Security Audit Report

Tapio Finance

SECBIT

June 16, 2023

1. Introduction

Tapio Finance aims to be the "middle layer" between liquidity collateral protocols and downstream applications in the DeFi ecosystem. Its goal is to incentivize asset holders through protocol-generated fees. Unlike traditional liquidity pools in decentralized exchanges (dexes), the LP token obtained by depositing assets into this protocol, called tapETH, is pegged to the native Ether and can be used in DeFi just like any other asset or as an exchange token. SECBIT Labs conducted an audit from May 15 to June 16, 2023, including an analysis of the smart contracts in 3 areas: **code bugs**, **logic flaws**, and **risk assessment**. The assessment shows that the Tapio Finance contract has 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
Gas optimization	4.3.1 Redundant Use of `SafeMath` Library Functions for Arithmetic Checks Can Increase Gas Consumption.	Info	Fixed
Design & Implementation	4.3.2 Discussion on the Logic of Calculating the `Ann` Parameter in the `_getD()` and `_getY()` Functions.	Info	Fixed
Design & Implementation	4.3.3 Limit the Upper Setting of Fees to Prevent Affecting the Normal Use of Contract Functions.	Info	Fixed
Gas optimization	4.3.4 Gas Optimization Strategy.	Info	Discussed
Design & Implementation	4.3.5 Users Can Actively Call the `Burn()` Function to	Low	Fixed

	Destroy the LP Tokens They Hold.		
Design & Implementation	4.3.6 Discussion on StableAssetApplication Contract's Assumption about Asset Types.	Info	Discussed
Design & Implementation	4.3.7 Be Aware of the Risk of Failure When Using the Transfer Method to Send Ether.	Low	Fixed
Design & Implementation	4.3.8 Discussion on Token Exchange Strategy Across Different Pools.	Info	Discussed
Design & Implementation	4.3.9 `StableAsset` Address Not Validated in `StableAssetApplication` Contract.	Low	Discussed
Design & Implementation	4.3.10 Changes to Key Parameters Should Emit Events.	Info	Fixed
Design & Implementation	4.3.11 Potential Risks of Different StableAsset Pools Using the Same LP Token.	Low	Discussed

2. Contract Information

This part describes the basic contract information and code structure.

2.1 Basic Information

The basic information about Tapio Finance Protocol is shown below:

- Smart contract code
    - initial review commit *d6d9687*
    - final review commit *6d130b0*

2.2 Contract List

The following content shows the contracts included in the Tapio Finance Protocol, which the SECBIT team audits:

Name	Lines	Description
StableAsset.sol	723	The `StableAsset.sol` contract allows users to trade between different tokens, with prices determined algorithmically based on the current supply and demand of each token.
StableAssetApplication.sol	189	The `StableSwapApplication.sol` contract allows users to mint pool tokens, swap between different tokens, and redeem pool tokens to underlying tokens.
StableAssetToken.sol	32	The `StableAssetToken.sol` contract represents the ERC20 token used by the StableSwap pool.

3. Contract Analysis

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

3.1 Role Classification

Two key roles in the Tapio Finance Protocol are Governance Account and Common Account.

- Governance Account
    - Description

      Contract Administrator
    - Authority
        - Update basic parameters

        - Pause mint/swap/redeem actions
        - Update the A value
    - Method of Authorization

      The contract administrator is the contract's creator or authorized by the original governance account.
- Common Account
    - Description

      Users participate in the Tapio Finance Protocol.
    - Authority
        - Mints new pool token and wrap ETH
        - Exchange between two underlying tokens with wrap/unwrap ETH
        - Redeem pool token to underlying tokens proportionally with unwrap ETH
        - Swap tokens across pool
    - Method of Authorization

      No authorization required

3.2 Functional Analysis

Tapio Finance's vision is to enhance Ethereum's position in crypto, making it the most productive asset. The LP token (tapETH) minted by the protocol aims to standardize LST liquidity into a single asset, to increase utility, yield, and capital efficiency. 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:

StableAsset

This contract allows users to trade between different tokens, with prices determined algorithmically based on the current supply and demand of each token:

- `mint()`

  Users can deposit assets and receive LP tokens generated by the protocol through this function.

- `swap()`

  Exchange between two underlying tokens.

- `redeemProportion()`

  Redeem pool tokens to underlying tokens proportionally.

- `redeemSingle()`

  Redeem pool tokens to one specific underlying token.

- `redeemMulti()`

  Redeems underlying tokens.

StableAssetApplication

As a periphery contract, it allows users to mint pool tokens, swap between different tokens, and redeem pool tokens to underlying tokens:

- `mint()`

  As a peripheral function, this function provides users with a more convenient calling interface. Users can directly participate in different liquidity pools through this function.

- `swap()`

  Exchange between two underlying tokens with wrap/unwrap ETH.

- `redeemProportion()`

  Redeem pool tokens to underlying tokens proportionally with unwrapping ETH.

- `redeemSingle()`

  Redeem pool tokens to one specific underlying token.
- `swapCrossPool()`

  Users can exchange tokens between different pools through this function.

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 Redundant Use of `SafeMath` Library Functions for Arithmetic Checks Can Increase Gas Consumption.

Risk Type	Risk Level	Impact	Status
Gas optimization	Info	More gas consumption	Fixed

Description

In Solidity 0.8.0 and later versions, overflow checks for arithmetic operations have already been built into the compiler. See the specific changes here. Therefore, there is no need to repeatedly use the `SafeMath` library in the current code, which increases gas consumption and raises the cost for users.

Problem code range:

All code involving arithmetic operations in this protocol.

Suggestion

Use mathematical operation symbols such as `+`, `-`, `*`, and `/` instead of mathematical operation functions like `add()`, `sub()`, `mul()`, and `div()`.

Status

The development team has adopted our suggestions and made the corresponding modifications in commit 5e11a8f.

4.3.2 Discussion on the Logic of Calculating the `Ann` Parameter in the `_getD()` and `_getY()` Functions.

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

Description

The current method of calculating `Ann` in the code has a clear difference from the original StableSwap code in Curve.

In the StableSwap code of Curve, there is a comment for the assignment of the `A` value, which actually multiplies it by `n * (n - 1)`. Therefore, the code only needs to multiply by n outside the loop to get `Ann`. This is a subtle optimization for the calculation of `Ann`.

```
from Curve code comment

@param _A Amplification coefficient multiplied by n * (n - 1)
```

The current StableAsset.sol uses the naive calculation method. It is recommended to explain in the comments the difference and reason from the original Curve code.

```
// @audit located in StableAsset.sol
function _getD(
    uint256[] memory _balances,
    uint256 _A
  ) internal pure returns (uint256) {
    uint256 sum = 0;
    uint256 i = 0;
```

```
        uint256 Ann = _A;
        for (i = 0; i < _balances.length; i++) {
          sum = sum.add(_balances[i]);
          Ann = Ann.mul(_balances.length);
        }
        if (sum == 0) return 0;
        uint256 prevD = 0;
        uint256 D = sum;
        for (i = 0; i < 255; i++) {
      ......
    }

  function _getY(
        uint256[] memory _balances,
        uint256 _j,
        uint256 _D,
        uint256 _A
    ) internal pure returns (uint256) {
        uint256 c = _D;
        uint256 S_ = 0;
        uint256 Ann = _A;
        uint256 i = 0;
        for (i = 0; i < _balances.length; i++) {
          Ann = Ann.mul(_balances.length);

          if (i == _j) continue;
          S_ = S_.add(_balances[i]);
          // c = c * D / (_x * N)
          c = c.mul(_D).div(_balances[i].mul(_balances.length));
        }
        ......
    }
```

Status

The development team has added relevant code comments in commit 5e11a8f.

4.3.3 Limit the Upper Setting of Fees to Prevent Affecting the Normal Use of Contract Functions.

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

Description

Administrators can set contract fees through the `setMintFee()`, `setSwapFee()`, and `setRedeemFee()` functions. It is recommended to add restrictions on the amount of fees to prevent exceeding `feeDenominator`, which could cause related contract functions to be unusable.

```
// @audit located in StableAsset.sol

uint256 public constant feeDenominator = 10 ** 10;
/**
 * @dev Updates the mint fee.
 */
function setMintFee(uint256 _mintFee) external {
  require(msg.sender == governance, "not governance");
  mintFee = _mintFee;
}

/**
 * @dev Updates the swap fee.
 */
function setSwapFee(uint256 _swapFee) external {
  require(msg.sender == governance, "not governance");
  swapFee = _swapFee;
}

/**
 * @dev Updates the redeem fee.
 */
function setRedeemFee(uint256 _redeemFee) external {
  require(msg.sender == governance, "not governance");
  redeemFee = _redeemFee;
}
```

Suggestion

Add a restriction condition for the setting of fee amounts, as follows:

```
  /**
   * @dev Updates the mint fee.
   */
  function setMintFee(uint256 _mintFee) external {
    require(msg.sender == governance, "not governance");
    // @audit add limit here
    require(_mintFee < feeDenominator, "exceed limit");
    mintFee = _mintFee;
  }

  /**
   * @dev Updates the swap fee.
   */
  function setSwapFee(uint256 _swapFee) external {
    require(msg.sender == governance, "not governance");
    // @audit add limit here
    require(_swapFee < feeDenominator, "exceed limit");
    swapFee = _swapFee;
  }

  /**
   * @dev Updates the redeem fee.
   */
  function setRedeemFee(uint256 _redeemFee) external {
    require(msg.sender == governance, "not governance");
    // @audit add limit here
    require(_redeemFee < feeDenominator, "exceed limit");
    redeemFee = _redeemFee;
  }
```

Status

The development team has updated the code and added relevant restrictions in commit 5e11a8f.

4.3.4 Gas Optimization Strategy.

Risk Type	Risk Level	Impact	Status
Gas optimization	Info	More gas consumption	Discussed

Description

In the contract bytecode, public/external functions and public variables are arranged in ascending order according to their Method ID. Smaller Method IDs will consume less gas (not considering the function content). For each increment in the order, an additional 22 gas will be consumed. When there are too many public/external functions and public variables in a contract, functions/variables with later Method IDs will consume more gas when called.

Therefore, the code can consider reducing the number of public global variables in the contract and adjusting them to `private` or `internal`, which can save gas consumption overall when calling functions.

```
// @audit located in StableAsset.sol
uint256 public constant FEE_DENOMINATOR = 10 ** 10;
uint256 public constant FEE_ERROR_MARGIN = 1000;
uint256 public constant YIELD_ERROR_MARGIN = 100000;
uint256 public constant MAX_A = 10 ** 6;
```

Status

The development team has decided to keep these variables unchanged to facilitate on-chain data querying.

4.3.5 Users Can Actively Call the `Burn()` Function to Destroy the LP Tokens They Hold.

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

Description

The `totalSupply` parameter under the `StableAsset.sol` contract represents the total amount of LP tokens `minted` in the contract. Note that users holding LP tokens can directly call the `burn()` function under the `ERC20BurnableUpgradeable.sol` contract to destroy the LP tokens they hold. This will cause the actual number of LP tokens to be inconsistent with the value of `totalSupply`. Currently, users actively calling the `burn()` function to destroy the LP tokens they hold does not appear to have a significant impact on the project, but it is necessary to confirm whether this logic meets design expectations.

```
//@audit located in StableAsset.sol
uint256 public totalSupply;

// @audit burn func from
https://github.com/OpenZeppelin/openzeppelin-contracts-
upgradeable/blob/master/contracts/token/ERC20/extensions/ERC20B
urnableUpgradeable.sol

function burn(uint256 amount) public virtual {
        _burn(_msgSender(), amount);
}

function burnFrom(address account, uint256 amount) public
virtual {
    _spendAllowance(account, _msgSender(), amount);
    _burn(account, amount);
}
```

Status

The team has fixed this issue in commit 5e11a8f and commit d4c249b..

4.3.6 Discussion on `StableAssetApplication` Contract's Assumption about Asset Types.

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

Description

When users call the `mint()` function to mint liquidity proofs (lp tokens) for a specific pool, they need to deposit funds in the form of Ether. These transferred Ethers are converted into Wrapped Ether and stored under the contract. However, the current contract does not allow users to deposit Wrapped Ether directly, which might be inconvenient for them.

A similar issue exists in the `swap()` function.

Additionally, we noticed that the current `StableAsset` contract does not stipulate that the asset must contain ETH. However, `StableAssetApplication` assumes that it does contain ETH. Is it correct to assume that the related functions in `StableAssetApplication` are only used for dealing with ETH-related logic, while the general ERC20-related logic is handled in the underlying `StableAsset` contract? For example, if users need to mint using WETH, they could directly call the interface of the `StableAsset` contract, bypassing `StableAssetApplication`.

```
// @audit located in StableAssetApplication.sol
function mint(
  StableAsset _swap,
  uint256[] calldata _amounts,
  uint256 _minMintAmount
) external payable nonReentrant {
  address[] memory tokens = _swap.getTokens();
```

```
    address poolToken = _swap.poolToken();
    uint256 wETHIndex = findTokenIndex(tokens, address(wETH));
    require(_amounts[wETHIndex] == msg.value, "msg.value equals
amounts");

    wETH.deposit{value: _amounts[wETHIndex]}();
    for (uint256 i = 0; i < tokens.length; i++) {
      if (i != wETHIndex) {
        IERC20Upgradeable(tokens[i]).safeTransferFrom(
          msg.sender,
          address(this),
          _amounts[i]
        );
      }
      IERC20Upgradeable(tokens[i]).safeApprove(address(_swap),
_amounts[i]);
    }
    uint256 mintAmount = _swap.mint(_amounts, _minMintAmount);
    IERC20Upgradeable(poolToken).safeTransfer(msg.sender,
mintAmount);
}
```

Status

The development team has confirmed the code logic and explained: "This contract is only intended for handling Ether, and for Wrapped Ether, it will be processed through the `StableAsset` contract."

4.3.7 Be Aware of the Risk of Failure When Using the Transfer Method to Send Ether.

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

Description

The `swap()`, `redeemProportion()`, and `redeemSingle()` functions use the `transfer` method to send Ether. This method only allows a maximum of 2300 gas, which may lead to the failure of sending Ether. For example, when the `recipient` address is a contract, the transfer of Ether using this method may fail due to insufficient gas. For more information, refer to this article.

```
// @audit located in StableAssetApplication.sol
function swap(
    StableAsset _swap,
    uint256 _i,
    uint256 _j,
    uint256 _dx,
    uint256 _minDy
  ) external payable nonReentrant {
    ......

    IERC20Upgradeable(tokens[_i]).safeApprove(address(_swap),
_dx);
    uint256 swapAmount = _swap.swap(_i, _j, _dx, _minDy);

    if (_j == wETHIndex) {
      wETH.withdraw(swapAmount);

      // @audit transfer could fail
      payable(msg.sender).transfer(swapAmount);
    } else {
      IERC20Upgradeable(tokens[_j]).safeTransfer(msg.sender,
swapAmount);
    }
  }

function redeemProportion(
    StableAsset _swap,
    uint256 _amount,
    uint256[] calldata _minRedeemAmounts
  ) external nonReentrant {
    ......
    for (uint256 i = 0; i < tokens.length; i++) {
      if (i == wETHIndex) {
        wETH.withdraw(amounts[i]);

        // @audit transfer could fail
        payable(msg.sender).transfer(amounts[i]);
      } else {
        IERC20Upgradeable(tokens[i]).safeTransfer(msg.sender,
amounts[i]);
      }
    }
  }

function redeemSingle(
    StableAsset _swap,
    uint256 _amount,
    uint256 _i,
    uint256 _minRedeemAmount
  ) external nonReentrant {
    ......

    if (_i == wETHIndex) {
      wETH.withdraw(redeemAmount);

      // @audit transfer could fail
      payable(msg.sender).transfer(redeemAmount);
    } else {
      IERC20Upgradeable(tokens[_i]).safeTransfer(msg.sender,
redeemAmount);
    }
  }
```

Status

The development team has made the necessary modifications to the code logic based on the suggestion, and the current code now uses the `call` instruction to send Ether in commit 08c2b759.

4.3.8 Discussion on Token Exchange Strategy Across Different Pools.

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

Description

The function `swapCrossPool()` is used for exchanging tokens between different pools. Suppose there are two pools: pool 1 and pool 2. The underlying tokens of pool 1 are tokenA, tokenB, and tokenC, corresponding to the liquidity token lp1; the underlying tokens of pool 2 are lp1 token and tokenD, corresponding to the liquidity token lp2. If the user wants to use tokenA to exchange for tokenD, according to the logic of the `swapCrossPool()` function, the user's tokenA transferred into the contract will first be used for adding liquidity to pool 1, obtaining lp1 token, and then the obtained lp1 token will be transferred into pool 2 for exchanging tokenD. The design pattern of `swapCrossPool()` follows the basePool + metaPool logic in the Curve protocol. However, the `swapCrossPool()` function can only support the one-way exchange from tokenA to tokenD, and there is no relevant function for the exchange from tokenD to tokenA in the contract. To exchange tokenD for tokenA, the user needs to:

(1). Exchange tokenD for lp1 token in pool 2; (2). Remove liquidity in pool 1 to obtain tokenA.

Therefore, it needs to be confirmed whether the contract implementation lacks the related function for reverse exchange.

In addition, for the exchange between underlying tokens of regular basePool, the current contract has not implemented relevant functions. Suppose there are two pools: pool 3 and pool 4. The underlying tokens of pool 3 are tokenE, tokenF, and tokenG; the underlying tokens of pool 4 are tokenF and tokenH. The exchange between these underlying tokens does not involve adding/removing liquidity, for example, to exchange tokenE for tokenH, it only needs to:

(1). Exchange tokenE for tokenF in pool 3; (2). Exchange tokenF for tokenH in pool 4.

Such exchange of underlying tokens in basePool is not applicable to the `swapCrossPool()` function. Therefore, it also needs to be confirmed whether the current contract lacks the logic for exchanging underlying tokens in the basePool.

```
// @audit located in StableAssetApplication.sol
function swapCrossPool(
  StableAsset _sourceSwap,
  StableAsset _destSwap,
  address _sourceToken,
  address _destToken,
  uint256 _amount,
  uint256 _minSwapAmount
) external nonReentrant {
  address[] memory sourceTokens = _sourceSwap.getTokens();
  address[] memory destTokens = _destSwap.getTokens();
  uint256 sourceIndex = findTokenIndex(sourceTokens,
_sourceToken);
  uint256 destIndex = findTokenIndex(destTokens, _destToken);

  IERC20Upgradeable(_sourceToken).safeTransferFrom(
    msg.sender,
    address(this),
    _amount
  );
  IERC20Upgradeable(_sourceToken).safeApprove(address(_sourceSw
ap), _amount);

  uint256[] memory _mintAmounts = new uint256[]
(sourceTokens.length);
  _mintAmounts[sourceIndex] = _amount;
  uint256 mintAmount = _sourceSwap.mint(_mintAmounts, 0);
  IERC20Upgradeable(_destSwap.poolToken()).safeApprove(
    address(_destSwap),
    mintAmount
  );
```

```
    uint256 redeemAmount = _destSwap.redeemSingle(
      mintAmount,
      destIndex,
      _minSwapAmount
    );

    IERC20Upgradeable(_destToken).safeTransfer(msg.sender,
  redeemAmount);
  }
```

Status

The developers have explained the design logic of the code: "The underlying LP tokens for lst pools is the same one, so exchange tokens between different pools can be done via minting LP tokens and a redeem single call."

4.3.9 `StableAsset` Address Not Validated in `StableAssetApplication` Contract.

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

Description

In the `StableAssetApplication` contract, the `StableAsset` address can be specified by the user in various functions, which may lead to unexpected execution results in the code. For example, in the `mint()` function below, the user can pass in the address of a malicious `StableAsset` contract created by them, bypassing other logic in the code and moving any token from the `StableAssetApplication` contract. It is suggested to validate whether the `StableAsset` is in a whitelist, or at least emphasize that the `StableAssetApplication` contract should never store any assets.

```
  // @audit located in StableAssetApplication.sol
    function mint(
      StableAsset _swap,
      uint256[] calldata _amounts,
      uint256 _minMintAmount
    ) external payable nonReentrant {
      address[] memory tokens = _swap.getTokens();
      address poolToken = _swap.poolToken();
      uint256 wETHIndex = findTokenIndex(tokens, address(wETH));
      require(_amounts[wETHIndex] == msg.value, "msg.value equals
  amounts");

      wETH.deposit{value: _amounts[wETHIndex]}();
      for (uint256 i = 0; i < tokens.length; i++) {
        if (i != wETHIndex) {
          IERC20Upgradeable(tokens[i]).safeTransferFrom(
            msg.sender,
            address(this),
            _amounts[i]
          );
        }
        IERC20Upgradeable(tokens[i]).safeApprove(address(_swap),
  _amounts[i]);
      }
      uint256 mintAmount = _swap.mint(_amounts, _minMintAmount);
      IERC20Upgradeable(poolToken).safeTransfer(msg.sender,
  mintAmount);
    }
```

Status

The development team has clarified there will be no assets stored in the `StableAssetApplication` contract in commit 08c2b75.

4.3.10 Changes to Key Parameters Should Emit Events.

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

Description

It is suggested to add events for changes to key parameters, for easier off-chain monitoring and tracking. For example, the `setMintFee()`, `setSwapFee()`, and `setRedeemFee()` functions.

Status

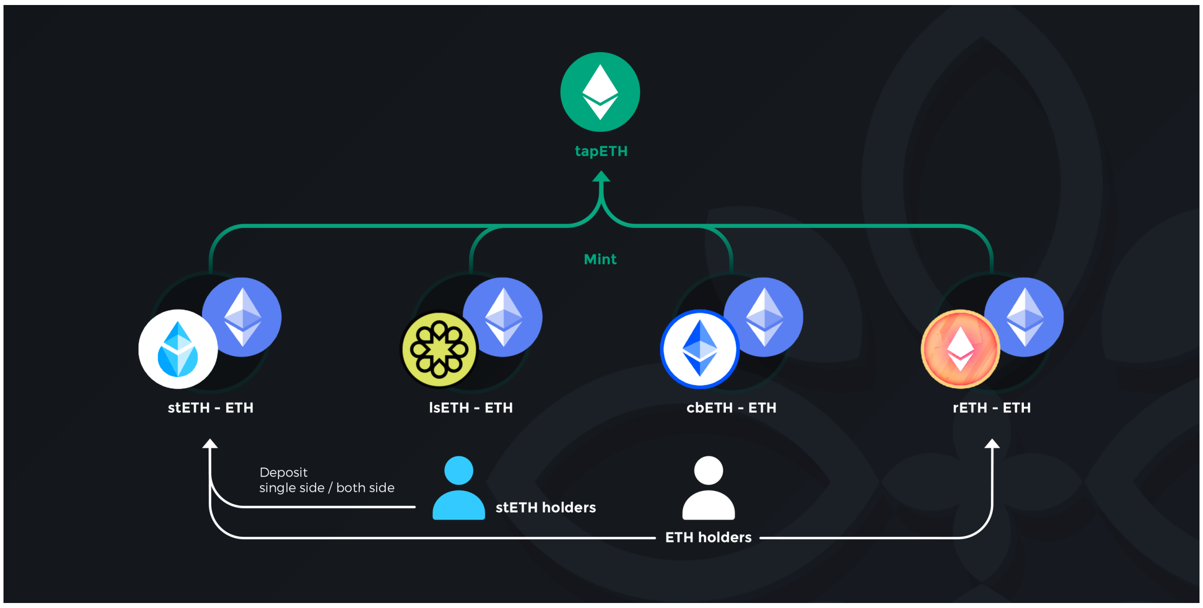
The developers have adopted our suggestions and added events to record modifications made to key parameters in commit 5e11a8f.

4.3.11 Potential Risks of Different StableAsset Pools Using the Same LP Token

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

`StableAsset`, in actual deployment, uses the same pool token (`tapETH`) as the LP token for different pools. This practice of different pairs using the same pool token is relatively rare. It implies that different pools share risks, and conceals an assumption that the value represented by the LP token in different pools is considered to be the same. If any underlying token in a pool encounters a problem, the risk will spread among different pools.

The following diagram displays tokens that Tapio might support.



Source: What is tapETH

Additionally, there could be an issue with the calculation of the LP token value in a depeg situation, as described in the official documentation:

```
As a result, the pool will institute the following pricing (as
an example):

ETH = $15 - A larger premium on the undersupplied asset (as the
pool always wants over-collateralization during times of
imbalance)
LST = $9 - A smaller discount on the oversupplied asset

As such, the values then reflect:
40 ETH = $15 each
    Total value of $600
160 LST = $9
    Total value of $1440

200 tapETH = $10 - however each tapETH is backed by $10.20
worth of assets
Total value of $2000 (Overcollateralization of 2%)
```

Source: Example Depeg Scenario

In reality, when an asset in any pool depegs, the value represented by `tapETH` will definitely decrease. The above calculation erroneously sets the price of ETH at $15. In fact, when depegging occurs, the price of ETH will not change according to the current pool price (the current pool cannot price ETH). When depegging occurs, the value represented by the LP could be `(40*10+160*9.8) = 1968`, which will always be less than `2000`. Thus, when an asset depegs, it will spread to other pools, forming risk-free arbitrage opportunities. Other normal pools will be emptied by arbitrageurs, making it difficult for the price of `tapETH` to anchor to ETH. In conclusion, the risk of different pools using the same LP token is significant, and the current model should be carefully considered.

Status

Based on the discussion with the team, the following measures are considered and proposed to reduce the identified risks in the StableAsset pools:

1. **Implementing a Mint Cap**: A mint cap for each pool, particularly for second-tier LST assets, was suggested. This would restrict the amount of `tapETH` that can be minted from a single pool, thereby helping to isolate and limit the risk related to individual pools. If a depeg situation occurs in one pool, the damage is capped and the risk does not spread uncontrollably to other pools.
2. **Active Monitoring of LST Asset Price and Arbitrage Activity**: We agreed that careful monitoring of the LST asset price and unusual arbitrage activities among pools is necessary. This would allow for early detection of price manipulation or other potential threats, helping to prevent a depeg situation from occurring.
3. **Utilizing the Mint Switch**: StableAsset currently possesses a mint switch function. We suggested that this function should be used strategically. If any abnormalities are detected through the above monitoring, the minting function could be turned off to prevent further risk. This stopgap measure could prevent the creation of new LP tokens, thus limiting the impact of a potential depeg event.

These solutions, when implemented, should significantly reduce the risk associated with different StableAsset pools using the same LP token, `tapETH`. However, it's important to continually review and assess these measures to ensure they remain effective as market dynamics change.

5. Conclusion

After auditing and analyzing the Tapio Finance Protocol contract, SECBIT Labs found some issues to optimize and proposed corresponding suggestions, which have been shown above.

Disclaimer

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 possibly 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