Security Audit Report for STND Smart contract

Date: March 12, 2022

Version: 1.7

Contact: contact@blocksecteam.com

Contents

1 Introduction **1**
1.1 About Target Contracts . . . 1
1.2 Disclaimer . . . 2
1.3 Procedure of Auditing . . . 2
1.3.1 Software Security . . . 2
1.3.2 DeFi Security . . . 3
1.3.3 NFT Security . . . 3
1.3.4 Additional Recommendation . . . 3
1.4 Security Model . . . 3

2 Findings **4**
2.1 Software Security . . . 5
2.1.1 The local variables `collateral` and `debt` shadow the global ones . . . 5
2.1.2 Fees can not be distributed as expected due to the unreachable branch . . . 5
2.1.3 Potential mistakes . . . 6
2.2 DeFi Security . . . 7
2.2.1 Anyone can withdraw the liquidated collaterals . . . 7
2.2.2 The `VaultManager` contract does not handle the debt decimals when judging whether it is a valid CDP . . . 8
2.2.3 The check in the function `liquidate` can not be passed . . . 9
2.2.4 Uninitialized CDP vaults may incur infinite MTR minted . . . 9
2.2.5 Incorrect usage to the function `isValidCDP` may incur infinite MTR minted . . . 10
2.2.6 The global variable `borrow` is not be updated correctly . . . 11
2.2.7 Users' collaterals may be locked in vaults . . . 12
2.2.8 The potential reentrancy risk . . . 12
2.2.9 The minting cap may be bypassed . . . 14
2.2.10 Price manipulation attacks against FeeRoll contract . . . 15
2.3 Additional Recommendation . . . 16
2.3.1 Finish the function `mintFromVault` . . . 17
2.3.2 Add more smart contracts in the audit list . . . 17
2.3.3 Make the codes and comments consistent . . . 18

Report Manifest

Item	Description
Client	DigitalNative
Target	STND Smart contract

Version History

Version	Date	Description
1.0	Jan 24, 2022	Report Draft
1.1	Jan 25, 2022	First Release
1.2	March 4, 2022	Issue fix
1.3	March 6, 2022	Issue fix
1.4	March 7, 2022	Issue fix
1.5	March 7, 2022	Issue fix
1.6	March 11, 2022	Add two contracts into audit range
1.7	March 12, 2022	Issue fix

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The repository that has been audited include standard-evm (STND) [1].

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for the only initial commit (`C1`)of codes, as well as new codes (in the following commits) to fix issues in the audit report.

Project		Commit SHA
standard-evm (STND)	C1	d7c016ca098a4e5a554583c499fc0cead4db7088
	C2	145469636148da56cb14f2fb3d0321f34a64d0d5
	C3	7c8677d672d43476ff897ed3a93e89462dc1ee46
	C4	263ecca4f14c1df7d8744a2170d99e43dd836fbf
	C5	31ad59271614b05b2f0e9fc8f1abccada9710b75
	C6	e7812d2b48708de9e0c66a27cfdf6b204f12efe4
	C7	0734f035c072ad806131994fa4ef1cff14045a20

Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include **vault** folder contracts and liquidator contract only. Specifically, the files covered in this audit include:

vault:

- Meter.sol
- proxy.sol
- V1.sol
- Vault.sol
- VaultFactory.sol
- VaultManager.sol
- FeeHelper.sol
- FeeRoll.sol
- dSTND.sol

pools:

- Liquidator.sol

[1] https://github.com/digitalnativeinc/standard-evm

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.
- **Fixed** The issue has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find 13 potential issues in the smart contract. We also have three recommendations, as follows:

- High Risk: 8
- Medium Risk: 2
- Low Risk: 3
- Recommendations: 3

ID	Severity	Description	Category	Status
1	Low	*The local variables* `collateral` *and* `debt` *shadow the global ones*	Software Security	Fixed
2	Low	*Fees can not be distributed as expected due to the unreachable branch*	Software Security	Fixed
3	Medium	*Potential mistakes*	Software Security	Fixed
4	High	*Anyone can withdraw the liquidated collaterals*	DeFi Security	Fixed
5	High	*The* `VaultManager` *contract does not handle the debt decimals when judging whether it is a valid CDP*	DeFi Security	Fixed
6	High	*The check in the function* `liquidate` *can not be passed*	DeFi Security	Fixed
7	High	*Uninitialized CDP vaults may incur infinite MTR minted*	DeFi Security	Fixed
8	High	*Incorrect usage to the function* `isValidCDP` *may incur infinite MTR minted*	DeFi Security	Fixed
9	High	*The global variable* `borrow` *is not be updated correctly*	DeFi Security	Fixed
10	High	*Users' collaterals may be locked in vaults*	DeFi Security	Fixed
11	Medium	*The potential reentrancy risk*	DeFi Security	Fixed
12	Low	*The minting cap may be bypassed*	DeFi Security	Fixed
13	High	*Price manipulation attacks against FeeRoll contract*	DeFi Security	Fixed
14	-	*Finish the function* `mintFromVault`	Recommendation	Fixed
15	-	*Add more smart contracts in the audit list*	Recommendation	Fixed
16	-	*Make the codes and comments consistent.*	Recommendation	Fixed

The details are provided in the following sections.

2.1 Software Security

2.1.1 The local variables `collateral` and `debt` shadow the global ones

Status Fixed

Description

Users can invoke the function `getStatus` of the contract `Vault` to check the vault's status, and the code as shown in below code snippet.

```
  function getStatus()
  external
  view
  override
  returns (
    address collateral,
    uint256 cBalance,
    address debt,
    uint256 dBalance
  )
{
  return (
    collateral,
    IERC20Minimal(collateral).balanceOf(address(this)),
    debt,
    IERC20Minimal(debt).balanceOf(address(this))
  );
}
```

Listing 2.1: Vault.sol

Since the local variables `collateral` and `debt` shadows the global ones, the function can not work as expected.

Impact Users can not check vaults' status by invoking the function `getStatus`.

Suggestion Remove the four local variables.

This issue was fixed by the commit `C3`.

2.1.2 Fees can not be distributed as expected due to the unreachable branch

Status Fixed

Description

As shown in below code snippet, all fees in vaults are distributed to three accounts: `dividend`, `feeTo`, and `treasury`.

```
  function _sendFee(
      address asset_,
      uint256 amount_,
      uint256 fee_
    ) internal returns (uint256 left) {
```

```
        address dividend = IVaultManager(manager).dividend();
        address feeTo = IVaultManager(manager).feeTo();
        address treasury = IVaultManager(manager).treasury();
        bool feeOn = feeTo != address(0);
        bool treasuryOn = treasury != address(0);
        bool dividendOn = dividend != address(0);
        // send fee to the pool
        if (feeOn) {
          if (dividendOn) {
            uint256 half = fee_ / 2;
            TransferHelper.safeTransfer(asset_, dividend, half);
            TransferHelper.safeTransfer(asset_, feeTo, half);
          } else if (dividendOn && treasuryOn) {
            uint256 third = fee_ / 3;
            TransferHelper.safeTransfer(asset_, dividend, third);
            TransferHelper.safeTransfer(asset_, feeTo, third);
            TransferHelper.safeTransfer(asset_, treasury, third);
          } else {
            TransferHelper.safeTransfer(asset_, feeTo, fee_);
          }
        }
        return amount_ - fee_;
      }
    }
```

Listing 2.2: FeeHelper.sol

However, the second branch (L285 L290) can not be reached.

Impact Fees can not be distributed as expected.

Suggestion Change the order between `if dividendOn` and `if (dividendOn && treasuryOn)`.
This issue was fixed by the commit C4.

2.1.3 Potential mistakes

Status Fixed.

Description

The code in line 812 passes the FeeRoll contract's balance of `lp` token to the internal function `removeLiquidity`, but it transfers the `msg.sender`'s `lp` token (in line 853). There might be a mistake.

The below codes come from the commit C6.

```
802   function tradeLP(
803       address lp
804   ) internal {
805       // Get each lp token specified in the LP array
806       address tokenA = IUniswapV2Pair(lp).token0();
807       address tokenB = IUniswapV2Pair(lp).token1();
808       // Remove liquidity from the old router with permit
809       (uint256 amountA, uint256 amountB) = removeLiquidity(
810           tokenA,
811           tokenB,
812           IERC20(lp).balanceOf(address(this)),
813           0,
```

```
            0,
            block.timestamp + 20000000
        );
        IUniswapV2Router01(router).swapExactTokensForTokens(amountA, 0, getPathToStnd(tokenA),
            dstnd, block.timestamp + 20000000);
        IUniswapV2Router01(router).swapExactTokensForTokens(amountB, 0, getPathToStnd(tokenB),
            dstnd, block.timestamp + 20000000);
    }
```

Listing 2.3: FeeRoll.sol

```
844    function removeLiquidity(
845        address tokenA,
846        address tokenB,
847        uint256 liquidity,
848        uint256 amountAMin,
849        uint256 amountBMin,
850        uint256 deadline
851    ) internal returns (uint256 amountA, uint256 amountB) {
852        IUniswapV2Pair pair = IUniswapV2Pair(pairForRouter(tokenA, tokenB));
853        pair.transferFrom(msg.sender, address(pair), liquidity);
854        (uint256 amount0, uint256 amount1) = pair.burn(address(this));
855        (address token0,) = UniswapV2Library.sortTokens(tokenA, tokenB);
856        (amountA, amountB) = tokenA == token0 ? (amount0, amount1) : (amount1, amount0);
857        require(amountA >= amountAMin, "BarrelRoll: INSUFFICIENT_A_AMOUNT");
858        require(amountB >= amountBMin, "BarrelRoll: INSUFFICIENT_B_AMOUNT");
859    }
```

Listing 2.4: FeeRoll.sol

Impact The invocation to the function `tradeLPs` will be reverted.

Suggestion Use the code `pair.transfer(address(pair), liquidity)` to replace the code in line 853.

2.2 DeFi Security

2.2.1 Anyone can withdraw the liquidated collaterals

Status Fixed

Description

The `liquidator` contract is designed to liquidate all invalid Collateral Debt Positions (CDPs) and get all the liquidated collaterals. After that, anyone can invoke the `distribute` function to add the liquidated collaterals into the Uniswap V2 STND/collateral pools. This design will distribute all the liquidated profits to all STND holders, since liquidated collaterals are used to support the price of STND in Uniswap V2 pools. The function `distribute` as shown in below:

```
    function distribute(address collateral) public {
        require(hasRole(DEFAULT_ADMIN_ROLE, _msgSender()), "IA"); // Invalid Access
        // check the pair if it exists
        address pair = IUniswapV2FactoryMinimal(v2Factory).getPair(
          collateral,
          debt
        );
```

```
32        require(pair != address(0), "Vault: Liquidating pair not supported");
33        // Distribute collaterals
34        TransferHelper.safeTransfer(collateral, pair, IERC20Minimal(collateral).balanceOf(address(
              this)));
35    }
```

Listing 2.5: Liquidator.sol

However, the `liquidator` contract does not deposit the liquidated collaterals correctly. As shown in the L34, it transfers the collaterals to the Uniswap V2 pool directly. These liquidated collaterals will not become the Uniswap V2 pool's reserves as expected due to the design of `sync` and `skim` in Uniswap V2 pool. In particular, the function `skim` can force the pool's balances to match reserves. Anyone can invoke the function `skim` that follows the invocation of the function `distribute` to withdraw the liquidated collaterals directly.

Impact Anyone can withdraw the liquidated collaterals by invoking the Uniswap V2 pool.`skim` right after the invocation to `distribute`.

Suggestion Invoke the function `pair.sync` to force pool's reserves to match balances right after the transfer in L34.

This issue was fixed by the commit C3.

2.2.2 The `VaultManager` contract does not handle the debt decimals when judging whether it is a valid CDP

Status Fixed

Description

The contract `VaultManager` judges whether a CDP is valid using the function `isValidCDP`, as shown in below code snippet.

```
156   function isValidCDP(address collateral_, address debt_, uint256 cAmount_, uint256 dAmount_)
          public view override returns (bool) {
157       (uint256 collateralValueTimes100Point00000, uint256 debtValue) = _calculateValues(
              collateral_, debt_, cAmount_, dAmount_);
158
159       uint mcr = getMCR(collateral_);
160       uint cDecimals = IERC20Minimal(collateral_).decimals();
161
162       uint256 debtValueAdjusted = debtValue / (10 ** cDecimals);
163
164       // if the debt become obsolete
165       return debtValueAdjusted == 0 ? true : collateralValueTimes100Point00000 /
              debtValueAdjusted >= mcr;
166   }
```

Listing 2.6: VaultManager.sol

However, it handles the collateral decimals only at L162. Since all vaults use the function to judge if the CDP is valid, the mistake is fatal to the project.

Impact All vaults in the project cannot properly know if their CDP is valid.

Suggestion Handle the debt decimals in the function `isValidCDP`.

This issue was fixed by the commit C3.

2.2.3 The check in the function `liquidate` can not be passed

Status Fixed

Description

The function `liquidate` always check if the CDP is valid before liquidating the collaterals. The check code as shown in below code snippet.

```
function liquidate() external override {
    require(
      !IVaultManager(manager).isValidCDP(
        collateral,
        debt,
        IERC20Minimal(collateral).balanceOf(address(this)),
        IERC20Minimal(debt).balanceOf(address(this))
      ),
      "Vault: Position is still safe"
    );
```

Listing 2.7: Vault.sol

Since the `IERC20Minimal(debt).balanceOf(address(this))` is zero, the invocation to the function `isValidCDP` always returns `true`. Therefore, this check will never be passed.

Impact The liquidation mechanism can not work.

Suggestion Use `getDebt()` to replace the code `IERC20Minimal(debt).balanceOf(address(this))`.

This issue was fixed by the commit C3.

2.2.4 Uninitialized CDP vaults may incur infinite MTR minted

Status Fixed

Description

The contract `VaultFactory` is designed to create vaults. According to the code in below, anyone can create vaults for any collaterals.

```
function createVault(address collateral_, address debt_, uint256 amount_, address recipient)
     external override returns (address vault, uint256 id) {
    uint256 gIndex = allVaultsLength();
    IV1(v1).mint(recipient, gIndex);
    bytes memory bytecode = type(Vault).creationCode;
    bytes32 salt = keccak256(abi.encodePacked(gIndex));
    assembly {
        vault := create2(0, add(bytecode, 32), mload(bytecode), salt)
    }
    Vault(vault).initialize(manager, gIndex, collateral_, debt_, v1, amount_, v2Factory, WETH);
    allVaults.push(vault);
    return (vault, gIndex);
}
```

Listing 2.8: VaultFactory.sol

If a CDP vault is not initialized in the contract `VaultManager`, the minimum collateralization ratio (`MCRConfig`) of the collateral is zero. That cause the function `isValidCDP` always returns `true`.

```
function isValidCDP(address collateral_, address debt_, uint256 cAmount_, uint256 dAmount_)
     public view override returns (bool) {
```

```
    (uint256 collateralValueTimes100Point00000, uint256 debtValue) = _calculateValues(
        collateral_, debt_, cAmount_, dAmount_);

    uint mcr = getMCR(collateral_);
    uint cDecimals = IERC20Minimal(collateral_).decimals();

    uint256 debtValueAdjusted = debtValue / (10 ** cDecimals);

    // if the debt become obsolete
    return debtValueAdjusted == 0 ? true : collateralValueTimes100Point00000 /
        debtValueAdjusted >= mcr;
}
```

Listing 2.9: VaultManager.sol

Furthermore, since all vaults created by the `VaultFactory` are authorized to mint MTR stable coins, the bypass of `isValidCDP` may incur infinite MTR minted.

Impact Uninitialized CDP vaults may incur infinite MTR minted.

Suggestion Limit the vault creator to the contract `VaultManager`.

This issue was fixed by the commit C2.

2.2.5 Incorrect usage to the function `isValidCDP` may incur infinite MTR minted

Status Fixed

Description

The functions `borrowMore` and `borrowMoreNative` are designed to mint MTR stable coins. As shown in below code snippet, vaults invoke the function `isValidCDP` to check if the CDP is valid before minting MTR stable coins.

```
function borrowMore(
    uint256 cAmount_,
    uint256 dAmount_
  ) external override onlyVaultOwner {
    // get vault balance
    uint256 deposits = IERC20Minimal(collateral).balanceOf(address(this));
    // check position
    require(IVaultManager(manager).isValidCDP(collateral, debt, cAmount_+ deposits, dAmount_),
        "IP"); // Invalid Position
    // check rebased supply of stablecoin
    require(IVaultManager(manager).isValidSupply(dAmount_), "RB"); // Rebase limited mtr borrow
    // transfer collateral to the vault, manage collateral from there
    TransferHelper.safeTransferFrom(collateral, msg.sender, address(this), cAmount_);
    // mint mtr to the sender
    IStablecoin(debt).mintFromVault(factory, vaultId, msg.sender, dAmount_);
  }

  function borrowMoreNative(
    uint256 dAmount_
  ) external payable onlyVaultOwner {
    // get vault balance
    uint256 deposits = IERC20Minimal(WETH).balanceOf(address(this));
    // check position
    require(IVaultManager(manager).isValidCDP(collateral, debt, msg.value + deposits, dAmount_)
        , "IP"); // Invalid Position
```

```
      // check rebased supply of stablecoin
      require(IVaultManager(manager).isValidSupply(dAmount_), "RB"); // Rebase limited mtr borrow
      // wrap native currency
      IWETH(WETH).deposit{value: address(this).balance}();
      // mint mtr to the sender
      IStablecoin(debt).mintFromVault(factory, vaultId, msg.sender, dAmount_);
    }
```

Listing 2.10: Vault.sol

However, the last parameter of `isValidCDP` is set to `dAmount_` that is the amount of MTR stable coins users want to mint. Note that, the parameter `dAmount_` is controlled by users. Anyone can use a suitable `dAmount_` to invoke the function `borrowMore` or `borrowMoreNative` to mint MTR stable coins repeatedly.

Impact Infinite MTR stable coins minted.

Suggestion Use `getDebt()` + `dAmount_` to replace the `dAmount_` as the last parameter.

The commit C2 tried to fix this issue but it did not fix the function `borrowMoreNative` .

The commit C3 fixed this issue.

2.2.6 The global variable `borrow` is not be updated correctly

Status Fixed

Description

The global variable `borrow` in the contract `Vault` is designed to record how many MTR stable coins are minted (or borrowed). However, the functions `borrowMore` and `borrowMoreNative` do not update it.

```
  function borrowMore(
      uint256 cAmount_,
      uint256 dAmount_
    ) external override onlyVaultOwner {
      // get vault balance
      uint256 deposits = IERC20Minimal(collateral).balanceOf(address(this));
      // check position
      require(IVaultManager(manager).isValidCDP(collateral, debt, cAmount_+ deposits, dAmount_),
          "IP"); // Invalid Position
      // check rebased supply of stablecoin
      require(IVaultManager(manager).isValidSupply(dAmount_), "RB"); // Rebase limited mtr borrow
      // transfer collateral to the vault, manage collateral from there
      TransferHelper.safeTransferFrom(collateral, msg.sender, address(this), cAmount_);
      // mint mtr to the sender
      IStablecoin(debt).mintFromVault(factory, vaultId, msg.sender, dAmount_);
    }

    function borrowMoreNative(
      uint256 dAmount_
    ) external payable onlyVaultOwner {
      // get vault balance
      uint256 deposits = IERC20Minimal(WETH).balanceOf(address(this));
      // check position
      require(IVaultManager(manager).isValidCDP(collateral, debt, msg.value + deposits, dAmount_)
          , "IP"); // Invalid Position
      // check rebased supply of stablecoin
      require(IVaultManager(manager).isValidSupply(dAmount_), "RB"); // Rebase limited mtr borrow
```

```
        // wrap native currency
        IWETH(WETH).deposit{value: address(this).balance}();
        // mint mtr to the sender
        IStablecoin(debt).mintFromVault(factory, vaultId, msg.sender, dAmount_);
    }
```

Listing 2.11: Vault.sol

Impact Since the variable `borrow` is critical to vaults, the mistake make vaults can not work.

Suggestion Add codes to update `borrow` in the functions: `borrowMore` and `borrowMoreNative`.

This issue was fixed by the commit C2.

2.2.7 Users' collaterals may be locked in vaults

Status Fixed

Description

Users can invoke the function `closeVault` to repay all the borrowed MTR stable coins (including stability fee) and close their CDPs. The function as shown in below code snippet.

```
    function closeVault(uint256 amount_) external override onlyVaultOwner {
        // calculate debt with interest
        uint256 fee = _calculateFee();
        require(fee + borrow == amount_, "Vault: not enough balance to payback");
        // send MTR to the vault
        TransferHelper.safeTransferFrom(debt, msg.sender, address(this), amount_);
        // send fee to the pool
        uint256 left = _sendFee(debt, amount_, fee);
        // burn mtr debt with interest
        _burnMTRFromVault(left);
        // burn vault nft
        _burnV1FromVault();
        emit CloseVault(vaultId, amount_, fee);
        // self destruct the contract, send remaining balance if collateral is native currency
        selfdestruct(payable(msg.sender));
    }
```

Listing 2.12: Vault.sol

Note that, the function `closeVault` does not transfer the remaining collaterals to users before executing `selfdestruct`.

Impact Users' collaterals will be locked in vaults if they invoke the function `closeVault`.

Suggestion Add codes to transfer the remaining collaterals to users before executing `selfdestruct`.

This issue was fixed by the commit C3.

2.2.8 The potential reentrancy risk

Status Fixed

Description

The variable `borrow` is updated after the code: `TransferHelper.safeTransferFrom(collateral, msg.sender, address(this), cAmount_)`. If the collateral is a token with callback mechanism, such as ERC-777 tokens, the function will be reentered to borrow more debt token than expected.

The below codes come from the commit C3.

```
function borrowMore(uint256 cAmount_, uint256 dAmount_)
external
override
onlyVaultOwner
{
// get vault balance
uint256 deposits = IERC20Minimal(collateral).balanceOf(address(this));
// check position
require(
  IVaultManager(manager).isValidCDP(
    collateral,
    debt,
    cAmount_ + deposits,
    borrow + dAmount_
  ),
  "IP"
); // Invalid Position
// check rebased supply of stablecoin
require(IVaultManager(manager).isValidSupply(dAmount_), "RB"); // Rebase limited mtr borrow
// transfer collateral to the vault, manage collateral from there
TransferHelper.safeTransferFrom(
  collateral,
  msg.sender,
  address(this),
  cAmount_
);
// mint mtr to the sender
IStablecoin(debt).mintFromVault(factory, vaultId, msg.sender, dAmount_);
// set new borrow amount
borrow += dAmount_;
emit BorrowMore(vaultId, cAmount_, dAmount_, borrow);
}

function borrowMoreNative(uint256 dAmount_) external payable onlyVaultOwner {
// get vault balance
uint256 deposits = IERC20Minimal(WETH).balanceOf(address(this));
// check position
require(
  IVaultManager(manager).isValidCDP(
    collateral,
    debt,
    msg.value + deposits,
    borrow + dAmount_
  ),
  "IP"
); // Invalid Position
// check rebased supply of stablecoin
require(IVaultManager(manager).isValidSupply(dAmount_), "RB"); // Rebase limited mtr borrow
// wrap native currency
IWETH(WETH).deposit{ value: address(this).balance }();
// mint mtr to the sender
IStablecoin(debt).mintFromVault(factory, vaultId, msg.sender, dAmount_);
```

```
230 // set new borrow amount
231 borrow += dAmount_;
232 emit BorrowMore(vaultId, msg.value, dAmount_, borrow);
233}
```

Listing 2.13: Vault.sol

Impact There is a potential reentrancy risk that can be exploited to mint more MTR.

Suggestion Use reentrancyGuard [1], otherwise, never support tokens with callback mechanism as collaterals. **This issue was fixed by the commit** C5.

2.2.9 The minting cap may be bypassed

Status Fixed

Description

In order to fix the issue 2.2.8, the project moves borrow change (in line 198 and 227) before fund transfer. However, the validation of minting limit (in line 196 and 225), the external call (in line 200 and line 229), and the MTR mint (in line 207 and 231) is still the classic reentrant pattern. Therefore, there is a potential risk to bypass the MTR minting cap.

The below codes come from the commit C5.

```
178   function borrowMore(uint256 cAmount_, uint256 dAmount_)
179   external
180   override
181   onlyVaultOwner
182  {
183   // get vault balance
184   uint256 deposits = IERC20Minimal(collateral).balanceOf(address(this));
185   // check position
186   require(
187     IVaultManager(manager).isValidCDP(
188       collateral,
189       debt,
190       cAmount_ + deposits,
191       borrow + dAmount_
192     ),
193     "IP"
194   ); // Invalid Position
195   // check rebased supply of stablecoin
196   require(IVaultManager(manager).isValidSupply(dAmount_), "RB"); // Rebase limited mtr borrow
197   // set new borrow amount
198   borrow += dAmount_;
199   // transfer collateral to the vault, manage collateral from there
200   TransferHelper.safeTransferFrom(
201     collateral,
202     msg.sender,
203     address(this),
204     cAmount_
205   );
206   // mint mtr to the sender
```

[1] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/4a9cc8b4918ef3736229a5cc5a310bdc17bf759f/contracts/security/ReentrancyGuard.sol

```
    IStablecoin(debt).mintFromVault(factory, vaultId, msg.sender, dAmount_);
    emit BorrowMore(vaultId, cAmount_, dAmount_, borrow);
  }

  function borrowMoreNative(uint256 dAmount_) external payable onlyVaultOwner {
    // get vault balance
    uint256 deposits = IERC20Minimal(WETH).balanceOf(address(this));
    // check position
    require(
      IVaultManager(manager).isValidCDP(
        collateral,
        debt,
        msg.value + deposits,
        borrow + dAmount_
      ),
      "IP"
    ); // Invalid Position
    // check rebased supply of stablecoin
    require(IVaultManager(manager).isValidSupply(dAmount_), "RB"); // Rebase limited mtr borrow
    // set new borrow amount
    borrow += dAmount_;
    // wrap native currency
    IWETH(WETH).deposit{ value: address(this).balance }();
    // mint mtr to the sender
    IStablecoin(debt).mintFromVault(factory, vaultId, msg.sender, dAmount_);
    emit BorrowMore(vaultId, msg.value, dAmount_, borrow);
  }
```

Listing 2.14: Vault.sol

Impact The minting cap may be bypassed.

Suggestion Also put the MTR mint before fund transfer.

This issue was fixed by the commit C6.

The project party must ensure that the supported collateral token(s) have the function: transferFrom(0x23b872dd). That's because the `TransferHelper.safeTransferFrom` will not revert if the collateral token(s) do not implement the function: transferFrom(0x23b872dd).

2.2.10 Price manipulation attacks against FeeRoll contract

Status Fixed.

Description

The function `tradeCollaterals` in the following code snippets is used to trade fees distributed from vaults for STND tokens, which can support the value of STND in DeFi market.

There are two price manipulation methods that can cause the FeeRoll contract to lose collaterals.

- For example, the collateral token to be sold is WETH. The function `tradeCollateralls` will sell WETH for MTR and then for STND. The attack consists of three steps. The first step, an attacker borrows a huge amount of WETH via flashloan to swap for the MTR, then the WETH's price in the pool is manipulated to be very low. Second, the attacker invokes `tradeCollaterals` of the FeeRoll contract that cheaply sells WETH reserves for little MTR and then for little STND. Third, the attacker swaps his MTR for WETH back, which can profit from the trade that FeeRoll contract makes.

- The second method based on an assumption: there has no swap pool for a collateral token and MTR. For example, the collateral token is WETH. An attacker creates a swap pool for the WETH and MTR but add little liquidity, then the pool has a very large slippage. After that, the attack invokes `tradeCollateralls` of the FeeRoll contract that also cheaply sells WETH reserves for little MTR and then for little STND. Finally, the attacker swaps a small amount of MTR for the FeeRoll contract's collaterals (WETH).

The below codes come from the commit C6.

```
821   function tradeCollaterals() public {
822       // for all lp tokens in the collateral array
823       uint256 len = allCollaterals.length;
824       for (uint256 i = 0; i < len; ++i) {
825           tradeCollateral(allCollaterals[i]);
826       }
827   }
828
829   function getPathToStnd(address input) private view returns (address[] memory) {
830       address[] memory path = new address[](3);
831       path[0] = input;
832       path[1] = stablecoin;
833       path[2] = stnd;
834
835       return path;
836   }
837
838   function tradeCollateral (
839       address collateral
840   ) internal {
841       IUniswapV2Router01(router).swapExactTokensForTokens(
842           IERC20(collateral).balanceOf(address(this)),
843           0,
844           getPathToStnd(collateral),
845           dstnd,
846           block.timestamp + 20000000
847       );
848   }
```

Listing 2.15: FeeRoll.sol

Impact The issue incurs price manipulation attacks.

Suggestion

- Add a check `require(msg.sender == tx.origin)` in the function `tradeCollateralls` to ensure the caller is EOA.
- Add a check to make sure the existence of the swap pool between `input` and `stablecoin` in the function `getPathToStnd`
- Leverage the price oracle to implement a slippage check for `swapExactTokensForTokens` rather than filling 0 (in line 843).

Note that, the function `tradeLPs` also has this issue.

2.3 Additional Recommendation

2.3.1 Finish the function `mintFromVault`

Status Fixed

Description

All vaults mint MTR stable coins by invoking the function `mintFromVault` that is critical for the project. However, it does not seem to be done.

```
function mintFromVault(address factory, uint256 vaultId_, address to, uint256 amount) external
     override {
   require(hasRole(FACTORY_ROLE, factory), "IA");
   require(IVaultFactory(factory).getVault(vaultId_) == _msgSender(), "Meter: Not from Vault")
       ;
}
```

Listing 2.16: Meter.sol

Impact We cannot make sure it's safe.

Suggestion Finish the function `mintFromVault`.

This recommendation is adopted by the commit C2

2.3.2 Add more smart contracts in the audit list

Status Fixed

Description

As shown in the following code snippet, all fees in vaults are transferred to three potential accounts: `dividend`, `feeTo`, and `treasury`. If at least one of them is smart contract account, the smart contract codes should be audited to make sure the security of fees.

```
function _sendFee(
    address asset_,
    uint256 amount_,
    uint256 fee_
  ) internal returns (uint256 left) {
    address dividend = IVaultManager(manager).dividend();
    address feeTo = IVaultManager(manager).feeTo();
    address treasury = IVaultManager(manager).treasury();
    bool feeOn = feeTo != address(0);
    bool treasuryOn = treasury != address(0);
    bool dividendOn = dividend != address(0);
    // send fee to the pool
    if (feeOn) {
      if (dividendOn) {
        uint256 half = fee_ / 2;
        TransferHelper.safeTransfer(asset_, dividend, half);
        TransferHelper.safeTransfer(asset_, feeTo, half);
      } else if (dividendOn && treasuryOn) {
        uint256 third = fee_ / 3;
        TransferHelper.safeTransfer(asset_, dividend, third);
        TransferHelper.safeTransfer(asset_, feeTo, third);
        TransferHelper.safeTransfer(asset_, treasury, third);
      } else {
        TransferHelper.safeTransfer(asset_, feeTo, fee_);
      }
```

```
        }
        return amount_ - fee_;
    }
}
```

Listing 2.17: Vault.sol

Impact NA

Suggestion If at least one of accounts: `dividend`, `feeTo`, and `treasury` is smart contract account, add it into audit list.

2.3.3 Make the codes and comments consistent

Status Fixed

Description

As shown in the following codes, the comments in L75 says: "Check that the calling account has the burner role", while the codes do not force it.

```
74  function burn(uint256 amount) external override {
75      // Check that the calling account has the burner role
76      _burn(_msgSender(), amount);
77  }
```

Listing 2.18: Meter.sol

Impact N/A

Suggestion Make the codes and comments consistent.

This recommendation was adopted by the commit C5**.**