AUTONOMINT Protocol Audit Report

PLATFORM: SHERLOCK

Prepared by:OLA Hamid

Table of Contents

Protocol Summary

Protocol does X, Y, Z

Disclaimer

T makes all effort 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 by the team 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

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

Audit Details

Scope

Roles

Executive Summary

Issues found

Severity	Number of issues found
High	2
Medium	0
Low	1
Info	0
Total	0

Findings

High

AUDIT 1

TITLE

Centralisation Control Risk in the EIP712 implementation from the `CDS.sol` contract, Allowing an admin to choose centralised power that stops users from withdrawing thier own funds independently.

SUMMARY

The design of the the `withdraw` function in the `CDS.sol` contract is unusual and has a design flaw, i believe the intented Design was to allow all users calling the withdraw function in the `cds.sol` contract, go through a verification process that allow the `hashedAdminTwo` actor first verify that he is the signer. despite the the user that called the `deposit` function being the legitimate owner. This gives centralised power to the `hashedAdminTwo` actor and also it restrict the `withdraw` function to be called by only `hashedAdminTwo` actor which creates a centralisation risk and functionality distruction(only admin can withdraw).

ROOT CAUSE

The root cause to this vulnerability is caused by the intended design, majorly in the `verify` function in the `CDS.sol` contract. https://github.com/sherlock-audit/2024-11-autonomint/blob/0d324e04d4c0ca306e1ae4d4c65f0cb9d681751b/Blockchain/Blockchian/contracts/Core_logic/CDS.sol#L285 https://github.com/sherlock-audit/2024-11-autonomint/blob/0d324e04d4c0ca306e1ae4d4c65f0cb9d681751b/Blockchain/Blockchian/contracts/Core_logic/CDS.sol#L911

IMPACT

The Impact of the vulnerability is as follows:

1. Users cannot withdraw their own funds independently
2. All withdrawals must be approved by a specific admin (hashedAdminTwo)
3. If admin is unavailable or compromised, user funds are effectively locked
4. Admin has complete control over who can and cannot withdraw funds

Internal Precondition:

nil

External Precondition:

nil

ATTACK PATH

1. User deposits USDa by callong the `deposit` function.
2. When withdrawal period is reached, user attempts to withdraw.
3. User cannot generate valid admin signature.

POC

The check makes it impossible for regular users to withdraw function, even though the function checks msg.sender's deposit.

```
function withdraw(
    uint64 index,
    uint256 excessProfitCumulativeValue,
    uint256 nonce,
    bytes memory signature
) external payable nonReentrant {

    if (!_verify(FunctionName.CDS_WITHDRAW, excessProfitCumulativeValue,
nonce, "0x", signature))
        revert CDS_NotAnAdminTwo();


    if (cdsDetails[msg.sender].index < index)
        revert CDS_InvalidIndex();
    ...
}
```

MITIGATION

The mitigations for this vulnerability is to redesign the `verify` function in the cds Contract to do a check that it allows the deposited user that calls the withdraw function to be the signer.

- In this mitigation i verified the msg.sender instead of the admin2 actor.

```
    function _verify(
    FunctionName functionName,
    uint256 excessProfitCumulativeValue,
    uint256 nonce,
    bytes memory odosExecutionData,
    bytes memory signature
) private view returns (bool) {
    bytes32 digest = _hashTypedDataV4(
        keccak256(
            abi.encode(
                keccak256(
                    "Permit(uint256 excessProfitCumulativeValue,uint256
nonce)"
                ),
                excessProfitCumulativeValue,
                nonce
            )
        )
    );

    address signer = ECDSA.recover(digest, signature);
-            if (hashedSigner == hashedAdminTwo)
+             if (signer == msg.sender) {
            return true;
        } else {
            return false;
}
```

AUDIT 2

TITLE

Unprotected access control in the updateDownsideProtected function, allows anyone to Update the Leading to Fund Extraction or potential state effect mishandling.

SUMMARY

The updateDownsideProtected function is declared as external visibility, which allow external malicious users or adddresses to call it since there is no restriction, The function lacks proper access control, enabling any external account to modify the downsideProtected state variable, The modified state is then called in the _updateCurrentTotalCdsDepositedAmount function, the _updateCurrentTotalCdsDepositedAmount function is called in the withdraw an deposit functions. An attacker can repeatedly call the updateDownsideProtected function increasing the downsideProtected state variable, the _updateCurrentTotalCdsDepositedAmount function is eventually called this lead to a possible distruption of functionality or DOS(denial of services).

ROOT CAUSE

The root cause of the vulnerability is the lack of access control(onlyOwner) in the updateDownsideProtected function.

```
function updateDownsideProtected(uint128 downsideProtectedAmount) external
{
    downsideProtected += downsideProtectedAmount;
}
```

IMPACT

1. A malicious user can inflate the `updateDownsideProtected` function, increasing the `downsideProtected` state value to arbitrarily high levels, leading to inaccurate calculations in `_updateCurrentTotalCdsDepositedAmount()` function resulting to a Potential DOS.
2. distription of state variable `totalCdsDepositedAmountWithOptionFees` and `totalCdsDepositedAmount` accounts

Internal Precondition:

External Precondition:

ATTACK PATH

FIRST PATH

1. Attacker calls `updateDownsideProtected` function with large value.
2. When legitimate withdrawer occurs, `_updateCurrentTotalCdsDepositedAmount` will revert because the `downsideProtected` var is larger than the `totalCdsDepositedAmount`.

POC

- do the following: i changed the `_updateCurrentTotalCdsDepositedAmount` to public. my intended aim to proof is that an attacker can increase the value of `downsideProtected` higher than `totalCdsDepositedAmount` or `totalCdsDepositedAmountWithOptionFees` variables

```
    function testExpliotDownsideProtected() public {
        vm.startPrank(attacker);
        for (uint256 i = 0; i < 10 /*1000*/; i++) {
            cdsTester.updateDownsideProtected(1e8); // Repeated calls to
inflate the value
            //console.log("this is the updated value:",
cdsTester.downsideProtected);
        }
        vm.expectRevert();
        cdsTester._updateCurrentTotalCdsDepositedAmount();
    }
```

MITIGATION

introduce an onlyOwner modifier from openZepplin library to put retriction on the `updateDownsideProtected` function.

```
-    function updateDownsideProtected(uint128 downsideProtectedAmount) external {
+   function updateDownsideProtected(uint128 downsideProtectedAmount) external onlyOwner {
        downsideProtected += downsideProtectedAmount;
    }
```

LOW

AUDIT 3

TITLE

Missing `disableinitializers()` in constructor can lead to malicious takeover of the implementation contract

SUMMARY

This issue occured in these contracts: `borrowing.sol`, `borrowLiquidation.sol` , `CDS.sol` , `GlobalVariables.sol`, `Options.sol` ,`Treasury.sol` , `Abond_Token.sol` AND `USDa.sol` . Lets take the `Abond_Token.sol` contract as an example for description, the contract uses UUPS upgradable methods to allow upgrade in the future. The contract calls the initialize function as a function to set up the first time set-up,It is best practices to have the `_disableInitializers()` in the constructor.

ROOT CAUSE

The contracts mentioned above does not have `_disableInitializers()` function in it's contructor.