

No authorization required

3.2 Functional Analysis

The team has developed the `fxSAVE` contract, allowing users to deposit their `fxBASE` or `fxBASE-gauge` tokens into the protocol to earn additional returns. Additionally, to improve capital efficiency and increase user returns, USDC tokens deposited into the protocol will be further deposited into the Aave protocol to generate yield. All earnings will be fully distributed to holders of Stability Pool tokens. The SECBIT team conducted a detailed audit of some of the contracts in the protocol. We can divide the critical functions of the contract into two parts:

SavingFxUSD、RewardHarvester and SavingFxUSDFacet

Users can deposit their `fxBASE` or `fxBASE-gauge` tokens into these contracts. The earned `wstETH` and `FXN` token rewards will be automatically converted into `fxBASE` tokens and reinvested into the `fxBASE-gauge` contract. The main functions in these contracts are as follows:

- `depositGauge()`

Users can use this function to deposit `fxBASE-gauge` tokens into the contract and receive corresponding `fxSAVE` token shares.

- `requestRedeem()`

When users call this function to request the redemption of `fxBASE` tokens, the pending `fxBASE` tokens will be transferred to a designated address and locked for a specified period. Upon maturity, these `fxBASE` tokens will be converted into `fxUSD` and `USDC` tokens.

- `claim()/claimFor()`

Users can call these functions after the fund lock-up period ends to redeem `fxUSD` and `USDC` tokens.

- `harvest()`

Users can call this function to collect the protocol's earnings and send them to the `harvester` contract.

- `swapAndDistribute()`

This function allows the administrator to convert the contract's earnings into target tokens and reinvest them into the `SavingFxUSD` contract.

- `depositToFxSave()`

Users can use this function to convert other tokens into `fxUSD` or `USDC`, deposit the converted funds into the `fxBASE` contract, and subsequently deposit the received `fxBASE` tokens into the `fxSAVE` contract.

- `redeemFromFxSave()`

This function helps users redeem `fxUSD` and `USDC` tokens from the `fxSAVE` contract, convert them into a specified token, and transfer these assets to the `receiver`.

AaveV3Strategy、StrategyBase and AssetManagement

USDC tokens deposited into the protocol will be further deposited into the Aave protocol by the system, allowing users to earn compound returns from both the f(x) protocol and the Aave protocol. The main functions in these contracts are as follows:

- `kill()`

In an emergency, the asset manager can call this function to withdraw all USDC tokens deposited in the Aave protocol.

- `manage()`

The asset manager can call this function to deposit USDC tokens from the contract into the Aave protocol.

4. Audit Detail

This part describes the process, and the detailed audit results also demonstrate the problems and potential risks.

4.1 Audit Process

The audit strictly followed the audit specification of SECBIT Lab. We analyzed the project from code bugs, logical implementation, and potential risks. The process consists of four steps:

- Fully analysis of contract code line by line.
- Evaluation of vulnerabilities and potential risks revealed in the contract code.
- Communication on assessment and confirmation.
- Audit report writing.

4.2 Audit Result

After scanning with adelaide, sf-checker, and badmsg.sender (internal version) developed by SECBIT Labs and open source tools, including Mythril, Slither, SmartCheck, and Securify, the auditing team performed a manual assessment. The team inspected the contract line by line, and the result could be categorized into the following types:

Number	Classification	Result
1	Normal functioning of features defined by the contract	✓
2	No obvious bug (e.g., overflow, underflow)	✓
3	Pass Solidity compiler check with no potential error	✓
4	Pass common tools check with no obvious vulnerability	✓
5	No obvious gas-consuming operation	✓

6	Meet with ERC20 standard	✓
7	No risk in low-level call (call, delegatecall, callcode) and in-line assembly	✓
8	No deprecated or outdated usage	✓
9	Explicit implementation, visibility, variable type, and Solidity version number	✓
10	No redundant code	✓
11	No potential risk manipulated by timestamp and network environment	✓
12	Explicit business logic	✓
13	Implementation consistent with annotation and other info	✓
14	No hidden code about any logic that is not mentioned in the design	✓
15	No ambiguous logic	✓
16	No risk threatening the developing team	✓
17	No risk threatening exchanges, wallets, and DApps	✓
18	No risk threatening token holders	✓
19	No privilege on managing others' balances	✓
20	No non-essential minting method	✓
21	Correct managing hierarchy	✓

4.3 Issues

4.3.1 Funds mistakenly deposited into the vault contract cannot be withdrawn.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Location

SavingFxUSD.sol#L138

Description

During the initialization of the `SavingFxUSD` contract, a dedicated fund management `vault` contract is created, with `SavingFxUSD` set as the `vault` contract's owner.

For example, in the case of address 0x79c4f5, to prevent funds from being mistakenly deposited into the `vault` contract and becoming irretrievable, the contract includes a `transferTokens()` function that allows the `vault` owner to withdraw mistakenly sent assets. However, since the `SavingFxUSD` contract is the owner of the `vault` contract but does not provide an interface to call `vault.transferTokens()`, any mistakenly deposited funds in the `vault` contract cannot be retrieved.

```
function initialize(address admin, InitializationParameters memory
params) external initializer {
    __Context_init();
    __ERC165_init();
    __AccessControl_init();

    __ERC20_init(params.name, params.symbol);
    __ERC20Permit_init(params.name);
    __ERC4626_init(IERC20(base));

    __ConcentratorBase_init(params.treasury, params.harvester);

    _grantRole(DEFAULT_ADMIN_ROLE, admin);

    //@audit create a dedicated vault for the `SavingFxUSD` contract
    vault = IConvexFXNBooster(BOOSTER).createVault(params.pid);
    _updateThreshold(params.threshold);

    IERC20(base).forceApprove(gauge, type(uint256).max);
  }
```

```
//@audit
https://etherscan.io/address/0x79c4f5ccbebbd4a6793885d8a985ee99f1b223b5#
code
//return any tokens in vault back to owner
function transferTokens(address[] calldata _tokenList) external
onlyOwner{
    //transfer tokens back to owner
    //fxn and gauge tokens are skipped
    _transferTokens(_tokenList);
```

```
    }

    //transfer other reward tokens besides fxn(which needs to have fees
    applied)
    //also block gauge tokens from being transfered out
    function _transferTokens(address[] memory _tokens) internal{
        //transfer all tokens
        for(uint256 i = 0; i < _tokens.length; i++){
            //dont allow fxn (need to take fee)
            //dont allow gauge token transfer
            if(_tokens[i] != fxn && _tokens[i] != gaugeAddress){
                uint256 bal = IERC20(_tokens[i]).balanceOf(address(this));
                if(bal > 0){
                    IERC20(_tokens[i]).safeTransfer(owner, bal);
                }
            }
        }
    }
```

Status

The developers explained that funds mistakenly deposited into the vault contract will not affect the regular operation of the protocol and can be disregarded.

4.3.2 Discussion on FXN Token yield handling in the `harvest()` function.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Fixed

Location

SavingFxUSD.sol#L223

Description

Users can retrieve the earnings from the `vault` contract using the `harvest()` function, which sends these rewards to the `harvester` address for further processing. According to the currently deployed contract on-chain, the gauge.getActiveRewardTokens() includes two types of rewards: FXN token and wstETH token. The `_transferRewards()` function within the for loop has already distributed the FXN token rewards from the contract, so handling FXN token earnings separately appears unnecessary after the for loop.

```
function harvest() external {
    IStakingProxyERC20(vault).getReward();
    address[] memory tokens =
IMultipleRewardDistributor(gauge).getActiveRewardTokens();
    address cachedHarvester = harvester;
    uint256 harvesterRatio = getHarvesterRatio();
    uint256 expenseRatio = getExpenseRatio();
    for (uint256 i = 0; i < tokens.length; ++i) {
      _transferRewards(tokens[i], cachedHarvester, harvesterRatio,
expenseRatio);
    }
    // @audit it seems that the reward token already includes FXN token
rewards, so there is no need for handling separately.
    _transferRewards(FXN, cachedHarvester, harvesterRatio,
expenseRatio);
  }
```

Status

The developers fixed this issue in a commit e1a06b7 by adding a check for the FXN token. If the reward has already been processed within the for loop, it will not be handled again afterward.

4.3.3 In extreme cases, users' asset redemption operations may fail.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Location

SavingFxUSD.sol#L246

Description

The `SavingFxUSD` contract allows users to deposit `fxBASE` or `fxBASE-gauge` tokens in exchange for `fxSAVE` token receipts. When users deposit `fxBASE-gauge` tokens via `depositGauge()`, these funds are directly stored in the `vault` contract. However, when users deposit `fxBASE` tokens through the `deposit()` or `mint()` functions, these tokens need to be further deposited into the `gauge` contract, which mints `fxBASE-gauge` tokens to the `vault` contract.

If the deposited `fxBASE` amount does not exceed a predefined threshold, the tokens remain temporarily stored in the `SavingFxUSD` contract rather than being immediately deposited into the `gauge` contract. However, when users request to redeem funds, the system only attempts to withdraw `fxBASE` from the `vault`, without considering the `fxBASE` temporarily stored in `SavingFxUSD`.

In extreme cases, this could lead to redemption failures. For example, if the threshold is set at 1000 `fxBASE`, and the `vault` holds only 800 `fxBASE-gauge` tokens, a user depositing 900 `fxBASE` will have these funds temporarily stored in `SavingFxUSD`. If the same user then attempts to redeem 900 `fxBASE`, the system will try to withdraw 900 `fxBASE-gauge` tokens from the `vault`, which only holds 800, resulting in a failed redemption.

That said, since the threshold is intended to reduce gas fees for small depositors and is unlikely to be set too high, the chances of a redemption failure should be low.

```
function _deposit(address caller, address receiver, uint256 assets,
uint256 shares) internal virtual override {
 ERC4626Upgradeable._deposit(caller, receiver, assets, shares);

    // batch deposit to gauge through convex vault
    uint256 balance = IERC20(base).balanceOf(address(this));
    // @audit When the amount of `fxBASE` tokens in the contract does
not exceed the threshold, these tokens will temporarily remain in the
`SavingFxUSD` contract instead of being immediately deposited into the
`gauge` contract.
    if (balance >= getThreshold()) {
      // @audit Deposit `fxBASE` tokens into the `gauge` contract, and
the `vault` contract will receive `fxBASE-gauge` tokens.
      ILiquidityGauge(gauge).deposit(balance, vault);
    }
}

function getThreshold() public view returns (uint256) {
    return _miscData.decodeUint(THRESHOLD_OFFSET, THRESHOLD_BITS);
}
```

Status

The developers have acknowledged this issue and explained that the probability of occurrence is very low. If the mentioned situation arises, the administrator can resolve it by calling the `swapAndDistribute()` function to deposit the `fxBASE` tokens held in the `SavingFxUSD` contract into `fxBASE-gauge`.

4.3.4 Theoretically, there is a risk of sandwich attacks being used to steal contract rewards.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Location

RewardHarvester.sol#L46-L58

SavingFxUSD.sol#L279-L289

Description

The `SavingFxUSD` contract handles rewards as follows:

1). Any user can call the `harvest()` function to claim the `vault` contract's rewards and send them (wstETH and FXN tokens) to the `RewardHarvester` contract.

2). A designated address then calls the `swapAndDistribute()` function in the `RewardHarvester` contract to swap the received wstETH and FXN tokens into target tokens (`fxBASE-gauge` tokens, `fxUSD`, or `USDC`) and reinvest them via the internal `onHarvest()` function.

Since the current implementation distributes rewards in a one-time manner and depositing `fxBASE` tokens has no redemption lock-up period (users can deposit via the `deposit()` function and immediately redeem via the `redeem()` function), an attacker could theoretically exploit a sandwich attack to steal rewards from the `vault`.

Specifically, upon detecting an admin call to `swapAndDistribute()` in the transaction pool, an attacker could:

1). Front-run the transaction by depositing a large amount of `fxBASE` tokens into the `SavingFxUSD` contract via the `deposit()` function.

2). Allow the admin's `swapAndDistribute()` function to execute, distributing the rewards.

3). Immediately call the `withdraw()` function to redeem their principal plus the allocated rewards (in `fxBASE` tokens).

```
// @audit RewardHarvester.sol
function swapAndDistribute(
    uint256 amountIn,
    address baseToken,
```

```
    address targetToken,
    TradingParameter memory params
  ) external returns (uint256 amountOut) {
    // swap base token to target
 amountOut = _doTrade(baseToken, targetToken, amountIn, params);

    // transfer target token to receiver
    IERC20(targetToken).safeTransfer(receiver, amountOut);
    IHarvesterCallback(receiver).onHarvest(targetToken, amountOut);
  }
```

```
// @audit ConcentratorBase.sol
function onHarvest(address token, uint256 amount) external override {
    if (_msgSender() != harvester) revert ErrorCallerNotHarvester();

    _onHarvest(token, amount);
  }

// @audit SavingFxUSD.sol
/// @inheritdoc ConcentratorBase
function _onHarvest(address token, uint256 amount) internal virtual
override {
  if (token == gauge) {
    IERC20(gauge).safeTransfer(vault, amount);
    return;
  } else if (token != base) {
    IERC20(token).forceApprove(base, amount);
    IFxUSDBasePool(base).deposit(address(this), token, amount, 0);
  }
 amount = IERC20(base).balanceOf(address(this));
  ILiquidityGauge(gauge).deposit(amount, vault);
}
```

Status

The developers explained that the risk of a sandwich attack in the current implementation is very low. The protocol incorporates the following four measures to mitigate potential risks:

1). The `swapAndDistribute()` function can only be called by a specific whitelisted address, and the timing of the calls is randomized. This increases the cost for attackers, as they need to monitor real-time transactions.

2). The protocol utilizes protected RPC nodes for reward distribution, reducing the likelihood of a sandwich attack.

3). The `swapAndDistribute()` function is called frequently, preventing excessive accumulation of rewards. As a result, the potential profit from a sandwich attack may not be sufficient to cover the associated gas costs.

4). The required `fxBASE` tokens have relatively low liquidity in the secondary market, making it difficult for attackers to acquire a large amount of `fxBASE` tokens to execute the attack.

Given these factors, the developers believe the current implementation effectively mitigates the risk.

4.3.5 The `execute()` function in the contract is not actually used.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Location

StrategyBase.sol#L33-L39

Description

The `FxUSDBasePool` contract has introduced the USDC tokens asset management module, allowing asset managers authorized by the administrator to deposit USDC tokens from the `FxUSDBasePool` contract into the Aave protocol to generate yield.

Currently, the `operator` of the `StrategyBase` contract is set to the `FxUSDBasePool` contract. As a result, the `execute()` function within this contract can only be called by `FxUSDBasePool`, which inherits from the `AssetManagement` contract. However, it appears that `FxUSDBasePool` does not require the functionality provided by this function.

Additionally, since the `execute()` function directly invokes the `call` instruction, it can execute arbitrary operations, including withdrawing USDC tokens from Aave and transferring them to any address. This introduces a high-risk scenario, where an attacker or unauthorized access could reduce the amount of principal available for user redemptions.

Given these security concerns, evaluating whether the `execute()` function should be retained in the contract is essential.

```
modifier onlyOperator() {
    if (msg.sender != operator) revert();
    _;
  }

function execute(
  address to,
  uint256 value,
  bytes calldata data
) external onlyOperator returns (bool success, bytes memory returnData)
{
  (success, returnData) = to.call{ value: value }(data);
 }
```

Status

The development team explained that the administrator could upgrade the `FxUSDBasePool` contract in emergencies and call the `execute()` function to perform necessary operations, such as urgently transferring contract funds to protect user assets.

4.3.6 Users may incur a loss of principal when redeeming fxABSE tokens.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Discussed

Location

FxUSDBasePool.sol#L170

Description

In the original code logic, the parameter `totalStableToken` records the amount of USDC tokens held by the contract. Its value changes under the following four conditions:

1). Users deposit USDC tokens into the `FxUSDBasePool` contract (increase).

2). Users redeem USDC tokens from the `FxUSDBasePool` contract (decrease).

3). Execution of the f(x) protocol's rebalance or liquidation operations (almost no change).

4). Execution of the `arbitrage()` function, where buying USDC tokens in the secondary market increases the value, and selling decreases it.

The parameter `totalStableToken` does not change over time without considering these four types of user-initiated interactions. Considering these interactions, since the total locked value `totalUSD` of the `FxUSDBasePool` contract remains unchanged, the total value of fxUSD and USDC tokens that users redeem remains relatively stable.

Under the current code logic, the parameter `totalStableToken` varies over time. Specifically, USDC tokens deposited into the `FxUSDBasePool` contract are further deposited into the Aave protocol to generate interest. The corresponding `strategy` address receives Aave's aUSDC tokens, which increase over time as Aave compounds the earnings within the aUSDC tokens. The `AaveV3Strategy` contract also provides a `harvest()` function, allowing the administrator to withdraw Aave earnings and send them to a designated address. When the administrator withdraws these earnings, the `totalStableToken` value decreases suddenly, and the total locked value `totalUSD` of the `FxUSDBasePool` contract also decreases.

Consider the following scenario:

1). The `FxUSDBasePool` contract has accumulated significant earnings in the Aave protocol, which, when converted into USDC token value, increases the `totalStableToken` parameter.

2). A user deposits fxUSD or USDC tokens into the `FxUSDBasePool` contract and receives a corresponding share of fxBASE tokens.

3). The administrator calls the `harvest()` function to withdraw the Aave earnings, causing the `totalStableToken` value to decrease and reducing the total locked value `totalUSD` in the `FxUSDBasePool` contract.

4). The user redeems fxBASE tokens. Since the total locked value `totalUSD` has decreased, the total value of fxUSD and USDC that the user can redeem is also lower, effectively resulting in a loss of principal.

Frequent calls to the `harvest()` function by the administrator can mitigate this issue by reducing the impact of sudden changes in `totalStableToken`.

Another issue is that since the earnings from depositing USDC into the Aave protocol are directly counted in the `totalStableToken` parameter, users redeeming fxUSD and USDC tokens proportionally through the `redeem()` function will inevitably withdraw a portion of the Aave earnings. This results in the protocol losing part of its revenue.

```
modifier sync() {
    {
      // we only manage stable token
      Allocation memory b = allocations[stableToken];
      if (b.strategy != address(0)) {
      // @audit retrieve the total principal deposited by users in the
contract, including the yield earned from Aave
        totalStableToken = IStrategy(b.strategy).totalSupply() +
IERC20(stableToken).balanceOf(address(this));
      }
    }
    _;
  }
```

```
// @audit https://github.com/AladdinDAO/fx-protocol-
contracts/blob/1b46767d6ebc94f2d85e158867f6034af55b45ee/contracts/fund/s
trategy/AaveV3Strategy.sol#L69-L78
function _harvest(address receiver) internal virtual override {
    uint256 rewards = totalSupply() - principal;

    if (rewards > 0) {
      IAaveV3Pool(POOL).withdraw(ASSET, rewards, receiver);
    }
    address[] memory assets = new address[](1);
    assets[0] = ATOKEN;
    IAaveRewardsController(INCENTIVE).claimAllRewards(assets, receiver);
  }
```

Status

The development team explained that the Aave protocol currently does not provide any earnings other than USDC tokens. Therefore, there are no plans to set up a harvester role and allow calls to the `harvest()` function. As a result, users will not experience any losses when redeeming their assets.

4.3.7 Attackers may steal the protocol's earnings generated on the Aave protocol.

Risk Type	Risk Level	Impact	Status
Design & Implementation	Info	Design logic	Fixed

Location

FxUSDBasePool.sol#L368-L406

Description

(1). According to the current code logic, any places where the `totalStableToken` parameter is used should call the sync() function beforehand to update the value of this parameter. However, both the instantRedeem() and rebalance() functions use the `totalStableToken` parameter but do not call the `sync()` function to update its value beforehand.

(2). The current protocol introduces the `instantRedeem()` function, which allows users to immediately redeem their fxUSD and USDC tokens by paying a fee, without waiting. Similar to issue 4.3.6, since the `totalStableToken` parameter includes the earnings from the Aave protocol, redeeming fxUSD and USDC tokens proportionally will inevitably involve redeeming some of the Aave protocol's earnings. If the earnings accumulated in the Aave protocol are sufficient to cover the redemption fees and gas fees, an attacker could exploit the `instantRedeem()` function to withdraw the protocol's earnings.

```
function instantRedeem(
    address receiver,
    uint256 amountSharesToRedeem
  ) external returns (uint256 amountYieldOut, uint256 amountStableOut) {
    if (amountSharesToRedeem == 0) revert ErrRedeemZeroShares();

    address caller = _msgSender();
    uint256 leftover = balanceOf(caller) -
redeemRequests[caller].amount;
    if (amountSharesToRedeem > leftover) revert
ErrorInsufficientFreeBalance();

    uint256 cachedTotalYieldToken = totalYieldToken;
    uint256 cachedTotalStableToken = totalStableToken;
    uint256 cachedTotalSupply = totalSupply();

    amountYieldOut = (amountSharesToRedeem * cachedTotalYieldToken) /
cachedTotalSupply;
    amountStableOut = (amountSharesToRedeem * cachedTotalStableToken) /
cachedTotalSupply;
    uint256 feeRatio = instantRedeemFeeRatio;

    _burn(caller, amountSharesToRedeem);

    if (amountYieldOut > 0) {
      uint256 fee = (amountYieldOut * feeRatio) / PRECISION;
```

```
        amountYieldOut -= fee;
        _transferOut(yieldToken, amountYieldOut, receiver);
        unchecked {
          totalYieldToken = cachedTotalYieldToken - amountYieldOut;
        }
      }
      if (amountStableOut > 0) {
        uint256 fee = (amountStableOut * feeRatio) / PRECISION;
        amountStableOut -= fee;
        _transferOut(stableToken, amountStableOut, receiver);
        unchecked {
          totalStableToken = cachedTotalStableToken - amountStableOut;
        }
      }

      emit InstantRedeem(caller, receiver, amountSharesToRedeem,
amountYieldOut, amountStableOut);
    }
```

Status

The development team fixed this issue in commit 772322e. In this update, the `sync()` function is called before the relevant functions, ensuring that the `totalStableToken` parameter is updated. With this fix, when an attacker deposits funds into the contract, the `totalStableToken` parameter will be synchronized, preventing the attacker from accessing the earnings accumulated in the Aave protocol.

5. Conclusion

After auditing and analyzing the fxSAVE and stability pool USDC yield strategy module, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above.

Disclaimer

SECBIT smart contract audit service assesses the contract's correctness, security, and performability in code quality, logic design, and potential risks. The report is provided "as is", without any warranties about the code practicability, business model, management system's applicability, and anything related to the contract adaptation. This audit report is not to be taken as an endorsement of the platform, team, company, or investment.

APPENDIX

Vulnerability/Risk Level Classification

Level	Description
High	Severely damage the contract's integrity and allow attackers to steal ethers and tokens, or lock assets inside the contract.
Medium	Damage contract's security under given conditions and cause impairment of benefit for stakeholders.
Low	Cause no actual impairment to contract.
Info	Relevant to practice or rationality of the smart contract could bring risks.

SECBIT Lab is devoted to constructing a common-consensus, reliable, and ordered blockchain economic entity.

https://secbit.io

audit@secbit.io

@secbit_io