Falcon Security Review

Pashov Audit Group

Conducted by: Ch_301, peanuts, zark, Udsen

February 17th 2025 - February 21st 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **falconfinance/falcon-contracts-evm** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Falcon

Falcon is a stablecoin system centered around USDf, a synthetic dollar ERC20 token, and sUSDf, an ERC4626 vault for staking USDf to earn yield, with FalconPosition, a unique ERC721 NFT, enabling time-bound, boosted yield positions. The platform also includes auxiliary contracts like FalconBundler for streamlined transactions, PreCollateralizedMinter for minting USDf, StakingRewardsDistributor for yield distribution, and USDfSilo for holding USDf during staking cooldown.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - 9c34a242ae6c39e2054d5e3bb62e44328339aaa1

fixes review commit hash - ee3655c8f747718a2b95d99d78500d71c4841664

Scope

The following smart contracts were in scope of the audit:

- `ClassicMinterV1`
- `FalconBundler`
- `FalconPosition`
- `PreCollateralizedMinter`
- `StakedUSDf`
- `StakingRewardsDistributor`
- `USDf`
- `USDfSilo`

7. Executive Summary

Over the course of the security review, Ch_301, peanuts, zark, Udsen engaged with Falcon to review Falcon. In this period of time a total of **12** issues were uncovered.

Protocol Summary

Protocol Name	Falcon
Repository	https://github.com/falconfinance/falcon-contracts-evm
Date	February 17th 2025 - February 21st 2025
Protocol Type	Stablecoin

Findings Count

Severity	**Amount**
Medium	2
Low	10
Total Findings	**12**

Summary of Findings

ID	Title	Severity	Status
[M-01]	Last withdrawal can be DoSed	Medium	Acknowledged
[M-02]	Shares cannot be minted on a deposit attack	Medium	Resolved
[L-01]	Redistribution in redistributeLockedAmount() may clash	Low	Acknowledged
[L-02]	Incorrect event emission	Low	Resolved
[L-03]	CooldownEnd pushed longer when cooldownAssets() is called	Low	Acknowledged
[L-04]	Low dollar price tokens can not be used as collateral in ClassicMinterV1	Low	Acknowledged
[L-05]	callerFundedMint() does not work with fee-on-transfer tokens	Low	Acknowledged
[L-06]	Restricted users can call the StakedUSDf::unstake	Low	Resolved
[L-07]	Changing vestingPeriod corrupts totalAssets returned value	Low	Resolved
[L-08]	Inconsistent unstake behavior after cooldown duration reduction	Low	Resolved
[L-09]	Pausing StakingRewardsDistributor prevents reward distribution	Low	Acknowledged
[L-10]	Restricted users can bypass asset confiscation via transfers	Low	Resolved

8. Findings

8.1. Medium Findings

[M-01] Last withdrawal can be DoSed

Severity

Impact: High

Likelihood: Low

Description

In `StakedUSDf.sol`, every withdrawal is performs a check (`_checkMinShares`) that enforces that if `totalSupply()` is non-zero, it must be greater than `MIN_SHARES == 1e18`. This restriction creates a DoS vulnerability for the last user attempting to withdraw their tokens.

An attacker can frontrun the last withdrawal transaction by depositing a small amount, causing `_checkMinShares` to revert the legitimate withdrawal. After successfully executing the DoS, the attacker can then backrun their own withdrawal, ensuring that the legitimate user remains unable to withdraw their tokens **indefinitely and at no cost**.

```
function _withdraw(
        address caller,
        address receiver,
        address owner,
        uint256 assets,
        uint256 shares
    )
        internal
        override
    {
        _checkZeroAmount(assets);
        _checkZeroAmount(shares);
        _checkRestricted(caller);
        _checkRestricted(receiver);
        _checkRestricted(owner);

        super._withdraw(caller, receiver, owner, assets, shares);
        _checkMinShares(); // <@ audit
    }
```

The impact of this vulnerability is that an attacker can permanently prevent the last user from withdrawing their funds by frontrunning with small deposits, effectively locking the victim's assets in the vault with no cost for any amount of time he wants.

In order to PoC this issue, you are kindly requested to create a new file in `falcon-contracts-evm/test` folder, name it `sUSDfLastWithdrawal.t.sol`, and paste the following smart contract inside. Then run `forge test --mt testDoSLastWithdrawal -v` to replicate the scenario:

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.8.28;

import {MockERC20} from "./mocks/MockERC20.sol";
import {SigUtils} from "./utils/SigUtils.sol";
import {ClassicMinterV1} from "src/ClassicMinterV1.sol";
import {StakedUSDf} from "src/StakedUSDf.sol";

import {StakingRewardsDistributor} from "src/StakingRewardsDistributor.sol";
import {USDf} from "src/USDf.sol";
import {USDfSilo} from "src/USDfSilo.sol";
import {IStakedUSDf} from "src/interfaces/IStakedUSDf.sol";

import
   {TransparentUpgradeableProxy} from "@openzeppelin/contracts/proxy/transparent/Trans
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {Test, console2} from "forge-std/Test.sol";

contract sUSDfLastWithdrawal is Test {
    // State variables
    USDf public usdf;
    StakedUSDf public stakedUsdf;
    ClassicMinterV1 public minter;
    USDfSilo public silo;
    StakingRewardsDistributor public distributor;
    MockERC20 public token;

    // Test accounts
    address internal admin;
    address internal operator;
    address internal treasury;
    address internal user1;
    address internal attacker;
    address internal backendSigner;
    uint256 internal backendPrivateKey;

    // Constants
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");
    bytes32 public constant REWARDER_ROLE = keccak256("REWARDER_ROLE");
    bytes32 public constant OPERATOR_ROLE = keccak256("OPERATOR_ROLE");

    // SigUtils
    SigUtils internal sigUtils;
    bytes32 internal classicMinterDomainSeparator;

    function setUp() public {
        // Setup test accounts
        admin = makeAddr("admin");
        operator = makeAddr("operator");
        treasury = makeAddr("treasury");
        user1 = makeAddr("user1");
        attacker = makeAddr("attacker");
        backendPrivateKey = 0x1234;
        backendSigner = vm.addr(backendPrivateKey);

        vm.startPrank(admin);

        // Deploy implementations
        USDf usdfImpl = new USDf();
        StakedUSDf stakedUsdfImpl = new StakedUSDf(treasury);

        // Initialize data for proxies
        bytes memory usdfData = abi.encodeWithSelector
          (USDf.initialize.selector, admin);

        // Deploy USDf proxy
        TransparentUpgradeableProxy usdfProxy = new TransparentUpgradeableProxy
```

```
          (address(usdfImpl), admin, usdfData);
        usdf = USDf(address(usdfProxy));

        silo = new USDfSilo(address(usdf));

        // Deploy mock token
        token = new MockERC20("Mock Token", "MTK");

        // Deploy StakedUSDf proxy
        bytes memory stakedUsdfData = abi.encodeWithSelector(
            StakedUSDf.initialize.selector,
            usdf,
            admin,
            silo,
            1 days, // vesting period
            1 days // 0 cooldown duration
        );
        TransparentUpgradeableProxy stakedUsdfProxy =
            new TransparentUpgradeableProxy(address
              (stakedUsdfImpl), admin, stakedUsdfData);
        stakedUsdf = StakedUSDf(address(stakedUsdfProxy));

        // Deploy StakingRewardsDistributor
        distributor = new StakingRewardsDistributor(stakedUsdf, IERC20(address
          (usdf)), admin, operator);

        // Deploy minter with treasury as fee recipient
        minter = new ClassicMinterV1(admin, address
          (usdf), treasury, backendSigner);

        // Setup basic roles
        usdf.grantRole(MINTER_ROLE, admin);
        usdf.grantRole(MINTER_ROLE, address(minter));

        // Grant REWARDER_ROLE to distributor
        stakedUsdf.grantRole(REWARDER_ROLE, address(distributor));

        stakedUsdf.setCooldownDuration(0); // NO COOLDOWN.

        // Mint `user1` and `attacker` some USDf.
        usdf.mint(address(distributor), 1000e18);
        usdf.mint(address(attacker), 2000e18);
        usdf.mint(address(user1), 2000e18);

        vm.stopPrank();
    }

    function testDoSLastWithdrawal() public {
        // User1 deposits into staking contract.
        vm.startPrank(user1);
        usdf.approve(address(stakedUsdf), 1000e18);
        console2.log("User1 deposits 1000e18 USDf.");
        stakedUsdf.deposit(1000e18, user1);
        vm.stopPrank();

        // At this point we can assume that times passes and only one depositor
        // has remained on the vault, the last user who tries to withdraw fully out of
        // Attacker has seen the tx of user1 and will DoSed it with no cost.

        console2.log
          ("User1 submits transaction to withdraw fully out of the vault now.");

        // Here, attacker frontruns user1's full withdrawal by depositing just
        // enough so `MinSharesViolation` error to be triggered.
        vm.startPrank(attacker);
        console2.log("Attacker frontruns him and deposit 0.5e18 USDf.");
        usdf.approve(address(stakedUsdf), 0.5e18);
        stakedUsdf.deposit(0.5e18, attacker);
        vm.stopPrank();

        // User1 is the last withdrawer and wants to withdraw fully but his
        // withdraw will revert.
        vm.startPrank(user1);
        uint256 allShares = stakedUsdf.balanceOf(user1);
        console2.log
          ("User1 withdrawal is reverting due to `MinSharesViolation` error.");
        vm.expectRevert();
        stakedUsdf.redeem(allShares ,user1, user1);
        vm.stopPrank();

        // Attacker backruns the tx with his withdrawal and at no cost, he has
        // effectivelly DoSed the withdraw of the last depositor of the vault with jus
        vm.startPrank(attacker);
        console2.log
          ("Attacker withdraws his 0.5e18 USDf and effectivelly has DoSed the last wit
        stakedUsdf.withdraw(0.5e18, attacker, attacker);
        vm.stopPrank();

        // The result is that the last user's withdrawal of the vault can be
        // DoSed for ever and at no cost.
    }
}
```

Recommendations

Consider not checking the minimum shares upon withdrawals.

[M-02] Shares cannot be minted on a deposit attack

Severity

Impact: High

Likelihood: Low

Description

When users deposit USDf into the stakedUSDf contract, they get back sUSDf. The calculation follows ERC4626, as so:

```
function _convertToShares(
    uint256assets,
    Math.Roundingrounding
  ) internal view virtual returns (uint256
        return assets.mulDiv(totalSupply() + 10 ** _decimalsOffset
          (), totalAssets() + 1, rounding);
    }
```

If the user deposits the first 100e18 USDf, he will get `assets * totalSupply() + 1 / totalAssets() + 1` = 1e18 * (0+1) / (0+1) = 1e18 shares

There is a `MIN_SHARES` check during deposit to ensure that the shares cannot be below 1e18.

```
function _checkMinShares() internal view {
        uint256 supply = totalSupply();
>       if (supply > 0 && supply < MIN_SHARES) {
            revert MinSharesViolation();
        }
    }

    function _deposit(
      addresscaller,
      addressreceiver,
      uint256assets,
      uint256shares
    ) internal override {
        _checkZeroAmount(assets);
        _checkZeroAmount(shares);
        _checkRestricted(caller);
        _checkRestricted(receiver);

        super._deposit(caller, receiver, assets, shares);
        _checkMinShares();
    }
```

The issue with this check is that a malicious user can directly deposit 1e18 USDf inside the stakedUSDf contract before anyone calls `deposit()`, making `totalAssets() = 1e18` and `totalSupply() = 0`

When the shares are calculated, `assets * totalSupply() + 1 / totalAssets() + 1` , 1e18 * 1 / (1e18 + 1) = 1, and even if more assets are deposited for the first time, eg 1e20, the shares returned will be less than 1e18, which will invoke the `ZeroAmount` error.

The test below describes the direct deposit attack, append under deposit.t.sol.

```
function test2_deposit() public {
        // User deposits USDf directly into the stakedUSDf contract, resulting
        // in "ZeroAmount" issue

        uint mintAmount = 1e25;

        vm.startPrank(user1);
        deal(address(usdf), user1, mintAmount);
        usdf.approve(address(stakedUSDf), mintAmount);
        // Directly deposit 1e18 worth of USDf before any `deposit()` is called
        usdf.transfer(address(stakedUSDf), 1e18);
        console2.log("USDF BALANCE of user:", usdf.balanceOf(address
          (stakedUSDf)));
        console2.log("SUSDF TOTALSUPPLY of user", stakedUSDf.totalSupply());
        // If this number is changed from 1e18 -> 1e25 , all will fail because
        // of "MIN_SHARE_VIOLATION"
        stakedUSDf.deposit(1e18, user1);
        vm.stopPrank();
        console2.log("USDF BALANCE TREASURY:", usdf.balanceOf(address
          (stakedUSDf.TREASURY())));
        console2.log("USDF BALANCE in contract:", usdf.balanceOf(address
          (stakedUSDf)));
        console2.log("SUSDF TOTALSUPPLY in contract", stakedUSDf.totalSupply());

        console2.log("USDF BALANCE of user:", usdf.balanceOf(address(user1)));
        console2.log("SUSDF TOTALSUPPLY of user", stakedUSDf.balanceOf(user1));
    }
```

Recommendations

To ensure this doesn't happen, sweep all the assets inside the contract if `totalSupply() == 0`. Override the deposit function instead of the `_deposit()` since in the deposit function will calculate `previewDeposit()` before calling `_deposit()`, which will return 0 for the above attack.

```
Append in stakedUSDf.sol:

     function deposit
       (uint256 assets, address receiver) public override returns (uint256) {
        if(IERC20(asset()).balanceOf(address(this)) != 0 && totalSupply
          () == 0) {
            SafeERC20.safeTransfer(IERC20(asset()), TREASURY,  IERC20(asset
              ()).balanceOf(address(this)));
        }
        super.deposit(assets, receiver);
     }
```

8.2. Low Findings

[L-01] Redistribution in `redistributeLockedAmount()` may clash

When `stakedUSDf.redistributeLockedAmount()` is called, if `burnShares` is true, it will burn the shares of `from` and call `_updateVestingAmount` to distribute USDf to all the shareholders.

```
function redistributeLockedAmount
   (address from, bool burnShares) external onlyRole(DEFAULT_ADMIN_ROLE) {
        // Redistribute only when `from` is restricted
        require(isRestricted[from], AddressNotRestricted(from));

        uint256 amountToDistribute = balanceOf(from);
        _checkZeroAmount(amountToDistribute);

        if (burnShares) {
            uint256 usdfToVest = previewRedeem(amountToDistribute);
>           _burn(from, amountToDistribute);
>           _updateVestingAmount(usdfToVest);
        } else {
            _transfer(from, TREASURY, amountToDistribute);
        }

        emit LockedAmountRedistributed(from, burnShares, amountToDistribute);
    }
```

The issue is that `transferInRewards()` also uses the same `updateVestingAmount()` function

```
function transferInRewards(uint256 amount) external onlyRole(REWARDER_ROLE) {
        require(totalSupply() > 0, MinSharesViolation());

        _updateVestingAmount(amount);
        IERC20(asset()).safeTransferFrom(msg.sender, address(this), amount);
        emit RewardsReceived(amount);
    }
```

If `vestingPeriod` > 0, then `getUnvestedAmount()` needs to be zero for `_updateVestingAmount()` to work.

```
function _updateVestingAmount(uint256 newVestingAmount) internal {
        _checkZeroAmount(newVestingAmount);
        require(getUnvestedAmount() == 0, RewardsStillVesting());
        vestingAmount = newVestingAmount;
        lastDistributionTimestamp = uint40(block.timestamp);
    }
```

For example, if `vestingPeriod` is 1 days, and the rewarder just transferred in some rewards, the admin cannot call `redistributeLockedAmount()` to redistribute the burned sUSDf shares until 1 day later. Also, the admin cannot call `redistributeLockedAmount()` more than once within a vestingPeriod, which is an issue if the admin intends to burn more than 1 user's sUSDf tokens and distribute them.

Recommendations:

Consider removing the `_updateVestingAmount()` function when burning sUSDf shares so the USDf tokens in the contract is immediately distributed to the share holders.

```
if (burnShares) {
            uint256 usdfToVest = previewRedeem(amountToDistribute);
            _burn(from, amountToDistribute);
-           _updateVestingAmount(usdfToVest);
        } else {
```

[L-02] Incorrect event emission

In `FalconPosition::withdraw()`, the function may be called by an authorized user rather than the actual owner of the NFT staking position. However, the emitted event `PositionClosed` incorrectly assumes that `msg.sender` is always the owner of the staking position, leading to an inaccurate event emission when the `msg.sender` is an authorized address.

```
event PositionClosed(
      uint256indexedtokenId,
      addressindexedowner,
      uint256principal,
      uint256duration
    );

    function withdraw(uint256 tokenId) external nonReentrant {
        _checkAuthorized(_ownerOf(tokenId), msg.sender, tokenId);

       // ...

        emit PositionClosed
        //(tokenId, msg.sender, totalAmount - tokensOwed, position.duration); // <@ au
        emit FeesCollected(tokenId, tokensOwed);
    }
```

Consider storing the actual owner of the staking position in memory and use it in the event emission.

[L-03] `CooldownEnd` pushed longer when `cooldownAssets()` is called

In `StakedUSDf.sol`, when a user initiates a withdrawal while the cooldown mechanism is active, they must wait for `cooldownDuration` before accessing their assets. However, if the user calls `cooldownAssets` again before the cooldown period expires, all previously cooled-down assets are locked again for a new cooldown period, resetting `cooldownEnd` to `block.timestamp` + `cooldownDuration.

```
function cooldownAssets(
      uint256assets,
      addressowner
    ) external ensureCooldownOn returns (uint256 shares
        cooldowns[owner].cooldownEnd = uint104
          (block.timestamp) + cooldownDuration;
        cooldowns[owner].underlyingAmount += uint152(assets);

        shares = super.withdraw(assets, address(silo), owner);
    }
```

The impact of this vulnerability is that users attempting to withdraw assets may experience indefinite delays if they repeatedly call `cooldownAssets`, as previously cooled-down assets will continue to be locked with each new request. To prevent resetting the cooldown for previously unstaked assets, modify the logic so that only newly added assets trigger a new cooldown, while maintaining the original cooldown period for assets that were already cooling down.

[L-04] Low dollar price tokens can not be used as collateral in `ClassicMinterV1`

In the `ClassicMinterV1::_mintAmount()` function, the calculation for minting `USDf` tokens uses a `PRICE_SCALE` of `10_000`. This means that tokens with a price lower than `$0.00009` will not be able to be used since their price in 4 decimals scale will need to be `>1` and Solidity doesn't support decimals numbers. For example, a token whose price is `$0.00009`, in 4 decimals its price is `0.00009e4` which is equal to `0.9` and such a value can not be given to the `params.price` variable.

```
function _mintAmount(
        uint256depositAmount,
        addresstoken,
        uint256price
    ) internal view returns (uint256
        uint256 collateralScale = 10 ** IERC20Metadata(token).decimals();
        uint256 scaledAmount = (depositAmount * price * _usdfScale);
        uint256 scaledPriceAmount = scaledAmount / PRICE_SCALE;
        return scaledPriceAmount / collateralScale;
    }
```

There is, actually, a lot of high market cap cryptocurrencies with such low prices like Shiba, BitTorrent and more so to mitigate this issue, consider increasing the precision scale of price from `1e4` to a bigger number like `1e8` or `1e18`.

[L-05] `callerFundedMint()` does not work with fee-on-transfer tokens

The `ClassicMinterV1::callerFundedMint` function is used to `mint USDf` tokens by depositing collateral tokens, after verifying signature of a `MintParams` message signed by the `BACKEND_SIGNER_ROLE address` of the protocol. After the `signature verification` the collateral amount is transferred to the `treasury contract` as shown below:

```
// Transfer collateral to treasury
        IERC20(params.token).safeTransferFrom
          (params.caller, treasury, depositAmount);
```

But the issue here is if the `params.token (collateral)` is a fee on transfer token then the deposited amount to the treasury will be less than the `depositAmount`. But the `USDf tokens` are minted on the `depositAmount` itself as shown below:

```
uint256 amountToMint = _mintAmount
        (depositAmount, params.token, params.price);
```

As a result the protocol will lose funds as less collateral tokens are deposited to the treasury compared to the amount of `USDf` minted to the `recipient`.

Hence it is recommended to calculate the `actual transferred collateral amount` to the `ClassicMinterV1` contract and use that amount to calculate the `USDf token amount to mint`.

[L-06] Restricted users can call the `StakedUSDf::unstake`

In the `StakedUSDf`, if a user is restricted after he has called the `StakedUSDf::cooldownAssets`, he is able to unstake his rewards after the cooldown period is over since the `StakedUSDf::unstake` does not check whether the `msg.sender` or the `reciever` is restricted.

```
function unstake(address receiver) external {
        _checkZeroAddress(receiver);

        UserCooldown storage userCooldown = cooldowns[msg.sender];
        uint256 assets = userCooldown.underlyingAmount;

        require(block.timestamp >= userCooldown.cooldownEnd, CooldownNotEnded
          ());

        userCooldown.cooldownEnd = 0;
        userCooldown.underlyingAmount = 0;

        silo.withdraw(receiver, assets);
    }
```

Since the `StakedUSDf::unstake` is called in the context of the `StakedUSDf` contract, this allows the malicioius user to unstake his funds even after he is tagged as `restricted`.

Hence recommended to check whether the `msg.sender` and `receiver` is restricted in the `StakedUSDf::unstake` execution and revert the transaction if either of the users are in the restricted state.

[L-07] Changing `vestingPeriod` corrupts `totalAssets` returned value

The `StakedUSDf.sol` contract is a ERC4626 vault for staking USDf to accrue yield, with linear vesting for reward deposits. the `StakingRewardsDistributor.sol` will trigger `transferInRewards()` function in `StakedUSDf.sol` to transfer the reward and update the vesting amount.

In case the admin calls `setVestingPeriod()` function to update the duration of the vesting period and the `getUnvestedAmount()` still returns a non-zero value. it will corrupt the current reward vesting or it can lead to loss/profit to the current stackers because the `totalAssets()` function uses the returned value from `getUnvestedAmount()` which will be directly affected by any update to `vestingPeriod`.

It also could lead to blocking multiple functions in `StakedUSDf.sol` contract e.g. deposit, mint, withdraw, redeem... these functions will revert with panic **arithmetic underflow or overflow**.

Recommendations:

You can revert in case `getUnvestedAmount() > 0`

```
function _setVestingPeriod(uint32 newPeriod) internal {
        uint32 oldVestingPeriod = vestingPeriod;
        require(newPeriod <= MAX_VESTING_PERIOD, DurationExceedsMax());
        require(oldVestingPeriod != newPeriod, DurationNotChanged());
        require(newPeriod > 0 || cooldownDuration > 0, ExpectedCooldownOn());
+       require(getUnvestedAmount() == 0, ExpectedCooldownOn());
```

[L-08] Inconsistent `unstake` behavior after cooldown duration reduction

The `StakedUSDf` contract allows users to initiate a cooldown period for their staked assets (either shares or underlying assets) using the `cooldownAssets` or `cooldownShares` functions. The `unstake` function is then used to withdraw the assets after the cooldown period has ended. The `unstake` function includes the following check:

```
require(block.timestamp >= userCooldown.cooldownEnd, CooldownNotEnded());
```

This correctly prevents withdrawals before the `cooldownEnd timestamp`. The cooldownEnd is calculated as `block.timestamp + cooldownDuration` when `cooldownAssets` or `cooldownShares` is called.

The `setCooldownDuration` function allows the `DEFAULT_ADMIN_ROLE` to change the `cooldownDuration`. If the `cooldownDuration is set to 0` after a user has initiated a cooldown, the user's `cooldownEnd timestamp remains unchanged` (it's still in the future).

The issue is that the `unstake` function does not consider the possibility of a `zero cooldownDuration`. Even if the global cooldownDuration is set to 0, the user who initiated a cooldown before the change is still forced to wait until their original cooldownEnd timestamp. This creates inconsistent behavior:

Users who initiate a cooldown after `cooldownDuration is set to 0` can effectively `withdraw immediately`. Users who initiated a cooldown before `cooldownDuration` is set to 0 are still subject to the original cooldown period. This inconsistency causes opportunity cost to the users who initiated the cooldown before the `cooldownDuration` was set to 0.

Recommendations:

The unstake function should be modified to account for the possibility of a zero cooldownDuration. The simplest and most effective solution is to add an additional check to the require statement:

```
require(block.timestamp >= userCooldown.cooldownEnd || (
  block.timestamp>=userCooldown.cooldownEnd||

), "Cooldown not ended or duration is 0"
```

This modification ensures that:

If cooldownDuration is greater than 0, the original cooldown check (block.timestamp >= userCooldown.cooldownEnd) is enforced. If cooldownDuration is 0, the withdrawal is allowed regardless of the userCooldown.cooldownEnd value.

[L-09] Pausing StakingRewardsDistributor prevents reward distribution

The `StakingRewardsDistributor` contract has a `pause()` function, callable by the `PAUSER_ROLE`, which halts the functionality of the `transferInRewards` function. The `transferInRewards` function in `StakingRewardsDistributor` is responsible for transferring rewards (USDf) to the `StakedUSDf` contract. The `StakedUSDf` contract's `transferInRewards` function, in turn, updates the internal accounting to distribute those rewards to stakers.

```
function transferInRewards
      (uint256 _rewardsAmount) external whenNotPaused onlyRole(OPERATOR_ROLE) {
        USDF_TOKEN.approve(address(STAKING_VAULT), _rewardsAmount);
        STAKING_VAULT.transferInRewards(_rewardsAmount);
    }
```

If the `StakingRewardsDistributor` is paused, the `transferInRewards` function cannot be called, and therefore no new rewards can be sent to the `StakedUSDf` contract. However, users can *still* deposit USDf into the `StakedUSDf` contract even when the `StakingRewardsDistributor` is paused.

This creates a situation where users deposit their USDf, expecting to earn staking rewards, but receive *no* rewards because the distribution mechanism is paused. Their funds are effectively locked in the `StakedUSDf` contract (subject to the cooldown period if enabled) without accruing any of the intended benefits. This is unfair to users and represents a loss of opportunity cost. The users are worse off than if they had simply held their USDf, or used it in other DeFi protocols.

This affects the users whom have already deposited `sUSDf` into the contract as well. Their funds will be locked till the `cooldownDuration` is over even though they do not accrue any rewards.

The core issue is the lack of synchronization between the deposit functionality of `StakedUSDf` and the reward distribution mechanism of `StakingRewardsDistributor`. Deposits are always allowed, but reward distribution can be unilaterally paused.

A similar issue is found in the `FalconPosition::withdraw` function since if the `duration is not supported` after user have minted into that `duration` he is unable to withdraw till the `matureTimestamp is reached` even though the `depositRewards` can not be called during that period to accrue rewards on the stakers. Hence this creates opportunity cost for the `FalconPositoin minters` since they are locking their funds in the contract without any rewards in return for that specific duration.

Recommendations:

There are several possible approaches to mitigate this issue:

1. **Pause Deposits in `StakedUSDf`:** The most direct solution is to also pause deposits in the `StakedUSDf` contract when the `StakingRewardsDistributor` is paused.

2. **Inform Users:** If pausing deposits is not desired, a clear warning should be displayed in the user interface (and documentation) whenever the `StakingRewardsDistributor` is paused, informing users that they will not receive rewards during this period.

[L-10] Restricted users can bypass asset confiscation via transfers

The `StakedUSDf` contract implements a restriction mechanism (`isRestricted` mapping) intended to prevent certain accounts from interacting with the contract in specific ways. The `redistributeLockedAmount` function is designed to confiscate (either by burning or transferring to the treasury) the sUSDf balance of a restricted account. This function is intended to be called by the `DEFAULT_ADMIN_ROLE`.

```
function redistributeLockedAmount
      (address from, bool burnShares) external onlyRole(DEFAULT_ADMIN_ROLE) {
        // Redistribute only when `from` is restricted
        require(isRestricted[from], AddressNotRestricted(from));

        uint256 amountToDistribute = balanceOf(from);
        _checkZeroAmount(amountToDistribute);

        if (burnShares) {
            uint256 usdfToVest = previewRedeem(amountToDistribute);
            _burn(from, amountToDistribute);
            _updateVestingAmount(usdfToVest);
        } else {
            _transfer(from, TREASURY, amountToDistribute);
        }

        emit LockedAmountRedistributed(from, burnShares, amountToDistribute);
    }
```

However, the `StakedUSDf` contract *does not* prevent restricted users from transferring their sUSDf tokens to other accounts. The `_transfer` function in `ERC20Upgradeable` (which `StakedUSDf` inherits) only checks if the `from` and `to` addresses are the zero address and if the sender has sufficient balance. It does *not* check the `isRestricted` status.

This means a restricted user can simply transfer their sUSDf tokens to a non-restricted account *before* the `redistributeLockedAmount` function is called by the admin. This effectively bypasses the restriction and allows the user to retain control of their assets, defeating the purpose of the `redistributeLockedAmount` mechanism.

The `USDf` contract *does* have a similar restriction mechanism, and *does* prevent transfers (via the `USDf::_update` function) if either the sender or receiver is restricted. This inconsistency between `USDf` and `StakedUSDf` is a major source of the problem.

The most effective solution is to prevent restricted users from transferring their sUSDf tokens. This can be achieved by overriding the `transfer` and `transferFrom` functions in `StakedUSDf` to include a check for the `isRestricted` status of both the `from` and `to` addresses, similar to how it's done in `USDf.sol`.