Vault Gaurdians Audit Report

Version 1.0

Ardeshir Gholami

March 16, 2024

Vault Gaurdians Security Review

Ardeshir Gholami

16 march 2024

Prepared by: Ardeshir Lead Auditors: - Patrick Collins, - Ardeshir :)

Table of Contents

- Table of Contents
- Protocol Summary
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-1] Potential for Asset Theft Through Share Minting
    * [H-2] Guardians can use `VaultShares::redeem` or `VaultShares::withdraw` functions to take out their intial input amount but the vault will still stay active
    * [H-3] Lack of UniswapV2 slippage protection in `UniswapAdapter::_uniswapInvest` enables frontrunners to steal profits
    * [H-4] Guardians can infinitely mint `VaultGuardianToken`s and take over DAO, stealing DAO fees and maliciously setting parameters
  - Medium

    * [M-1] Potential for Governance Attack Due to Low Quorum
  - Low
    * [L-1] in `VaultGaurdiansBase::becomeTokenGuardian` function will set wrong name and symbol for vault
    * [L-2] [L-2] Inconsistent Event Emission Order in `UniswapAdapter::UniswapInvested`
  - Informational
    * [I-1] Unused and Empty Interfaces in `InvestableUniverseAdapter.sol` and `IVaultGuardians.sol`

Protocol Summary

Core Functionality:

- Asset Management: The protocol allows users to deposit their assets into vaults, which are then managed by the guardians. The guardians are responsible for investing these assets in various financial instruments or DeFi strategies to maximize returns.
- Governance: The guardians, who are selected based on their contributions to the protocol, have the authority to make decisions regarding the allocation of assets, the selection of investment strategies, and the overall management of the vaults. This governance model ensures that the protocol's operations are transparent and decentralized.
- Risk Management: The protocol likely includes mechanisms for risk management, such as setting limits on the types of assets that can be invested in, diversifying investments across different financial instruments, and implementing safeguards against market volatility.
- Incentives: To encourage participation and ensure the long-term success of the protocol, the guardians and other participants may be incentivized through rewards or fees. This could include a portion of the investment returns, transaction fees, or tokens that represent a stake in the protocol.# Disclaimer

I make 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

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

Audit Details

The findings described in this document correspond the following commit hash:

```
hash
```

Scope

```
## Roles

# Executive Summary

## Issues found

| Severity | number of issues found |
| -------- | ---------------------- |
| High     | 3                      |
| Medium   | 1                      |
| Low      | 2                      |
| Info     | 1                      |
| Total    | 7                      |

# Findings
## High
### [H-1] Potential for Asset Theft Through Share Minting

**Description:** The `VaultShares::deposit` function utilizes the `
   _mint` function to create shares for Guardians and the DAO as a
   percentage cut (fee). This mechanism for reducing fees inadvertently
    allows guardians and the DAO to potentially misappropriate a
   significant portion of the protocol's assets. This risk arises
```

~~~
because guardians can deposit a large amount of assets (e.g., using flash loans) and then redeem them, effectively stealing most of the funds in the protocol.


**Impact:** The ability for guardians and the DAO to misappropriate assets through this mechanism poses a significant risk to the protocol's integrity and the security of its stakeholders. It could lead to a loss of trust in the protocol, damage its reputation, and result in financial losses for users. Additionally, it could undermine the protocol's governance model, as the DAO's actions could be influenced by guardians who have a vested interest in manipulating the system to their advantage.


**Proof of Concept:**
The following test suite demonstrates the issue:

1. A guardian starts the vault.
2. A user deposits 100 ETH.
3. The guardian deposits a large amount of money (e.g., 10,000 ETH) and redeems it immediately.
4. The user redeems their money, but it is significantly less than the deposit amount.

<details>

```javascript
function testGaurdianCanEmptyProtocolByDepositingAndRedeeming()
        public
        hasGuardian
    {
        //gaurdian starts usdc vault
        usdc.mint(mintAmount, guardian);
        vm.startPrank(guardian);
        usdc.approve(address(vaultGuardians), mintAmount);
        address tokenVault = vaultGuardians.becomeTokenGuardian(
            allocationData,
            usdc
        );
        usdcVaultShares = VaultShares(tokenVault);
        vm.stopPrank();
        //user deposits some money
        usdc.mint(mintAmount, user);
        uint256 userBalancebefore = usdc.balanceOf(user);

        vm.startPrank(user);
        usdc.approve(address(usdcVaultShares), mintAmount);
        usdcVaultShares.deposit(mintAmount, user);
        vm.stopPrank();
        console.log(userBalancebefore);
~~~

```
        //guardian deposits for himeself with a big amount to get a big
            guardian cut of shares
        usdc.mint(mintAmount * 100, guardian);
        vm.startPrank(guardian);
        usdc.approve(address(usdcVaultShares), mintAmount * 100);
        usdcVaultShares.deposit(mintAmount * 100, guardian);
        uint256 maxredeem = usdcVaultShares.maxRedeem(guardian);
        usdcVaultShares.redeem(maxredeem, guardian, guardian);
        vm.stopPrank();

        vm.startPrank(user);
        uint256 maxredeemUser = usdcVaultShares.maxRedeem(user);
        usdcVaultShares.redeem(maxredeemUser, user, user);
        vm.stopPrank();
        uint256 userBalanceAfter = usdc.balanceOf(user);
        console.log(userBalanceAfter);
    }
```

Recommended Mitigation: To mitigate this issue, the protocol's design should be reviewed to ensure that the share minting mechanism does not inadvertently enable asset misappropriation. This could involve adjusting the fee structure to prevent guardians from exploiting the system in this manner. Additionally, implementing stricter controls on the use of flash loans and the redemption of shares could help to safeguard against such exploits. One way to fix this issue is to instead of minting shares as guardian and dao cuts you can send them a part of deposits at the start.

[H-2] Guardians can use `VaultShares::redeem` or `VaultShares::withdraw` functions to take out their intial input amount but the vault will still stay active

Description: as protocol states when gaurdains want their intial deposit back they should call quitGuardian() function which will set the vault status to not active after paying the gaurdian their money. but gurdians are given shares of the vault when they initiate it and so they can use them to `withdraw` or `redeem` their asset leaving them still in control of the active vault but with no money in it.

Impact: The ability for guardians to withdraw their assets without deactivating the vault could lead to a situation where the vault appears to be operational but is not effectively managing its assets. This could result in inefficiencies or mismanagement of funds, potentially affecting the vault's performance and the interests of its stakeholders. Additionally, it might create confusion among users and developers about the vault's status and the guardians' roles within it. also if the guardian is a malicous persion he can cause loss for other users.

Proof of Concept: 1. gaurdian intiates the usdc vault by depositing the input amount via `becomeTokenGuardian` 2. then he uses the `redeem` function to take out his balance. 3. we check if the guardian has any assets left and if the vault is still active which returns true.

```
    function testGaurdianCanQuitWithRedeemFunction() public hasGuardian
         {
        usdc.mint(mintAmount, guardian);
        vm.startPrank(guardian);
        usdc.approve(address(vaultGuardians), mintAmount);
        address tokenVault = vaultGuardians.becomeTokenGuardian(
            allocationData,
            usdc
        );
        usdcVaultShares = VaultShares(tokenVault);

        uint256 balanceBefore = usdcVaultShares.balanceOf(guardian);
        uint256 maxredeem = usdcVaultShares.maxRedeem(guardian);
        console.log(maxredeem);

        usdcVaultShares.redeem(maxredeem, guardian, guardian);
        uint256 balanceAfter = usdcVaultShares.balanceOf(guardian);

        vm.stopPrank();
        assertEq(balanceAfter, 0);
        assert(usdcVaultShares.getIsActive());
    }
```

Recommended Mitigation: to mitigate this issue a notGaurdian modifier can be added to `redeem` and `withdraw` functions.

```
+     error VaultShares__NotActive();
      .
      .
      .
+     modifier notGuardian() {
+         if (msg.sender == i_guardian) {
+             revert VaultShares__GuardianCantUseRedeemOrWitdraw();
+         }
+         _;
+     }
      .
      .
      .
      function withdraw(
          uint256 assets,
          address receiver,
          address owner
      )
          public
          override(IERC4626, ERC4626)
+         notGaurdian
          divestThenInvest
          nonReentrant
          returns (uint256)
```

```
    {
        uint256 shares = super.withdraw(assets, receiver, owner);
        return shares;
    }
```

[H-3] Lack of UniswapV2 slippage protection in `UniswapAdapter::_uniswapInvest` enables frontrunners to steal profits

Description: In `UniswapAdapter::_uniswapInvest` the protocol swaps half of an ERC20 token so that they can invest in both sides of a Uniswap pool. It calls the `swapExactTokensForTokens` function of the `UnisapV2Router01` contract , which has two input parameters to note:

```
    function swapExactTokensForTokens(
        uint256 amountIn,
@>      uint256 amountOutMin,
        address[] calldata path,
        address to,
@>      uint256 deadline
    )
```

The parameter `amountOutMin` represents how much of the minimum number of tokens it expects to return. The `deadline` parameter represents when the transaction should expire.

As seen below, the `UniswapAdapter::_uniswapInvest` function sets those parameters to `0` and `block.timestamp`:

```
    uint256[] memory amounts = i_uniswapRouter.swapExactTokensForTokens
        (
        amountOfTokenToSwap,
@>      0,
        s_pathArray,
        address(this),
@>      block.timestamp
    );
```

Impact: This results in either of the following happening: - Anyone (e.g., a frontrunning bot) sees this transaction in the mempool, pulls a flashloan and swaps on Uniswap to tank the price before the swap happens, resulting in the protocol executing the swap at an unfavorable rate. - Due to the lack of a deadline, the node who gets this transaction could hold the transaction until they are able to profit from the guaranteed swap.

Proof of Concept:

1. User calls `VaultShares::deposit` with a vault that has a Uniswap allocation.

   1. This calls `_uniswapInvest` for a user to invest into Uniswap, and calls the router's `swapExactTokensForTokens` function.
2. In the mempool, a malicious user could:
   1. Hold onto this transaction which makes the Uniswap swap
   2. Take a flashloan out
   3. Make a major swap on Uniswap, greatly changing the price of the assets
   4. Execute the transaction that was being held, giving the protocol as little funds back as possible due to the `amountOutMin` value set to 0.

This could potentially allow malicious MEV users and frontrunners to drain balances.

Recommended Mitigation:

For the deadline issue, we recommend the following:

DeFi is a large landscape. For protocols that have sensitive investing parameters, add a custom parameter to the `deposit` function so the Vault Guardians protocol can account for the customizations of DeFi projects that it integrates with.

In the `deposit` function, consider allowing for custom data.

```
- function deposit(uint256 assets, address receiver) public override(
    ERC4626, IERC4626) isActive returns (uint256) {
+ function deposit(uint256 assets, address receiver, bytes customData)
    public override(ERC4626, IERC4626) isActive returns (uint256) {
```

This way, you could add a `deadline` to the Uniswap swap, and also allow for more DeFi custom integrations.

For the `amountOutMin` issue, we recommend one of the following:

1. Do a price check on something like a Chainlink price feed before making the swap, reverting if the rate is too unfavorable.
2. Only deposit 1 side of a Uniswap pool for liquidity. Don't make the swap at all. If a pool doesn't exist or has too low liquidity for a pair of ERC20s, don't allow investment in that pool.

Note that these recommendation require significant changes to the codebase.

[H-4] Guardians can infinitely mint `VaultGuardianTokens` and take over DAO, stealing DAO fees and maliciously setting parameters

Description: Becoming a guardian comes with the perk of getting minted Vault Guardian Tokens (vgTokens). Whenever a guardian successfully calls `VaultGuardiansBase::becomeGuardian` or `VaultGuardiansBase::becomeTokenGuardian`, `_becomeTokenGuardian` is executed, which mints the caller `i_vgToken`.

```
    function _becomeTokenGuardian(IERC20 token, VaultShares tokenVault)
        private returns (address) {
        s_guardians[msg.sender][token] = IVaultShares(address(
            tokenVault));
@>      i_vgToken.mint(msg.sender, s_guardianStakePrice);
        emit GuardianAdded(msg.sender, token);
        token.safeTransferFrom(msg.sender, address(this),
            s_guardianStakePrice);
        token.approve(address(tokenVault), s_guardianStakePrice);
        tokenVault.deposit(s_guardianStakePrice, msg.sender);
        return address(tokenVault);
    }
```

Guardians are also free to quit their role at any time, calling the `VaultGuardianBase::quitGuardian` function. The combination of minting vgTokens, and freely being able to quit, results in users being able to farm vgTokens at any time.

Impact: Assuming the token has no monetary value, the malicious guardian could accumulate tokens until they can overtake the DAO. Then, they could execute any of these functions of the `VaultGuardians` contract:

```
  "sweepErc20s(address)": "942d0ff9",
  "transferOwnership(address)": "f2fde38b",
  "updateGuardianAndDaoCut(uint256)": "9e8f72a4",
  "updateGuardianStakePrice(uint256)": "d16fe105",
```

Proof of Concept:

1. User becomes WETH guardian and is minted vgTokens.
2. User quits, is given back original WETH allocation.
3. User becomes WETH guardian with the same initial allocation.
4. Repeat to keep minting vgTokens indefinitely.

Code

Place the following code into `VaultGuardiansBaseTest.t.sol`

```
    function testDaoTakeover() public hasGuardian hasTokenGuardian {
        address maliciousGuardian = makeAddr("maliciousGuardian");
        uint256 startingVoterUsdcBalance = usdc.balanceOf(
            maliciousGuardian);
        uint256 startingVoterWethBalance = weth.balanceOf(
            maliciousGuardian);
        assertEq(startingVoterUsdcBalance, 0);
        assertEq(startingVoterWethBalance, 0);

        VaultGuardianGovernor governor = VaultGuardianGovernor(payable(vaultGuardians.owner()));
        VaultGuardianToken vgToken = VaultGuardianToken(address(governor.token()));

        // Flash loan the tokens, or just buy a bunch for 1 block
        weth.mint(mintAmount, maliciousGuardian); // The same amount as the other guardians
        uint256 startingMaliciousVGTokenBalance = vgToken.balanceOf(maliciousGuardian);
        uint256 startingRegularVGTokenBalance = vgToken.balanceOf(guardian);
        console.log("Malicious vgToken Balance:\t", startingMaliciousVGTokenBalance);
        console.log("Regular vgToken Balance:\t", startingRegularVGTokenBalance);

        // Malicious Guardian farms tokens
        vm.startPrank(maliciousGuardian);
        weth.approve(address(vaultGuardians), type(uint256).max);
        for (uint256 i; i < 10; i++) {
            address maliciousWethSharesVault = vaultGuardians.becomeGuardian(allocationData);
            IERC20(maliciousWethSharesVault).approve(
                address(vaultGuardians),
                IERC20(maliciousWethSharesVault).balanceOf(maliciousGuardian)
            );
            vaultGuardians.quitGuardian();
        }
        vm.stopPrank();

        uint256 endingMaliciousVGTokenBalance = vgToken.balanceOf(maliciousGuardian);
        uint256 endingRegularVGTokenBalance = vgToken.balanceOf(guardian);
        console.log("Malicious vgToken Balance:\t", endingMaliciousVGTokenBalance);
        console.log("Regular vgToken Balance:\t", endingRegularVGTokenBalance);
    }
```

Recommended Mitigation: There are a few options to fix this issue:

1. Mint vgTokens on a vesting schedule after a user becomes a guardian.
2. Burn vgTokens when a guardian quits.
3. Simply don't allocate vgTokens to guardians. Instead, mint the total supply on contract deployment. ## Medium ### [M-1] Potential for Governance Attack Due to Low Quorum

Description: The protocol's governance mechanism is configured with a quorum of 4%, which is considered low. This low quorum threshold could potentially make the governance process vulnerable to attacks, where a small group of malicious actors could manipulate the voting process to pass proposals that are not in the best interest of the protocol's stakeholders.

Impact: A low quorum threshold can lead to a situation where a small percentage of the total stake can influence the governance decisions, potentially leading to the adoption of harmful or exploitative proposals. This could undermine the protocol's integrity, erode trust among stakeholders, and result in financial losses or other negative outcomes for the protocol and its users.

Proof of Concept: To demonstrate this issue, one could calculate the minimum number of votes required to reach the 4% quorum based on the total staked amount. If this number is relatively small, it confirms the vulnerability of the governance process to attacks.

Recommended Mitigation: To mitigate this issue, the governance quorum should be increased to a higher percentage that better reflects the protocol's stakeholder base. This change would make it more difficult for a small group of actors to influence governance decisions. Additionally, implementing stricter voting requirements, such as a minimum stake requirement for voting, could further enhance the security and integrity of the governance process. The protocol documentation should also be updated to reflect the new quorum threshold and to provide guidance on the importance of participating in governance decisions.

Low

[L-1] in `VaultGaurdiansBase::becomeTokenGuardian` function will set wrong name and symbol for vault

Description: In `becomeTokenGuardian` if the input arg token is equal to `i_tokenOne` which is usdc we make a usdc vault using `IVaultShares::ConstructorData` and if its equal to `i_tokenTwo` which is LINK we should create a LINK token vault but the values for name and symbol put into `ConstructorData` are wrong.

```
    function becomeTokenGuardian(
        AllocationData memory allocationData,
        IERC20 token
    ) external onlyGuardian(i_weth) returns (address) {
        //slither-disable-next-line uninitialized-local
        VaultShares tokenVault;
        if (address(token) == address(i_tokenOne)) {
            tokenVault = new VaultShares(
                IVaultShares.ConstructorData({
                    asset: token,
                    vaultName: TOKEN_ONE_VAULT_NAME,
                    vaultSymbol: TOKEN_ONE_VAULT_SYMBOL,
                    guardian: msg.sender,
                    allocationData: allocationData,
                    aavePool: i_aavePool,
                    uniswapRouter: i_uniswapV2Router,
                    guardianAndDaoCut: s_guardianAndDaoCut,
                    vaultGuardians: address(this),
                    weth: address(i_weth),
                    usdc: address(i_tokenOne)
                })
            );
        } else if (address(token) == address(i_tokenTwo)) {
            tokenVault = new VaultShares(
                IVaultShares.ConstructorData({
                    asset: token,
@>                  vaultName: TOKEN_ONE_VAULT_NAME,
@>                  vaultSymbol: TOKEN_ONE_VAULT_SYMBOL,
                    guardian: msg.sender,
                    allocationData: allocationData,
                    aavePool: i_aavePool,
                    uniswapRouter: i_uniswapV2Router,
                    guardianAndDaoCut: s_guardianAndDaoCut,
                    vaultGuardians: address(this),
                    weth: address(i_weth),
                    usdc: address(i_tokenOne)
                })
            );
        } else {
            revert VaultGuardiansBase__NotApprovedToken(address(token));
        }
        return _becomeTokenGuardian(token, tokenVault);
    }
```

Impact: this will cause confusion and may cause confusion, because users might try to send usdc token to Link vaults. **Proof of Concept:** add this to your test suit, and run `forge test --mt testBecomeTokenGuardianTokentwoName` the assertion will fail.

```
    function testBecomeTokenGuardianTokentwoName() public hasGuardian {
        link.mint(mintAmount, guardian);
        vm.startPrank(guardian);
        link.approve(address(vaultGuardians), mintAmount);
        address tokenVault = vaultGuardians.becomeTokenGuardian(
            allocationData,
            link
        );
        linkVaultShares = VaultShares(tokenVault);
        vm.stopPrank();

        assertEq(linkVaultShares.name(), vaultGuardians.
           TOKEN_TWO_VAULT_NAME());
        assertEq(
            linkVaultShares.symbol(),
            vaultGuardians.TOKEN_TWO_VAULT_SYMBOL()
        );
    }
```

Recommended Mitigation: change the name and symbol to correct values.

```
    function becomeTokenGuardian(
        AllocationData memory allocationData,
        IERC20 token
    ) external onlyGuardian(i_weth) returns (address) {
        //slither-disable-next-line uninitialized-local
        VaultShares tokenVault;
        if (address(token) == address(i_tokenOne)) {
            tokenVault = new VaultShares(
                IVaultShares.ConstructorData({
                    asset: token,
                    vaultName: TOKEN_ONE_VAULT_NAME,
                    vaultSymbol: TOKEN_ONE_VAULT_SYMBOL,
                    guardian: msg.sender,
                    allocationData: allocationData,
                    aavePool: i_aavePool,
                    uniswapRouter: i_uniswapV2Router,
                    guardianAndDaoCut: s_guardianAndDaoCut,
                    vaultGuardians: address(this),
                    weth: address(i_weth),
                    usdc: address(i_tokenOne)
                })
            );
        } else if (address(token) == address(i_tokenTwo)) {
            tokenVault = new VaultShares(
                IVaultShares.ConstructorData({
                    asset: token,
-                   vaultName: TOKEN_ONE_VAULT_NAME,
-                   vaultSymbol: TOKEN_ONE_VAULT_SYMBOL,
+                   vaultName: TOKEN_TWO_VAULT_NAME,
+                   vaultSymbol: TOKEN_TWO_VAULT_SYMBOL,
                    guardian: msg.sender,
                    allocationData: allocationData,
                    aavePool: i_aavePool,
                    uniswapRouter: i_uniswapV2Router,
                    guardianAndDaoCut: s_guardianAndDaoCut,
                    vaultGuardians: address(this),
                    weth: address(i_weth),
                    usdc: address(i_tokenOne)
                })
            );
        } else {
            revert VaultGuardiansBase__NotApprovedToken(address(token))
```

```
                ;
            }
            return _becomeTokenGuardian(token, tokenVault);
        }
```

[L-2] [L-2] Inconsistent Event Emission Order in `UniswapAdapter::UniswapInvested`

Description: In the `UniswapAdapter::_uniswapInvest` function, when the input ERC20 token is set to WETH, the `UniswapInvested` event is emitted with the parameters in the order of `tokenAmount`, `counterPartyTokenAmount`, and `liquidity`. However, in the event declaration, the order is reversed, with `counterPartyTokenAmount` listed before `tokenAmount`. This discrepancy between the event emission and its declaration can lead to confusion and potential misinterpretation of the event data.

Impact: The inconsistency in the event emission order can affect the readability and usability of the contract's event logs. Developers and users who rely on these logs for monitoring or debugging purposes might misinterpret the data, leading to incorrect conclusions about the contract's behavior. This could potentially impact the contract's security, as it might obscure the true flow of assets or the contract's state changes.

Proof of Concept: In the `UniswapAdapter::_uniswapInvest` function first we set the `counterPartyToken` variable using the inline if statement if the `token` is equal to weth the `counterPartyToken` will be set to `i_tokenOne` which is usdc. but in event declaration we can see that the weth amount is the second parameter.

```
@>   event UniswapInvested(
         uint256 tokenAmount,
         uint256 wethAmount,
         uint256 liquidity
     );
.
.
.

     function _uniswapInvest(IERC20 token, uint256 amount) internal {
@>       IERC20 counterPartyToken = token == i_weth ? i_tokenOne :
    i_weth;
         .
         .
         .
@>       emit UniswapInvested(tokenAmount, counterPartyTokenAmount,
    liquidity);
 }
```

Recommended Mitigation: To address this issue, the event declaration for `UniswapInvested` should be updated to match the order of parameters in the event emission. This change will ensure consistency and clarity in the event logs, making it easier for developers and users to accurately interpret the contract's behavior. Additionally, it's advisable to review and update any documentation or code comments that reference the `UniswapInvested` event to reflect the corrected order of parameters.

```
     function _uniswapInvest(IERC20 token, uint256 amount) internal {
         .
         .
         .
+        if (token==i_weth){
+            emit UniswapInvested(counterPartyTokenAmount, tokenAmount,
    liquidity);
+        }else{
+            emit UniswapInvested(tokenAmount, counterPartyTokenAmount,
    liquidity);
+        }
-        emit UniswapInvested(tokenAmount, counterPartyTokenAmount,
    liquidity);
  }
```

Informational

[I-1] Unused and Empty Interfaces in `InvestableUniverseAdapter.sol` and `IVaultGuardians.sol`

Description: The `InvestableUniverseAdapter.sol` and `IVaultGuardians.sol` interfaces are present in the codebase but are empty and not utilized anywhere within the contracts. This situation can lead to confusion and potential misuse, as developers might assume these interfaces are meant to be implemented or used, only to find out they are not.

Impact: The presence of unused and empty interfaces can lead to a cluttered and confusing codebase, making it harder for developers to understand the contract's functionality and intent. It can also introduce potential security risks if future developers mistakenly believe these interfaces are part of the contract's security mechanisms.