Zellic

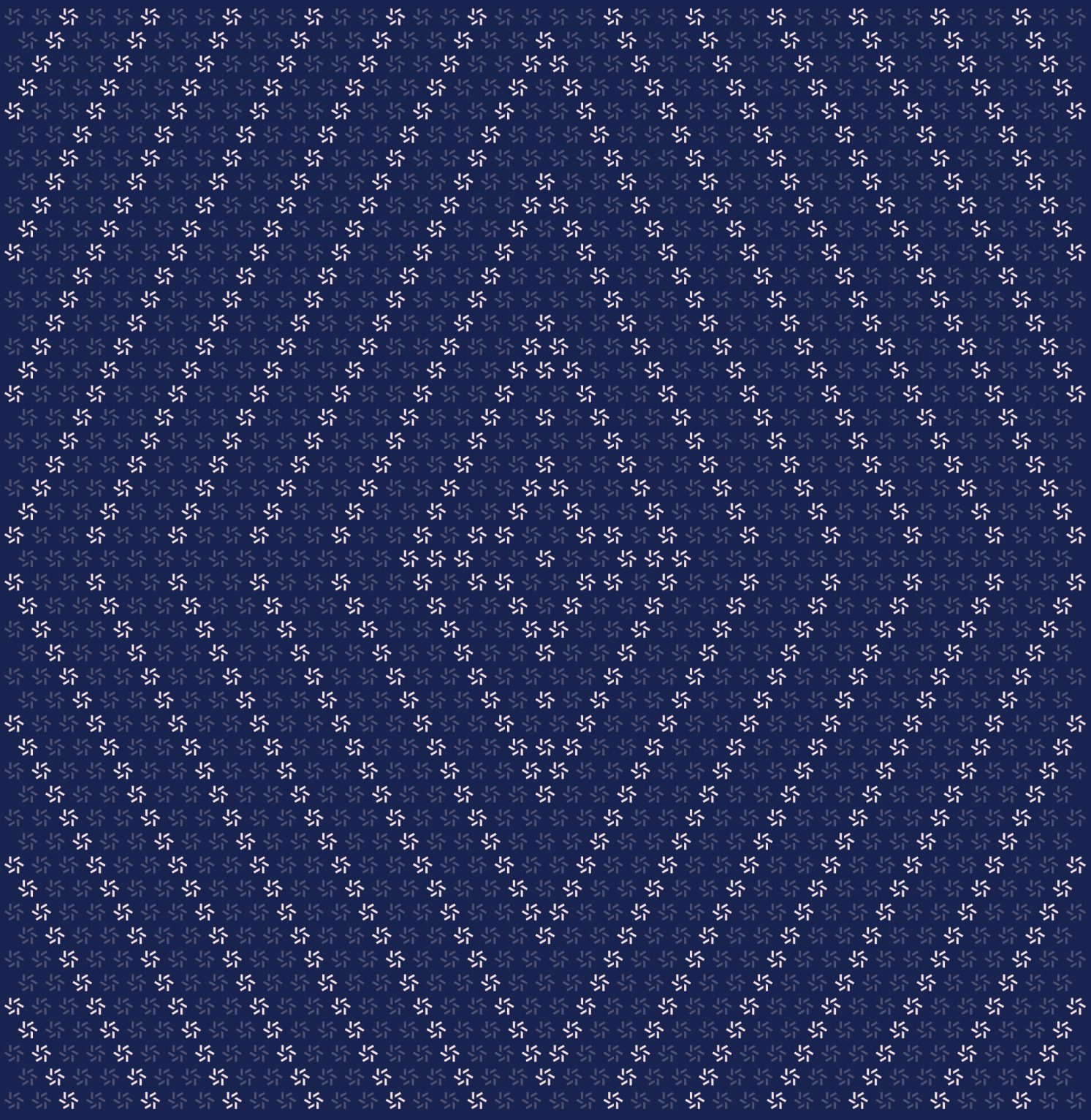
Prepared for
Solera Markets

Prepared by
Aaron Esau
Jinheon Lee
Zellic

February 4, 2025

Solera

Smart Contract Security Assessment



Contents

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 Solera Markets from February 3rd to February 4th, 2025. During this engagement, Zellic reviewed Solera'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:

- Could an on-chain attacker drain the vault?
- Could the vesting admin cause denial of service (DOS) and lock up user funds?
- Is it possible for an attacker to siphon riskless yield (profit) from the vault?
- Are there any exploitable rounding errors?
- Are there read-only reentrancy opportunities, despite there being `nonReentrant` flags on state-modifying functions?

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 Solera contracts, we discovered seven findings. No critical issues were found. Three findings were of high impact, one was of medium impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	3
Medium	1
Low	0
Informational	3

2. Introduction

2.1. About Solera

Solera Markets contributed the following description of Solera:

> Solera is the first fully integrated DeFi suite with a custom framework for collateralizing and borrowing against RWAs and yield bearing assets on Plume network.

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:

Solera Contracts

Type	Solidity
Platform	EVM-compatible
Target	solera-staking
Repository	https://github.com/SoleraMarkets/solera-staking ↗
Version	`280badbfcc472cd486982094e528008da888d678`
Programs	`src/SoleraStaking.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 1.5 person-days. The assessment was conducted by two consultants over the course of two 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:

Aaron Esau
Engineer
aaron@zellic.io

Jinheon Lee
Engineer
jinheon@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 3, 2025	Kick-off call
February 3, 2025	Start of primary review period
February 4, 2025	End of primary review period

3. Detailed Findings

3.1. Vesting admin can manipulate price ratio

Target	SoleraStaking		
Category	Protocol Risks	**Severity**	Critical
Likelihood	Medium	**Impact**	High

Description

The price ratio is the ratio of assets to shares. The following function determines how many assets are in the vault:

```
function totalAssets() public view override returns (uint256) {
    return super.totalAssets() - getVestingDeposit()
    + vestedAmount(block.timestamp.toUint64());
}
```

Since `super.totalAssets()` includes the vesting deposit, it first subtracts that and then adds the amount of vesting deposit that has vested up until the current timestamp. The vesting admin has the ability to deposit and withdraw funds at any time, but it does not have the ability to change the vesting period.

If the vesting admin deposits or withdraws during the vesting period, the price ratio will immediately increase or decrease, respectively. This is a powerful ability of the vesting admin.

Impact

The aforementioned fact — that the vesting admin can change the price ratio during the vesting period — is problematic even if the vesting admin is not malicious/compromised.

> For the remainder of this section, we will use the following variables:
> - Let n be the number of shares in the vault.
> - Let x be the total number of assets in the vault, including vesting deposits.

We see at least two impacts of the fact.

1. Vesting admin can drain the vault

After (or during) a vesting period, the vesting admin can completely drain the vault by manipulating the price ratio.

The following steps demonstrate this attack:

1. Atomically, in one transaction,
    - The vesting admin deposits the same number of assets as is in the vault as a normal deposit so that they own half the shares. The new price ratio is $\frac{2x}{2n}$.
    - Funding the attack using a flash loan, the vesting admin deposits the same number of assets as is now in the vault ($x + x$) as a vesting deposit, thereby doubling the price ratio. The new price ratio is $\frac{4x}{2n}$.
    - The vesting admin initiates redemption for n shares. It locks in the `WithdrawRequest` that $2x$ assets will be given to the vesting admin.
    - The vesting admin withdraws the vesting deposit of $2x$ and repays the flash loan. The new price ratio is $\frac{2x}{2n}$.
2. After the redemption timelock, the vesting admin withdraws $2x$ assets from their `WithdrawRequest`. The vault is now drained.

2. Users can cause the vault to become insolvent

Whether done intentionally or not, if the vesting admin withdraws during or after the vesting period — provided at least one user has deposited assets before the vesting admin deposits a vesting deposit — it can cause the vault to become insolvent.

The following steps demonstrate an extreme case of this to prove that insolvency is possible. Toward the end of a vesting period, consider if the following transactions executed in order:

1. A user deposits the same number of assets as is in the vault (x) as a normal deposit so that they own n, which is half of the shares. The new price ratio is $\frac{2x}{2n}$.
2. The vesting admin deposits the same number of assets as is now in the vault ($x + x$) as a vesting deposit, thereby doubling the price ratio. The new price ratio is $\frac{4x}{2n}$.
3. Some time passes, and the vesting period is almost over. The user observes step 4 in the mempool and front-runs the transaction to initiate redemption for n shares. It locks in the `WithdrawRequest` that $2x$ assets will be given to the vesting admin.
4. Vesting admin withdraws the vesting deposit of $2x$. The new price ratio is $\frac{2x}{2n}$.
5. After the redemption timelock, the user withdraws $2x$ assets from their `WithdrawRequest`. The vault is now drained.

In practice, the vesting admin probably would not have such a large deposit, but the point is still true that vesting withdrawals during the time period lead to insolvency. Step 3 in particular is likely to happen by accident, as a user may want to redeem their shares early to take advantage of the vested earnings.

Recommendations

It is difficult to prevent this. We recommend that the vesting admin should not be able to withdraw vested assets during or after the vesting period.

Another alternative is to disallow the vesting admin from withdrawing when open redemption requests exist, but this enables DOS attacks since a user can create a redemption request and never call `redeem`.

Finally, a more difficult solution is to entirely separate vesting assets from the vault assets so that vesting deposits and withdrawals do not impact the price ratio.

Remediation

This issue has been acknowledged by Solera Markets, and fixes were implemented in the following PRs:

- PR #14 ↗
- PR #15 ↗

3.2. Vesting admin can take advantage of open approvals

Target	SoleraStaking		
Category	Protocol Risks	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

The vesting admin has the ability to call the following two functions:

```
function vestingDeposit(
    address from,
    uint256 amount
) public nonReentrant onlyRole(VESTING_ADMIN_ROLE) returns (uint256) {
    SafeERC20.safeTransferFrom(IERC20(asset()), from, address(this), amount);
    _vestingDeposit += amount;
    emit VestingDeposit(from, amount);

    return amount;
}

function vestingWithdraw(
    address to,
    uint256 amount
) public nonReentrant onlyRole(VESTING_ADMIN_ROLE) returns (uint256) {
    if (amount > getVestingDeposit()) {
        revert VestingWithdrawAmountExceeded(amount, getVestingDeposit());
    }
    SafeERC20.safeTransfer(IERC20(asset()), to, amount);
    _vestingDeposit -= amount;
    emit VestingDeposit(to, amount);

    return amount;
}
```

By calling `vestingDeposit(victim, amount);` immediately followed by `vestingWithdraw(attacker, amount);`, the vesting admin essentially has the following primitive, where the `victim`, `attacker`, and `amount` are arbitrarily controllable values:

```
asset().transferFrom(victim, attacker, amount);`
```

Impact

Depending on which asset the contract is configured to use, the vesting admin may be able to drain the entire vault by using the aforementioned `transferFrom` primitive, where the `victim` is the vault address. In the latest OpenZeppelin implementation of the ERC20 contract, `transferFrom` would require an approval from the vault address to the vault address (i.e., there is a requirement that `allowance(vault, vault) > 0` — see source here ↗).

However, the SoleraStaking contract is ERC-20 implementation-agnostic. Other implementations may not have this same requirement; for example, WETH and DAI both exclude this check (see source here ↗).

Regardless of the ERC-20 that SoleraStaking is intended to be deployed with, leaving the vesting admin with this ability hinders the reusability of this contract. In the future, other developers who want to build on the contract may not be aware of the requirement that prevents a critical bug from existing.

And in any case, if a user leaves open approvals to the SoleraStaking contract (e.g., if they approve more than necessary to call `deposit`, which relies on allowances), or if the user approves and deposits in separate transactions, then the vesting admin can steal the user's approved assets.

Exploitation in any scenario requires that the vesting admin is malicious or compromised, so we rate this finding as Low likelihood. However, in our opinion, leaving the vesting admin with this primitive entirely defeats the purpose of having a separate role — this finding is therefore part of the contract's threat model.

Recommendations

The vesting admin should not have the ability to withdraw from an arbitrary address. Though it may cost the vesting admin slightly more gas, we recommend having the vesting admin transfer the assets to their wallet first. The `vestingDeposit` function should be changed to only allow deposits from the vesting admin's wallet.

Remediation

This issue has been acknowledged by Solera Markets, and a fix was implemented in `PR #11` ↗.

3.3. Race condition allows admin to drain the vault

Target	SoleraStaking		
Category	Protocol Risks	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

There is an opportunity for a race condition during the redemption step. After calling `requestRedeem` but before `redeem` is executed, an admin changing the withdraw fee can cause more assets than a user paid for to be transferred out of the vault. The root cause is that the fee percentage is calculated at a separate step from the calculation of the number of assets to transfer.

There are two scenarios:

1. **The fee percentage increases.** The user will have transferred fewer shares than necessary in the `requestRedeem` step to cover the new fees. The user would have underpaid in shares, and more of the vault's assets than intended would be used to cover the difference.
2. **The fee percentage decreases.** The user will have transferred more shares than necessary. The user would have underpaid, and the assets would remain in the vault (value distributed to remaining users).

In the scenario that the fee percentage increases, another edge-case situation may be encountered in low-liquidity situations. When shares are burned and assets are calculated in the `requestRedeem` step, the vault may not have enough assets to cover the increased fee in the `redeem` step. This would cause the redeem to abort, and the funds would be stuck in the vault.[1]

Impact

This issue may be encountered in practice accidentally, but in our opinion, the most risky scenario is if a malicious or compromised admin changes the fee percentage intentionally to steal funds.

The following steps describe how the admin could drain the entire vault:

1. They deposit exactly double the number of assets as are in the vault.

[1] Technically, someone could `deposit`, which is essentially donating enough assets to cover the increased fees.

2. In one transaction, they atomically
    - change the fee to 0%,
    - set the fee receiver to the admin's address,
    - request a redemption of all the admin-deposited shares, and
    - change the fee to 100%.[2]
3. After waiting the required time, they redeem the shares. The vault's assets will be used to cover the fees, leaving all remaining shares worthless.

Recommendations

Store the fee percentage in the withdrawal request.

Alternatively, require that there are no open withdrawal requests to change fee percentage. Note that this solution would enable anyone to block fees by repeatedly creating zero-asset withdrawal requests with overlapping timelock periods so that the queue is never fully cleared.

Remediation

This issue has been acknowledged by Solera Markets, and a fix was implemented in PR #8.

[2] Using an amount higher than 100% would result in an integer-underflow reversion in `previewRedeem`.

3.4. Fee is not limited to 100%

Target	SoleraStaking		
Category	Protocol Risks	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

Note that the admin can configure a fee of more than 100%:

```
function setFeeBasisPoints(
    uint256 _feeBasisPoints
) public onlyRole(CONFIG_ADMIN_ROLE) returns (uint256) {
    feeBasisPoints = _feeBasisPoints;
    return feeBasisPoints;
}
```

Among other potential places in the code, the following multiplication in `_feeOnRaw` could revert:

```
function _withdraw(
    address caller,
    address receiver,
    address owner,
    uint256 assets,
    uint256 shares
) internal virtual override {
    uint256 fee = _feeOnRaw(assets, _getFeeBasisPoints()); // in units of
        underlying asset
    address recipient = _getFeeRecipient();
    // [...]
}

// [...]
function _feeOnRaw(uint256 assets, uint256 _feeBasisPoints)
    private pure returns (uint256) {
    return assets.mulDiv(_feeBasisPoints, _BASIS_POINT_SCALE, Math.Rounding.
        Ceil);
}
```

Impact

The admin could block redemptions by preventing `_withdraw` from being able to be executed.

Recommendations

This centralization risk may be of low concern if the admin is a governance contract or multi-sig. However, we recommend limiting the fee to 100% (`feeBasisPoints < _BASIS_POINT_SCALE`) to be logically consistent and easily reduce centralization risk.

To further limit centralization risk, consider further limiting the fee (e.g., to 5%) so that users do not have to trust the admin as much.

Remediation

This issue has been acknowledged by Solera Markets, and a fix was implemented in PR #12 ↗.

3.5. Consider overriding obselete OpenZeppelin ERC4626 functions

Target	SoleraStaking		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The SoleraStaking contract overrides the following functions to disable them, since the intended use flow does not involve them:

```
/* Unused overriden functions */

function withdraw(uint256, address, address)
    public pure override returns (uint256) {
    revert UnusedFunction();
}

function redeem(uint256, address, address)
    public pure override returns (uint256) {
    revert UnusedFunction();
}

function mint(uint256, address) public pure override returns (uint256) {
    revert UnusedFunction();
}
```

However, these functions from OpenZeppelin's ERC4626 contract — which are now also obselete — are not overriden:

```
/** @dev See {IERC4626-maxMint}. */
function maxMint(address) public view virtual returns (uint256) {
    return type(uint256).max;
}

/** @dev See {IERC4626-maxWithdraw}. */
function maxWithdraw(address owner) public view virtual returns (uint256) {
    return _convertToAssets(balanceOf(owner), Math.Rounding.Floor);
}

// [...]
```

```
/** @dev See {IERC4626-previewMint}. */
function previewMint(uint256 shares) public view virtual returns (uint256) {
    return _convertToAssets(shares, Math.Rounding.Ceil);
}
```

Additionally, SoleraStaking unnecessarily contains the following function override, despite the fact that the `withdraw` function is disabled:

```
/// @dev Preview adding an exit fee on withdraw. See
    {IERC4626-previewWithdraw}.
function previewWithdraw(uint256 assets)
    public view virtual override returns (uint256) {
    uint256 fee = _feeOnRaw(assets, _getFeeBasisPoints());
    return super.previewWithdraw(assets + fee);
}
```

Impact

These may lead to reduced code readability and maintainability, and increased deployment cost (unnecessary, extra compiled code in the contract).

Recommendations

Consider explicitly overriding and disabling functions that are no longer relevant to the child contract.

Remediation

This issue has been acknowledged by Solera Markets, and a fix was implemented in PR #9 ↗.

3.6. Integer underflow on `redeemRequest(0)`

Target	SoleraStaking		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

When calling `redeemRequest(0)`, an integer underflow will occur due to fees rounding up:

```
function previewRedeem(uint256 shares)
    public view virtual override returns (uint256) {
    uint256 assets = super.previewRedeem(shares);
    return assets - _feeOnTotal(assets, _getFeeBasisPoints());
}

// [...]
function _feeOnTotal(uint256 assets, uint256 _feeBasisPoints)
    private pure returns (uint256) {
    return assets.mulDiv(_feeBasisPoints, _feeBasisPoints
    + _BASIS_POINT_SCALE, Math.Rounding.Ceil);
}
```

Impact

There is no security impact of this. However, the error may be confusing.

If the underflow did not occur in `previewRedeem`, a bug would exist where users could grief the vault by redeeming no assets, which burns one base unit of asset every time.[3]

Recommendations

Require that `shares` is greater than 0 in `redeemRequest`.

[3] This attack would likely be precluded by gas expenses.

Remediation

This issue has been acknowledged by Solera Markets, and a fix was implemented in PR #10.

3.7. Vesting withdrawals emit the wrong event

Target	Solera		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `vestingWithdraw` function mistakenly emits the `VestingDeposit` event instead of `VestingWithdraw`.

```
function vestingWithdraw(
    address to,
    uint256 amount
) public nonReentrant onlyRole(VESTING_ADMIN_ROLE) returns (uint256) {
    if (amount > getVestingDeposit()) {
      revert VestingWithdrawAmountExceeded(amount, getVestingDeposit());
    }
    SafeERC20.safeTransfer(IERC20(asset()), to, amount);
    _vestingDeposit -= amount;
    emit VestingDeposit(to, amount);

    return amount;
}
```

Impact

This will be misleading for users and developers who are monitoring the contract events.

Recommendations

Emit a `VestingWithdraw` event instead of `VestingDeposit`.

Remediation

This issue has been acknowledged by Solera Markets, and a fix was implemented in PR #11 ↗.

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. Redemption often underpays user by one base unit

We do not consider this issue to have any impact because of how insignificant the underpayment is. However, we decided to document this behavior for the extremely unlikely scenario where this contract is configured with an asset that has few decimals and for the sake of completeness.

When determining whether the code rounds in the correct direction, we observed an interesting behavior: when redeeming, the contract often underpays what is due to the user by one base unit. The root cause is the fact that the fee is calculated (and rounded up) twice.

For the remainder of this section, we will use the following variables:

- Let n be the number of shares to redeem.
- Let x be n converted to assets at the time `requestRedeem` is called.
- Let m be the amount it transfers to the user.
- Let o be the amount it transfers to the fee receiver.

The following invariant should exist to ensure that there is no underpayment or overpayment: *the amount owed to the user, added to the amount owed to the fee receiver, must equal the shares passed in converted to assets in* **`redeem`***.*

That is, we want to test the invariant that $m + o = x$.

Converting the logic of the code,

- in `requestRedeem`

$$x = x = \text{sharesToAssets}(n)$$

- atomically in `redeem`,

$$m = x - \text{ceil}\left(x \times \frac{\text{fee}}{\text{fee} + \text{bps}}\right)$$

$$o = \text{ceil}\left(m \times \frac{\text{fee}}{\text{bps}}\right)$$

Verifying the invariant without rounding

Ignoring `ceil` for now,

$$m = x - \left(x \times \frac{\text{fee}}{\text{fee} + \text{bps}}\right)$$
$$m = x \times \frac{\text{bps}}{\text{fee} + \text{bps}}$$

and substituting `m` into `o`,

$$o = m \times \frac{\text{fee}}{\text{bps}}$$
$$o = x \times \frac{\text{bps}}{\text{fee} + \text{bps}} \times \frac{\text{fee}}{\text{bps}}$$
$$o = x \times \frac{\text{fee}}{\text{fee} + \text{bps}}$$

we now have this:

$$m = x \times \frac{\text{bps}}{\text{fee} + \text{bps}}$$
$$o = x \times \frac{\text{fee}}{\text{fee} + \text{bps}}$$

Checking the invariant,

$$m + o = x$$
$$\frac{\text{bps}}{\text{fee} + \text{bps}} + \frac{\text{fee}}{\text{fee} + \text{bps}} = 1$$
$$1 = 1$$

the invariant holds without rounding.

Testing the invariant with rounding

The following counterexample shows the invariant does not hold when considering rounding:[4]

$$n = 5, \quad x = 5$$

[4] Note that the fee we used was 90% for this specific example. However, the bug is not only triggered when the fees are high.

$$\mathsf{fee} = 9, \quad \mathsf{bps} = 10$$

$$\begin{aligned} m &= x - \mathsf{ceil}\left(x \times \frac{\mathsf{fee}}{\mathsf{fee} + \mathsf{bps}}\right) \\ &= 5 - \mathsf{ceil}\left(5 \times \frac{9}{9 + 10}\right) \\ &= 5 - 3 \\ &= 2 \end{aligned}$$

$$\begin{aligned} o &= \mathsf{ceil}\left(m \times \frac{\mathsf{fee}}{\mathsf{bps}}\right) \\ &= \mathsf{ceil}\left(2 \times \frac{9}{10}\right) \\ &= 2 \end{aligned}$$

$$\begin{aligned} m + o &= x \\ 2 + 2 &\neq 5 \end{aligned}$$

The invariant does not hold when rounding is considered. The user would be underpaid by one base unit.

Visualizing when the bug is triggered

We graphed the behavior on Desmos.com here ↗[5] to visualize which parameters trigger the bug. In the graph at the link, we use 20% fees. The x-axis represents the number of assets the user wants to redeem (x), and the y-axis represents the overpayment or underpayment.

Overpayments will never occur — only underpayments by exactly one base unit.

Underpayment by one base unit occurs only if the assets are a multiple of the BPS divided by the fee, or

$$x \left(\mathsf{mod}\, \frac{\mathsf{bps}}{\mathsf{fee}}\right) = 0$$

[5] The Desmos graph is continuous despite the function being discrete; fractional base units are impossible.

The bug is more likely to be triggered as the fee increases.

4.2. Regarding inflation attacks

Please see GitHub issue #3706 ↗ in the openzeppelin-contract repository for discussion about how to mitigate this attack.

In short, the first deposit to a new vault could be made by a trusted admin during vault construction to ensure that `totalSupply` remains greater than zero. However, this remediation has the drawback that this deposit is essentially locked, and it needs to be high enough relative to the first few legitimate deposits so that front-running them is unprofitable. However, even if this prevents the attack from being profitable, an attacker can still grief legitimate deposits with donations, making the user gain less shares than they should have gained.

Also, the ERC4626Upgradeable OpenZeppelin contract ↗ explains their proposed solution to this type of attack:

> The `_decimalsOffset()` corresponds to an offset in the decimal representation between the underlying asset's decimals and the vault decimals. This offset also determines the rate of virtual shares to virtual assets in the vault, which itself determines the initial exchange rate.
>
> While not fully preventing the attack, analysis shows that the default offset (0) makes it non-profitable, as a result of the value being captured by the virtual shares (out of the attacker's donation) matching the attacker's expected gains. With a larger offset, the attack becomes orders of magnitude more expensive than it is profitable.

We do not expect it to be an issue, but in general we recommend atomically deploying vaults and depositing some initial amount.

4.3. Test suite

While the tests appear to have good code coverage, we highly recommend writing more tests involving time manipulation (`vm.warp`) and more redemption scenarios.

Additionally, the contract lends itself to fuzz testing, which could help uncover edge cases that are not easy to discover with other dynamic testing.

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed.

During our assessment on the scoped Solera contracts, we discovered seven findings. No critical issues were found. Three findings were of high impact, one was of medium impact, and the remaining findings were informational in nature.

5.1. 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.