Zellic

Prepared for

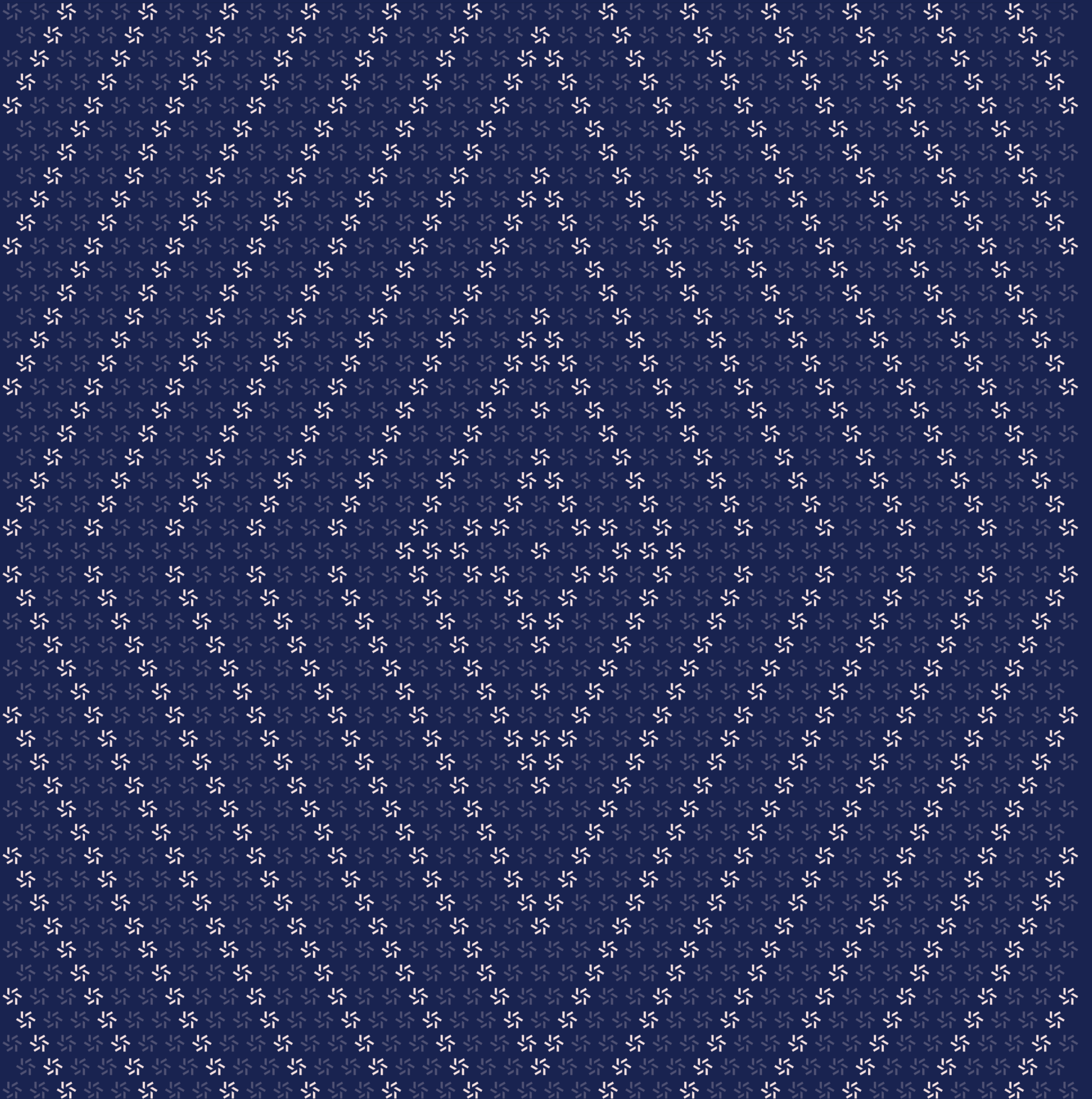
Falcon

Prepared by
Katerina Belotskaia
Juchang Lee
Zellic

March 7, 2025

Falcon Finance

Smart Contract Security Assessment



Contents

About Zellic **4**

1. Overview **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. Introduction **6**

2.1. About Falcon Finance 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. Incorrect `maturityFeeGrowthX128` may lead to miscalculated rewards 11

3.2. Function `_cleanupMaturedBuckets` may run out of gas 13

3.3. The `rescueTokens` does not validate that the token does not match USDf token 15

3.4. Function `updateDurationSettings` allows resetting `totalLiquidity` and `feeGrowthX128` 17

3.5. Account restrictions can be bypassed in StakedUSDf contract 19

3.6. The `setVestingPeriod` may relock previously released assets 21

4. Discussion **22**

4.1. The initialization of StakedUSDf may fail 23

4.2. The `vestingPeriod` and `cooldownDuration` can be set to zero simultaneously 24

5. Threat Model **24**

5.1. Module: ClassicMinterV1.sol 25

5.2. Module: FalconBundler.sol 27

5.3. Module: FalconPosition.sol 28

5.4. Module: PreCollateralizedMinter.sol 31

5.5. Module: StakedUSDf.sol 33

5.6. Module: USDfSilo.sol 37

6. Assessment Results **38**

6.1. Disclaimer 39

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Falcon from February 11th to February 17th, 2025. During this engagement, Zellic reviewed Falcon Finance's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are there any vulnerabilities that could result in unexpected minting or burning of USDf?
- Are there any scenarios where a user could inadvertently lock their funds in staking positions that are excessively long or otherwise unrecoverable?
- Are there any bugs where users may receive greater yield than intended against their proportional share of the staked value?
- Are there any bugs that would prevent users from withdrawing their assets from the staking contracts once matured?
- Are there any bugs or implementation errors that deviate from the expected standards that could result in integration issues and potentially loss of funds with external protocols?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

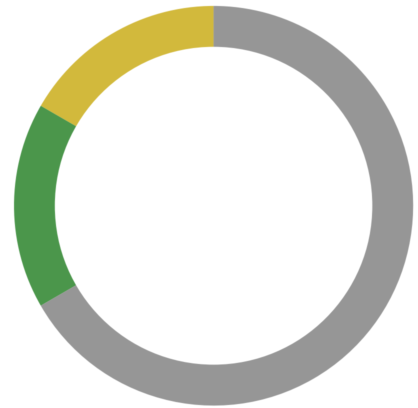
During our assessment on the scoped Falcon Finance contracts, we discovered six findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the

remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Falcon in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	1
Informational	4



2. Introduction

2.1. About Falcon Finance

Falcon contributed the following description of Falcon Finance:

> Falcon Finance is a next-generation synthetic dollar protocol. Preserving users' multi-assets with industry-competitive yields across any market conditions, it sets a new standard in the industry, along with transparency, security, and institutional-grade risk management.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect

its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Falcon Finance Contracts

Type	Solidity
Platform	EVM-compatible

Target	falcon-contracts-evm
Repository	https://github.com/falconfinance/falcon-contracts-evm ↗
Version	`d082d36019a0fda922f65740528a1d5a249cec20`
Programs	`src/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.2 person-weeks. The assessment was conducted by two consultants over the course of five calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Juchang Lee
Engineer
lee@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 11, 2025	Start of primary review period
February 11, 2025	Kick-off call
February 17, 2025	End of primary review period

3. Detailed Findings

3.1. Incorrect `maturityFeeGrowthX128` may lead to miscalculated rewards

Target	FalconPosition		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `maturityFeeGrowthX128` mapping is used to store the current fee growth at maturity time for a specific `duration` supported by the contract.

```
function mature(uint256 duration, uint256 timestamp) public {
    // Round timestamp to days to match mint behavior
    timestamp = (timestamp / 1 days) * 1 days;
    require(timestamp <= block.timestamp, ImmaturePosition());

    // Iterate through supported durations
    MaturityBucket storage bucket = maturityBuckets[duration][timestamp];

    if (bucket.totalLiquidity > 0) {
        // Snapshot current fee growth for this duration
        maturityFeeGrowthX128[timestamp]
    = _durationInfo[duration].feeGrowthX128;
    [...]
    }
}
```

Additionally, `maturityFeeGrowthX128` is utilized for reward calculations for positions locked for a certain `duration` and associated with a specific `maturityTime`.

```
function _unrealizedRewards(Position memory position)
    internal view returns (uint256, uint256) {
    uint256 currentFeeGrowth;

    // If position is matured, use the snapshotted fee growth
    if (block.timestamp >= position.maturityTime) {
        currentFeeGrowth = maturityFeeGrowthX128[position.maturityTime];
    [...]
    uint256 feeGrowthDeltaX128 = currentFeeGrowth
    - position.feeGrowthInsideLastX128;
    return ((position.principal * feeGrowthDeltaX128) >> 128,
```

```
        currentFeeGrowth);
}
```

The issue arises when the contract supports multiple `durations`, as over time, positions with different durations can share the same maturity time.

Impact

The `maturityFeeGrowthX128` value for a specific `timestamp` can be overwritten by the `feeGrowthX128` value corresponding to a different `duration`.

As a result, in the `_unrealizedRewards` function, an incorrect `maturityFeeGrowthX128` snapshot may be used, containing fee-growth data for a `duration` unrelated to the given position. Consequently, this function may return an incorrect reward amount, which could be either higher or lower than expected.

Recommendations

We recommend modifying the `maturityFeeGrowthX128` mapping to store values not only by timestamp but also by duration. This change would ensure that fee-growth data is accurately maintained for each duration separately, preventing unintended overwrites and ensuring correct reward calculations.

Remediation

This issue has been acknowledged by Falcon, and a fix was implemented in commit 9c34a242 ↗.

3.2. Function `_cleanupMaturedBuckets` may run out of gas

Target	FalconPosition		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `_cleanupMaturedBuckets` function may run out of gas if too many days have elapsed since the last `lastMatured` update.

```
function _cleanupMaturedBuckets(uint256 duration) internal {
    uint256 lastMatured = lastMaturedDate[duration];
    uint256 currentDate = (block.timestamp / 1 days) * 1 days;

    // [...]

    for (uint256 date = lastMatured; date <= currentDate; date += 1 days) {
        MaturityBucket storage bucket = maturityBuckets[duration][date];
        if (bucket.totalLiquidity > 0) {
            mature(duration, date);
        }
    }

    // [...]
}
```

Impact

If the last `lastMatured` update was too long ago, the function using `_cleanupMaturedBuckets` (i.e., `depositReward`) may revert due to an out-of-gas error.

Recommendations

To prevent excessive looping, impose a maximum loop limit to restrict the number of loops.

Remediation

This issue has been acknowledged by Falcon.

Falcon provided the following response:

> In the real world we will be calling depositRewards on a daily basis.

3.3. The `rescueTokens` does not validate that the token does not match USDf token

Target	StakingRewardsDistributor		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The StakingRewardsDistributor contract is designed to hold USDf tokens until they are transferred as rewards to the STAKING_VAULT contract. However, this contract also supports the `rescueTokens` function, which allows withdrawing mistakenly transferred tokens or native tokens.

The issue is that there is no validation in `rescueTokens` to ensure that the withdrawn token is not USDf. As a result, reward tokens can also be withdrawn using this function.

```
function rescueTokens(
    address _token,
    address _to,
    uint256 _amount
)
    external
    nonReentrant
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    // [...]

    if (_token == _ETH_ADDRESS) {
        // [...]
    } else {
        IERC20(_token).safeTransfer(_to, _amount);
    }

    // [...]
}
```

Impact

The reward USDf tokens can be unintentionally withdrawn from the contract, potentially affecting the reward-distribution mechanism.

Recommendations

Add a validation check in the `rescueTokens` function to ensure that `_token` does not match the address of USDf token, preventing reward-token withdrawals.

Remediation

This issue has been acknowledged by Falcon.

Falcon provided the following response:

> We believe this is an intentional design choice. The ability for the admin to withdraw any tokens, including USDf, provides necessary flexibility for emergency situations or when funds need to be reallocated. This admin privilege is part of our trust model and will be clearly documented for users. Since the function is protected by the DEFAULT_ADMIN_ROLE, only trusted administrators can execute it, mitigating the risk of misuse.

3.4. Function `updateDurationSettings` allows resetting `totalLiquidity` and `feeGrowthX128`

Target	FalconPosition		
Category	Business Logic	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The FalconPosition contract supports different staking durations, and using the `updateDurationSettings` function, a caller with the `DEFAULT_ADMIN_ROLE` can enable a new duration or disable an already supported one.

If the provided `duration` is not yet supported, the `_durationInfo` mapping will be updated with a new `DurationInfo` object, initializing `totalLiquidity` and `feeGrowthX128` to zero. The `totalLiquidity` variable tracks the current immature liquidity for the specified `duration`, while `feeGrowthX128` serves as an accumulator for fees per `duration`.

```
function updateDurationSettings(
    uint256 duration,
    bool isSupported,
    bool mintEnabled
)
    external
    onlyRole(DEFAULT_ADMIN_ROLE)
{
    // If duration wasn't previously supported, require mintEnabled to be false
    if (!_durationInfo[duration].isSupported && isSupported) {
        require(duration > 0, InvalidDuration());
        _durationInfo[duration] =
            DurationInfo({isSupported: true, mintEnabled: mintEnabled,
    totalLiquidity: 0, feeGrowthX128: 0});
    } else {
        _durationInfo[duration].isSupported = isSupported;
        _durationInfo[duration].mintEnabled = mintEnabled;
    }

    emit DurationUpdated(duration, isSupported, mintEnabled);
}
```

However, if the `updateDurationSettings` function is used to temporarily disable a specified

`duration` and enable it again, the existing `totalLiquidity` and `feeGrowthX128` values will be reset to zero.

Impact

Resetting `totalLiquidity` and `feeGrowthX128` to zero will lock withdrawal and reward-collection functionalities for all currently immature positions associated with the specified `duration`, making these actions impossible to perform. However, since this function is controlled by a `DEFAULT_ADMIN_ROLE` and is not intended to be used for disabling previously activated durations, the impact of this issue is classified as Informational.

Recommendations

We recommend adding a verification step to check whether the `duration` has been previously supported and ensuring that existing `totalLiquidity` and `feeGrowthX128` values are not reset when temporarily disabling a duration.

Remediation

This issue has been acknowledged by Falcon, and a fix was implemented in commit 82d42cb8 ↗.

3.5. Account restrictions can be bypassed in StakedUSDf contract

Target	StakedUSDf		
Category	Business Logic	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The StakedUSDf contract is an ERC-4626 vault implementation that supports account restrictions for the owners of sUSDf share tokens, preventing these accounts from depositing or withdrawing assets. However, since sUSDf tokens are transferable and the contract does not verify whether the initiator of a token transfer is currently restricted, this allows for effectively bypassing these restrictions.

```
function _deposit(address caller, address receiver, uint256 assets,
    uint256 shares) internal override {
    [...]
    _checkRestricted(caller);
    _checkRestricted(receiver);
    [...]
}

function _withdraw(address caller, address receiver, address owner,
    uint256 assets, uint256 shares)
    internal
    override
{
    [...]
    _checkRestricted(caller);
    _checkRestricted(receiver);
    _checkRestricted(owner);
    [...]
}
```

Impact

Restricted accounts may still transfer sUSDf tokens, which could allow them to bypass withdrawal restrictions. However, since there is no intention or plan to restrict transfers of sUSDf tokens for compliance or operational purposes, the impact of this issue is classified as Informational.

Recommendations

We recommend implementing a validation check within the transfer logic to ensure that restricted accounts cannot initiate token transfers.

Remediation

This issue has been acknowledged by Falcon, and a fix was implemented in commit `88d224fa` ↗.

3.6. The setVestingPeriod may relock previously released assets

Target	StakedUSDf		
Category	Business Logic	Severity	Informational
Likelihood	N/A	Impact	Informational

Description

The `setVestingPeriod` function in the StakedUSDf contract allows changing the `vestingPeriod` duration, but it does not verify whether the current unvested amount is zero.

As a result, if the vesting period is increased, a portion of previously released assets may become locked again.

```
function _setVestingPeriod(uint32 newPeriod) internal {
    uint32 oldVestingPeriod = vestingPeriod;
    require(newPeriod <= MAX_VESTING_PERIOD, DurationExceedsMax());
    require(oldVestingPeriod != newPeriod, DurationNotChanged());
    require(newPeriod > 0 || cooldownDuration > 0, ExpectedCooldownOn()); // if
    period is 0, cooldown must be on

    vestingPeriod = newPeriod;
    emit VestingPeriodUpdated(oldVestingPeriod, newPeriod);
}

function getUnvestedAmount() public view returns (uint256) {
    uint256 timeSinceLastDistribution = uint40(block.timestamp)
    - lastDistributionTimestamp;
    if (timeSinceLastDistribution >= vestingPeriod) {
        return 0;
    }
    uint256 deltaT;
    unchecked {
        deltaT = (vestingPeriod - timeSinceLastDistribution);
    }
    return (deltaT * vestingAmount) / vestingPeriod;
}
```

Impact

If assets were already withdrawn before the vesting period increase, the `totalAssets` calculation will be incorrect, leading to potential discrepancies in the system's accounting. However, since this function is controlled by a `DEFAULT_ADMIN_ROLE` and is intended to be used only for reducing the vesting duration, the impact of this issue is classified as Informational.

Recommendations

Implement a verification check in the `setVestingPeriod` function to ensure that the current unvested amount is zero before allowing changes to the vesting period.

Remediation

This issue has been acknowledged by Falcon, and a fix was implemented in commit `4a5ce0b9` ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. The initialization of StakedUSDf may fail

The USDfSilo contract's `initialize` function calls to set the `_stakingVault` address using `silo_.setStakingVault()`.

However, since `setStakingVault` can be invoked by any caller before the initialization, this may lead to a failed contract initialization.

```
contract StakedUSDf is IStakedUSDf, AccessControlUpgradeable,
    ERC20PermitUpgradeable, ERC4626Upgradeable {
    // [...]
    function initialize(
        IERC20 usdf,
        address admin,
        USDfSilo silo_,
        uint32 initialVesting,
        uint24 initialCooldown
    )
        external
        initializer
    {
        // [...]

        silo = silo_;
        silo_.setStakingVault();
    }
    // [...]
}
```

```
contract USDfSilo {
    // [...]
    constructor(address usdf) {
        _USDF = IERC20(usdf);
    }
    // [...]
    function setStakingVault() external {
        require(_stakingVault == address(0), AlreadySet());
        _stakingVault = msg.sender;
    }
```

```
    // [...]
}
```

Falcon provided the following response:

> we already deployed the contracts successfully and the issue is not relevant anymore.

4.2. The `vestingPeriod` and `cooldownDuration` can be set to zero simultaneously

In the StakedUSDf contract, the `_setVestingPeriod` function includes a validation check to ensure that either the new `vestingPeriod` or the current `cooldownDuration` is nonzero. However, the `setCooldownDuration` function lacks a similar verification.

As a result, it is possible for both `vestingPeriod` and `cooldownDuration` to be set to zero simultaneously. However, it is expected that if `vestingPeriod` is set to zero, the cooldown mechanism should remain active. We recommend adding the relevant validation check to `setCooldownDuration`.

Falcon provided the following response:

> This configuration is controlled by trusted administrators, and we have appropriate governance processes in place to ensure that any changes to these parameters are thoroughly considered.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: ClassicMinterV1.sol

Function: `callerFundedMint(MintParams params, uint256 depositAmount, bytes signature)`

This function mints an amount of USDf tokens for the recipient, calculated based on the token price.

Inputs

- `params`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Struct of `MintParams` that has information for minting.
- `depositAmount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Amount of tokens to deposit.
- `signature`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The address recovered using the generated hash and corresponding signature must possess the `MINTER_ROLE`.
    - **Impact**: Bytes of signature.

Branches and code coverage

Intended branches

- Check if the caller is equal to `params.caller`.
    - ☑ Test coverage
- Check if `depositAmount` is not zero.
    - ☑ Test coverage
- Check if `depositAmount` is smaller than `params.maxAmount`.

- ☑ Test coverage
- Check if `params.recipient` is not the zero address.
    - ☑ Test coverage
- Check if `params.expiry` is bigger than `block.timestamp` to confirm the transaction is not expired.
    - ☑ Test coverage
- Check that the nonce has not been used before.
    - ☑ Test coverage
- Check if the recovered address using the signature has `BACKEND_SIGNER_ROLE`.
    - ☑ Test coverage
- Transfer collateral to treasury.
    - ☑ Test coverage
- Calculate mint amount with precise scaling and mint an amount of USDf.
    - ☑ Test coverage

Negative behavior

- If the caller is not equal to `params.caller`, the transaction will be reverted.
    - ☐ Negative test
- If `depositAmount` is zero, the transaction will be reverted.
    - ☐ Negative test
- If `depositAmount` is bigger than or equal to `params.maxAmount`, the transaction will be reverted.
    - ☐ Negative test
- If `params.recipient` is the zero address, the transaction will be reverted.
    - ☑ Negative test
- If `params.expiry` is smaller than or equal to `block.timestamp`, the transaction will be reverted.
    - ☐ Negative test
- If the nonce has been used before, the transaction will be reverted.
    - ☐ Negative test
- If the recovered address using the signature does not have `BACKEND_SIGNER_ROLE`, the transaction will be reverted.
    - ☑ Negative test

5.2. Module: FalconBundler.sol

Function: `stakeToFalconPosition(uint256 amount, uint256 duration, address recipient)`

This function stakes existing USDf directly to the FalconPosition contract.

Inputs

- `amount`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Amount of USDf to stake.
- `duration`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Duration of staking for the FalconPosition contract.
- `recipient`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Address for receiving staking position NFT.

Branches and code coverage

Intended branches

- Check if the amount is not zero.
    - ☑ Test coverage
- Check if the recipient is not the zero address.
    - ☑ Test coverage
- Transfer USDf from the caller.
    - ☑ Test coverage
- Stake USDf and receive sUSDf.
    - ☑ Test coverage
- Call FalconPosition's mint function to mint position NFT.
    - ☑ Test coverage
- Transfer position NFT to the recipient.

- [x] Test coverage

Negative behavior

- If the amount is zero, the transaction will be reverted.
  - [ ] Negative test
- If the recipient is the zero address, the transaction will be reverted.
  - [ ] Negative test

5.3. Module: FalconPosition.sol

Function: `collect(uint256 tokenId)`

This function collects accrued fees for the position.

Inputs

- `tokenId`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: ID of position NFT.

Branches and code coverage

Intended branches

- Check if yield collection is enabled.
  - [x] Test coverage
- Update position with accrued fees.
  - [x] Test coverage
- Transfer `tokensOwed` to the caller.
  - [x] Test coverage

Negative behavior

- If yield collection is not enabled, the transaction will be reverted.
  - [ ] Negative test

Function: `mature(uint256 duration, uint256 timestamp)`

This function matures positions for a specific timestamp.

Inputs

- `duration`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Duration of staking in seconds.
- `timestamp`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Maturity timestamp to process.

Branches and code coverage

Intended branches

- Round down the timestamp and check if the timestamp is smaller than or equal to `block.timestamp`.
  - ☑ Test coverage
- Snapshot current fee growth for the duration.
  - ☑ Test coverage
- Mature the position if it hasn't been matured yet.
  - ☑ Test coverage

Negative behavior

- If rounded-down timestamp is bigger than `block.timestamp`, the transaction will be reverted.
  - ☑ Negative test

Function: `mint(uint256 principal, uint256 duration)`

This function creates a new staking position.

Inputs

- `principal`

    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Amount of sUSDf to stake.
- `duration`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Duration of staking in seconds.

Branches and code coverage

Intended branches

- Check if the principal is bigger than zero.
    - ☑ Test coverage
- Check if the duration is supported.
    - ☑ Test coverage
- Check if the duration is enabled for mint.
    - ☑ Test coverage
- Update duration liquidity for tracking.
    - ☑ Test coverage
- Create position and transfer principal from the caller to the contract.
    - ☑ Test coverage
- Mint NFT to the caller.
    - ☑ Test coverage

Negative behavior

- If the principal is smaller than or equal to zero, the transaction will be reverted.
    - ☐ Negative test
- If the duration is not supported, the transaction will be reverted.
    - ☑ Negative test
- If the duration is not enabled, the transaction will be reverted.
    - ☑ Negative test

Function: `withdraw(uint256 tokenId)`

This function withdraws principal after maturity.

Inputs

- `tokenId`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: ID of position NFT.

Branches and code coverage

Intended branches

- Check if the owner of position NFT is the caller.
    - ☑ Test coverage
- Check if the position duration has matured.
    - ☑ Test coverage
- If the position has not matured, call `mature` to mature the position.
    - ☑ Test coverage
- Collect any remaining fees and calculate the total amount of principal plus `tokenOwed`.
    - ☑ Test coverage
- Burn NFT and transfer the `totalAmount` to the caller.
    - ☑ Test coverage

Negative behavior

- If the owner of position NFT is not the caller, the transaction will be reverted.
    - ☑ Negative test
- If the position duration is immature, the transaction will be reverted.
    - ☑ Negative test

5.4. Module: PreCollateralizedMinter.sol

Function: `preCollateralizedMint(MintParams params, bytes signature)`

This function mints an amount of USDf tokens to the recipient with a signature.

Inputs

- `params`
    - **Control**: Fully controlled by the caller.

    - **Constraints**: N/A.
    - **Impact**: Struct of `MintParams` that has information for minting.
- `signature`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: The address recovered using the generated hash and corresponding signature must possess the `MINTER_ROLE`.
    - **Impact**: Bytes of signature.

Branches and code coverage

Intended branches

- Check if `params.collateralRef` is not zero.
    - ☑ Test coverage
- Check if `params.amount` is bigger than zero.
    - ☑ Test coverage
- Check if `params.recipient` is not the zero address.
    - ☑ Test coverage
- Check if `params.expiry` is bigger than `block.timestamp`.
    - ☑ Test coverage
- Check if `params.nonce` is not used.
    - ☑ Test coverage
- Build signature with `MintParams` for verification.
    - ☑ Test coverage
- Check if the recovered address using the signature has `MINTER_ROLE`.
    - ☑ Test coverage
- Mint an amount of `params.amount` USDf for `params.recipient`.
    - ☑ Test coverage

Negative behavior

- If `params.collateralRef` is zero, the transaction will be reverted.
    - ☑ Negative test
- If `params.amount` is not bigger than zero, the transaction will be reverted.
    - ☑ Negative test
- If `params.recipient` is the zero address, the transaction will be reverted.
    - ☑ Negative test

- If `params.expiry` is not bigger than `block.timestamp`, the transaction will be reverted.
    - ☑ Negative test
- If `params.nonce` is used, the transaction will be reverted.
    - ☑ Negative test
- If the recovered address using the signature does not have `MINTER_ROLE`, the transaction will be reverted.
    - ☑ Negative test

5.5. Module: StakedUSDf.sol

Function: `cooldownAssets(uint256 assets, address owner)`

This function withdraws assets with a cooldown period.

Inputs

- `assets`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Amount of assets to withdraw.
- `owner`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Address of the owner of assets.

Branches and code coverage

Intended branches

- Set owner's `cooldownEnd` to current `block.timestamp` plus `cooldownDuration`.
    - ☑ Test coverage
- Add owner's `underlyingAmount` amount of assets.
    - ☑ Test coverage
- Call parent's withdrawal for silo, which receives withdrawals.
    - ☑ Test coverage

Function: `cooldownShares(uint256 shares, address owner)`

This function redeems shares with a cooldown period.

Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount of shares to redeem.
- `owner`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address of the owner of shares.

Branches and code coverage

Intended branches

- Call parent's redemption for silo, which receives redemptions.
  - ☑ Test coverage
- Set the owner's `cooldownEnd` to the current `block.timestamp` and `cooldownDuration`.
  - ☑ Test coverage
- Add the owner's `underlyingAmount` amount of assets that return redemptions.
  - ☑ Test coverage

Function: `redeem(uint256 shares, address receiver, address owner)`

This function overrides the redeem function of ERC-4626 and can be called when the cooldown is off.

Inputs

- `shares`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount of shares to be redeemed.
- `receiver`

    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Address of the receiver who will receive the redeemed assets.
- `owner`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Owner's address of the vault shares to be burned for the withdrawal.

Branches and code coverage

Intended branches

- Check if the cooldown is off.
    - [x] Test coverage
- Call parent's redeem function.
    - [x] Test coverage

Negative behavior

- If the cooldown is not off, the transaction will be reverted.
    - [x] Negative test

Function: `unstake(address receiver)`

This function unstakes assets after the cooldown period ends.

Inputs

- `receiver`
    - **Control**: Fully controlled by the caller.
    - **Constraints**: N/A.
    - **Impact**: Address to receive the assets.

Branches and code coverage

Intended branches

- Check that the receiver is not the zero address.
    - [x] Test coverage

- Check that the caller's cooldown period has ended.
  - ☑ Test coverage
- Set caller's cooldown and `underlyingAmount` to zero.
  - ☑ Test coverage
- Call silo's withdraw function to withdraw assets with the amount of `underlyingAmount`.
  - ☑ Test coverage

Negative behavior

- If the receiver is the zero address, the transaction will be reverted.
  - ☐ Negative test
- If the caller's cooldown period has not ended, the transaction will be reverted.
  - ☑ Negative test

Function: `withdraw(uint256 assets, address receiver, address owner)`

This function overrides the withdraw function of ERC-4626 and can be called when the cooldown is off.

Inputs

- `assets`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Amount of assets to be withdrawn.
- `receiver`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Address of the receiver who will receive the withdrawn shares.
- `owner`
  - **Control**: Fully controlled by the caller.
  - **Constraints**: N/A.
  - **Impact**: Owner's address of the vault shares to be burned for the withdrawal.

Branches and code coverage

Intended branches

- Check if cooldown is off.
    - ☑ Test coverage
- Call parent's withdraw function.
    - ☑ Test coverage

Negative behavior

- If cooldown is not off, the transaction will be reverted.
    - ☑ Negative test

5.6. Module: USDfSilo.sol

Function: `setStakingVault()`

This function sets the caller of this function as the `_stakingVault` address if it has not been set before. The `_stakingVault` address can call the `withdraw` function to withdraw `_USDF` tokens from this contract.

Function: `withdraw(address to, uint256 amount)`

This function allows `_stakingVault` to withdraw `_USDF` tokens from this contract.

Inputs

- `to`
    - **Control**: Full control.
    - **Constraints**: N/A.
    - **Impact**: The address of the receiver of `_USDF` tokens.
- `amount`
    - **Control**: Full control.
    - **Constraints**: N/A.
    - **Impact**: Amount of `_USDF` tokens to be withdrawn.

Branches and code coverage

Negative behavior

- Caller is not a `_stakingVault`.
    - ☐ Negative test

Function call analysis

- `this._USDF.transfer(to, amount)`
    - **What is controllable?** `to` and `amount`.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the `_USDF` balance of this contract is less than the provided `amounts`.

6. Assessment Results

At the time of our assessment, the reviewed code was deployed to the Ethereum Mainnet.

During our assessment on the scoped Falcon Finance contracts, we discovered six findings. No critical issues were found. One finding was of medium impact, one was of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.