Zellic

Prepared for
Sam
Takara Lend

Prepared by
Ethan Lee
Kuilin Li
Zellic

January 21, 2025

Takara Lend Contracts

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 Takara Lend from January 10th to January 17th, 2025. During this engagement, Zellic reviewed Takara Lend contracts' 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:

- How do the changes that Takara Lend made to the contracts over the upstream Moonwell V2 contracts affect the threat model?
- Do the addition of the centralized borrow guardian, redeem guardian, and blacklist cause any security issues or unexpected outcomes?
- Is market deprecation implemented securely?
- Could an on-chain attacker take advantage of the new functionality where all mints cause the minter to enter all of the nondeprecated markets?
- Is the usage of the math library in line with the library's security expectations?

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 Takara Lend contracts, we discovered 11 findings. One critical issue was found. Two were of high impact, one was of medium impact, four were 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 Takara

Lend in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	1
🟧 High	2
🟨 Medium	1
🟩 Low	4
⬜ Informational	3

2. Introduction

2.1. About Takara Lend Contracts

Takara Lend contributed the following description of Takara Lend contracts:

> Takara is an open, native decentralized lending and borrowing platform on the Sei blockchain. With its simple interface, users can easily manage and grow their digital assets. The platform makes better use of capital while making DeFi more accessible to everyone.

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:

Takara Lend Contracts

Type	Solidity
Platform	EVM-compatible

Target	Takara_Contract
Repository	https://github.com/Takara-Lend/Takara_Contract ↗
Version	`8098516cfcd8bbb06c2fc432458043f39302cc8f`

Programs	`Comptroller.sol`
	`ErrorReporter.sol`
	`TErc20.sol`
	`TErc20Delegator.sol`
	`TToken.sol`
	`TTokenInterfaces.sol`
	`irm/JumpRateModel.sol`
	`Rate.sol`
	`oracles/Api3Aggregator.sol`
	`oracles/Api3LinkedAggregator.sol`
	`oracles/CompositeOracle.sol`
	`oracles/LinkedAssetAggregator.sol`
	`utils/Addresses.sol`
	`utils/AssetDeployer.sol`
	`utils/MarketState.sol`

2.4. Project Overview

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

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:

Ethan Lee
Engineer
ethl@zellic.io ↗

Kuilin Li
Engineer
kuilin@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 10, 2025	Start of primary review period
January 13, 2025	Kick-off call
January 17, 2025	End of primary review period

3. Detailed Findings

3.1. Debts owed by blacklisted users cannot be liquidated

Target	Comptroller		
Category	Business Logic	**Severity**	Critical
Likelihood	Medium	**Impact**	Critical

Description

The Comptroller contract controls and restricts actions that users can take on the lending platform. One of the modifications made to this contract is the new implementation of an address blacklist, stored in the Comptroller. Admins can add addresses to the blacklist using the `_setBlacklist` external function, and, if an address is blacklisted, actions such as minting and borrowing are rejected by the Comptroller:

```
function mintAllowed(address tToken, address minter, uint256 mintAmount)
    external override returns (uint256) {
    // [...]

    if (blackList[minter]) {
        return uint256(Error.REJECTION);
    }

// [...]
function borrowAllowed(address tToken, address borrower, uint256 borrowAmount)
    external
    override
    returns (uint256)
{
    // [...]

    if (blackList[borrower]) {
        return uint256(Error.REJECTION);
    }
```

Out of all of the permissions-check functions in the Comptroller, the blacklist check was added to `mintAllowed`, `redeemAllowed` (via `redeemAllowedInternal`), `transferAllowed` (via `redeemAllowedInternal`), `borrowAllowed`, and `repayBorrowAllowed`. On the other hand, the check was not added to `liquidateBorrowAllowed` and `seizeAllowed`. This omission was intended to allow positions held by blacklisted addresses to be liquidated.

However, during the actual execution of a liquidation, the business logic actually performs a check that is rejected due to the blacklist. This can be seen in `TToken.liquidateBorrowFresh`, which calls `repayBorrowFresh`, which then calls `Comptroller.repayBorrowAllowed`, which uses the blacklist.

Impact

Positions held by blacklisted addresses cannot be liquidated and will continue accruing interest until they are unblacklisted, even as the prices of the underlying assets change over time. This can result in an unresolvable protocol skew that threatens the solvency of the protocol.

Recommendations

We recommend reviewing the liquidation workflow and adjusting the modifications accordingly in order to allow for the liquidation of a blacklisted position. An alternative design option would be to add functionality that allows admins to immediately liquidate or repay positions that would be blacklisted and to always do that first so that blacklisted addresses do not hold any positions.

In any case, we recommend adding comprehensive tests so that this and any other divergence between expected and actual functionality can be detected without manual review.

Remediation

This issue has been acknowledged by Takara Lend, and a fix was implemented in commit `4f09afa0` ↗.

3.2. AssetDeployer deploys tokens that are vulnerable to inflation attacks

Target	AssetDeployer		
Category	Business Logic	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The AssetDeployer contract deploys the TErc20 contract and sets up the contract's metadata in the Comptroller. During the setup process, 1 Wei of the token is minted with a payment of 1 Wei of the underlying and then burned to the zero address:

```
function deployAsset(
    // [...]
) public returns (address) {
    // [...]

    underlyingERC20.approve(address(tToken), 1);
    TErc20Delegator(payable(address(tToken))).mint(1);
    tToken.approve(address(0), 1);
    tToken.transfer(address(0), 1);
```

Note that the parameter to `TErc20.mint` is the amount of underlying, but the amount of tToken minted may be greater than 1 Wei, depending on the initial exchange rate. So this may leave tToken value in the AssetDeployer contract.

Although minting 1 Wei does initialize the contract, the exchange rate that it uses after initialization is dependent on the live balance of the underlying token that is owned by the contract. Thus, it is vulnerable to ERC-4626 inflation–style attacks, where an attacker front-runs the first real mint in order to donate underlying assets and change the exchange rate into one that causes the victim to mint at an unfavorably rounded price.

Impact

If an ERC-4626 inflation–style attack is conducted shortly after the deployment of a legitimate asset, then the first few depositors are at risk of front-runners.

Recommendations

We recommend minting and burning a real amount of value and setting the exchange rate so that this real amount of value corresponds to an appropriate amount of shares. The feasibility of an ERC-4626 inflation–style attack is diminished if the initial amount of shares is high, because it relies on the rounding behavior when there are only a few shares.

Alternatively, instead of burning this value, the admin could hold onto it until the protocol has sufficient standing liquidity such that it is not plausible for everyone to withdraw, at which point the admin can redeem those tokens.

Either way, we recommend reviewing the deployment strategies of other Compound V2 forks in order to ensure that the deployment is not arbitrageable by attackers who try to employ ERC-4626 inflation–style attacks.

Remediation

This issue has been mitigated by deprecating this AssetDeployer. In the new deployment process, the team will make sure that at least 10e6 Wei of underlying shares exist at the time the mint function becomes publicly available.

3.3. Lack of freshness check in Api3Aggregator and LinkedAssetAggregator

Target	Api3Aggregator, LinkedAssetAggregator		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The functions `Api3Aggregator.latestRoundData()` and `LinkedAssetAggregator.latestRoundData()` currently lack checks to verify the freshness of the retrieved price data. While Api3LinkedAggregator has a freshness check for the exchange-rate feed, it does not do the same verification for the timestamps returned by `originPriceFeed.latestRoundData()`:

```
function latestRoundData()
    public
    view
    override
    returns (/* [...] */)
{
    (int224 value, uint32 exchangeRateTimestamp) = exchangeRateFeed.read();
    uint256 duration = uint256(exchangeRateTimestamp) - block.timestamp;

    require(duration > freshCheck, "Not valid price");

    (uint80 tokenRoundId, int256 ethPrice,,, uint80 tokenAnsweredInRound)
    = originPriceFeed.latestRoundData();

    // [...]
}
```

This means that the affected aggregators could return outdated price information if the original price feed has a problem, which may mislead the protocol into relying on it as if it was an accurate and up-to-date price.

Impact

The absence of freshness checks could lead to the use of outdated price data. It can potentially cause loss of funds in the protocol.

Recommendations

Add checks to verify the freshness of the price data retrieved from the original price feed in Api3Aggregator and LinkedAssetAggregator.

Remediation

This issue has been acknowledged by Takara Lend, and a fix was implemented in commit 3d738974 ↗.

3.4. Underlying tokens can be swept by the admin

Target	TErc20		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

In the TErc20 contract, the `sweepToken` function transfers unrelated tokens or tokens that may have accidentally been sent to the contract address, to the admin. This change was implemented in the following function:

```
function sweepToken(EIP20NonStandardInterface token) external override {
    require(msg.sender == admin, "TErc20::sweepToken: only admin can sweep
    tokens");
   require(address(token) != underlying, "TErc20::sweepToken: can not sweep
       underlying token");
   require(address(token) == underlying, "TErc20::sweepToken: can not sweep
       underlying token");
    uint256 balance = token.balanceOf(address(this));
    token.transfer(admin, balance);
}
```

This change requires that the swept token be equal to the underlying token, instead of requiring it to be the unrelated token. Since the rest of the function, including the interface and the string error message, was not changed, this appears to be an unintentional edit or an edit made to facilitate testing instead of implementing an intentional part of the design.

Impact

The swept token is no longer an unrelated token. This significantly changes the purpose of this function, because withdrawing the underlying token increases the risk that the contract becomes insolvent.

Recommendations

We recommend changing this back to properly check that the swept token is not the underlying token.

Alternatively, if the design intent is to allow the admin to freely withdraw liquidity from the contract, then we recommend adding another function to clearly document this design decision. In this case, the original `sweepToken` functionality is still useful, as unrelated tokens can always accidentally be transferred to the contract. Additionally, documenting the addition of this admin function communicates the design decision to the user, increasing clarity and transparency about the amount of centralization that this fork implements.

Remediation

This issue has been acknowledged by Takara Lend, and a fix was implemented in commit `fa302d10` ↗.

3.5. Markets entered automatically can be exited

Target	Comptroller		
Category	Business Logic	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

In the Comptroller contract, `enterAllMarkets` is a newly added function that enters all of the non-deprecated markets upon request from a TToken contract:

```
function enterAllMarkets(address account) public override returns (uint256[]
    memory) {
    address[] memory assetsAddresses = new address[](allMarkets.length);

    bool isRTokenContract;

    for (uint256 i = 0; i < allMarkets.length; i++) {
        assetsAddresses[i] = address(allMarkets[i]);

        if (assetsAddresses[i] == msg.sender) {
            isRTokenContract = true;
        }
    }

    require(isRTokenContract, "Sender must be a TToken contract");

    uint256 len = assetsAddresses.length;

    uint256[] memory results = new uint256[](len);

    for (uint256 i = 0; i < len; i++) {
        // Prevent paused asset enter market
        if (borrowGuardianPaused[assetsAddresses[i]]) continue;

        TToken tToken = TToken(assetsAddresses[i]);
        results[i] = uint256(addToMarketInternal(tToken, account));
    }
    return results;
}
```

This function is called in TToken whenever someone mints assets:

```
function mintFresh(address minter, uint256 mintAmount)
    internal returns (uint256, uint256) {
    // [...]

   comptroller.enterAllMarkets(minter);

    return (uint256(Error.NO_ERROR), vars.actualMintAmount);
}
```

However, although this additional call makes depositors enter all the markets, the `exitMarket` function was not changed at all, and users can still exit markets for which they do not have any borrow balance.

Impact

Users may bundle transactions or operate through a smart wallet in order to immediately exit all the markets they do not wish to stay entered in, after minting assets. This reduces this contract change from a restriction to a convenience, because the action can be undone transactionally.

Additionally, this implementation favors sophisticated users, because regular users will either not realize that they can exit markets they do not wish to stay entered in or exit in a separate manually submitted public-mempool transaction, which risks a liquidation occuring before the exit is confirmed.

Recommendations

If remaining in all nondeprecated markets is mandatory, then we recommend disabling `exitMarket` for the markets that `enterAllMarkets` enters.

Alternatively, if this is a single-transaction convenience, then we recommend adding a parameter to the mint function that allows users to opt out of this automatic entry, so that unsophisticated users have the same easy ability to opt out of this convenience as sophisticated ones. Another way to implement this option, if the function interface for minting should not be changed from the upstream implementation, is to have an account-specific setting that users can arbitrarily change by calling a function on the Comptroller contract.

Remediation

This issue has been acknowledged by Takara Lend, and a fix was implemented in commit `7ee9b938` ↗. The `enterAllMarkets` function has been removed.

3.6. Infinite token approval is triggered outside the spec

Target	Rate		
Category	Protocol Risks	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The Rate contract is an ERC-20–compliant token that implements EIP-717 unlimited authorization for permits and regular allowances:

```
function approve(address spender, uint256 rawAmount) external returns (bool) {
    uint96 amount;
    if (rawAmount == type(uint256).max) {
        amount = type(uint96).max;
    } else {
        amount = safe96(rawAmount, "Rate::approve: amount exceeds 96 bits");
    }

    allowances[msg.sender][spender] = amount;

    emit Approval(msg.sender, spender, amount);
    return true;
}
```

However, the recordkeeping of token amounts is implemented in `uint96`, instead of `uint256`. This implementation decision leaks a corner case into the interface due to the overlap in meaning between unlimited approval and the maximum amount of approval. If the caller sets the `uint256 rawAmount` parameter to exactly `type(uint96).max`, then the amount approved will be unlimited, instead of limited to `type(uint96).max`.

The unadopted EIP-717 specification ↗ is the current standard for unlimited token allowance, and it specifies exactly `type(uint256).max` as the only sentinel allowance value that means unlimited allowance. So, although it is unlikely, a caller that manipulates arbitrary ERC-20 tokens under the assumption that EIP-717 is strictly how unlimited approvals work can be misled into unintentionally setting unlimited allowance for the Rate token.

Impact

The impact is fairly low, since the token-balance recordkeeping is done in `uint96`, which means that an allowance that is that high can be interpreted as effectively infinite since it is large relative to the total supply. However, the impact is not zero, as this edge case may contribute to DeFi-exploit chains that include two or more independent systems.

Recommendations

In order to fully comply with the specification, except for allowance and token balances not representable as `uint96`, an extra boolean flag must be added to the state to represent unlimited approval.

Alternatively, the representable region can be reduced by one, down to `type(uint96).max - 1`. This can be done by slightly tightening the range of the `safe96` internal function and adding associated logic.

Remediation

This issue has been acknowledged by Takara Lend, and a fix was implemented in commit `43c32ffc` ↗. The Rate token has been deprecated.

3.7. Nonce collision in `permit` and `delegateBySig` functions

Target	Rate		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

In the Rate token implementation, both `permit()` and `delegateBySig()` share the same nonce for signature verification:

```
function permit(address owner, address spender, uint256 rawAmount,
    uint256 deadline, uint8 v, bytes32 r, bytes32 s)
        external
{
    // ...
    bytes32 domainSeparator =
        keccak256(abi.encode(DOMAIN_TYPEHASH, keccak256(bytes(name)),
    getChainId(), address(this)));
    bytes32 structHash =
        keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, rawAmount,
    nonces[owner]++, deadline)); // nonce is same for both permit and
    delegateBySig
    // ...

    allowances[owner][spender] = amount;

    emit Approval(owner, spender, amount);
}

function delegateBySig(address delegatee, uint256 nonce, uint256 expiry,
    uint8 v, bytes32 r, bytes32 s) public {
    // ...
    require(nonce == nonces[signatory]++, "Rate::delegateBySig: invalid
    nonce");  // nonce is same for both permit and delegateBySig
    require(block.timestamp <= expiry, "Rate::delegateBySig: signature
    expired");
    return _delegate(signatory, delegatee);
}
```

Note that the two signatures operate in different signature domains yet share the same nonce. This could lead to front-end confusion, since normally, a nonce is tied to a specific signature domain and incremented only when a signature in that domain is consumed.

So, if a user with a zero nonce wishes to do both actions at the same time, a front-end UI that operates under standardized assumptions will sign both messages with the zero nonce. This means that only the first such signature will work and the second will fail since the check will expect the nonce to be one.

Recommendations

Separate the nonce for `permit()` and `delegateBySig()` to avoid front-end confusion.

Remediation

This issue has been acknowledged by Takara Lend, and a fix was implemented in commit `43c32ffc` ↗.. The Rate token has been deprecated.

3.8. Potential underflow in `getUnderlyingPrice` due to high decimals

Target	Api3Aggregator		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

In the `CompositeOracle.getUnderlyingPrice()` function, the calculation of the price includes a scaling operation that may lead to underflow errors if the underlying token has decimals higher than 18 (such as YAM-V2):

```
function getUnderlyingScaledPrice(TToken tToken)
    internal view returns (uint256 price) {
    // [...]

    price = scalePrice(rawPrice, feedDecimals, decimals);
    price = price * 10 ** (36 - 2 * decimals);
```

The line `price = price * 10 ** (36 - 2 * decimals);` performs a multiplication based on the token's decimals. If the decimals value exceeds 18, the operation may attempt to perform calculations with values that result in underflow, causing the function to revert.

Impact

This vulnerability can lead to function reverts if tokens have more than 18 decimals. The protocol would not be able to get the correct price, and the market would not work as expected.

Recommendations

Document the lack of support for tokens with more than 18 decimals, or scale the price differently to accommodate those tokens.

Remediation

This issue has been acknowledged by Takara Lend. Takara Lend will carefully review this aspect to avoid using assets with precision higher than 18 decimals.

3.9. Api3Aggregator returns a duration for the `updatedAt` field

Target	Api3Aggregator, CompositeOracle		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

All of the oracles implement the `latestRoundData` function, which is defined in the AggregatorV3Interface:

```
function latestRoundData()
    external
    view
    returns (uint80 roundId, int256 answer, uint256 startedAt,
    uint256 updatedAt, uint80 answeredInRound);
```

This interface is set by Chainlink, whose documentation ↗ specifies that the `startedAt` and `updatedAt` values are timestamps.

However, in the Api3Aggregator function, the returned `updatedAt` value is a duration:

```
function latestRoundData()
    public
    view
    override
    returns (uint80 roundId, int256 answer, uint256 startedAt,
    uint256 updatedAt, uint80 answeredInRound)
{
    (int224 value, uint32 timestamp) = originPriceFeed.read();
    uint256 updatedAt_ = uint256(timestamp);
    int256 scaledTokenPrice = int256(int224(uint224(value)));
    return (uint80(1), scaledTokenPrice, updatedAt_, block.timestamp
    - updatedAt_, uint80(0));
}
```

Note that in the above code, the local variable `updatedAt_` is a timestamp that is copied into the `startedAt` return value, and the `updatedAt` return value is calculated as `block.timestamp - updatedAt_`.

On the other hand, in the Api3LinkedAggregator contract, the `updatedAt` field is assigned a timestamp:

```
function latestRoundData()
    public
    view
    override
    returns (uint80 roundId, int256 answer, uint256 startedAt,
    uint256 updatedAt, uint80 answeredInRound)
{
    (int224 value, uint32 exchangeRateTimestamp) = exchangeRateFeed.read();

    // [...]

    return (
        tokenRoundId,
        finalPrice,
        uint256(exchangeRateTimestamp),
        uint256(exchangeRateTimestamp),
        tokenAnsweredInRound
    );
}
```

And, in the CompositeOracle contract, the `updatedAt` field is used as if it is a timestamp:

```
(, int256 answer,, uint256 updatedAt,) = abi.decode(data, (uint80, int256,
    uint256, uint256, uint80));
if (answer <= 0 || (block.timestamp - updatedAt) >= freshCheck) { // skip
    iteration if price is outdated.
    continue;
}
```

So in contrast with the second and third examples, the first example incorrectly assigns that return field.

Impact

There is currently no impact, because in the audited, frozen version of these contracts, the Api3Aggregator contract is unused, and the only oracle in use is the CompositeOracle, with underlying mock Chainlink oracles.

However, this issue represents a significant footgun for future oracle setups. For example, if a CompositeOracle is deployed whose aggregators include an Api3Aggregator, then the returned `updatedAt` value will be `block.timestamp - updatedAt_`, which would cause the comparison `(block.timestamp - updatedAt) >= freshCheck` to check `updatedAt_ >= freshCheck`. This is a comparison between a timestamp and a duration, so the duration will be interpreted as a timestamp shortly after January 1st, 1970. Thus, it will erroneously skip the current iteration and cause the protocol to retrieve suboptimal price data.

Recommendations

Fix the `latestRoundData()` function in Api3Aggregator to return the `updatedAt` value as a timestamp.

Remediation

This issue has been acknowledged by Takara Lend, and a fix was implemented in commit `a5677e7b` ↗.

3.10. Missing math-error handling in `redeemFresh` function

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

Description

In the `TToken.redeemFresh()` function, the lines that call `divScalarByExpTruncate()` assign a math error to `vars.mathErr`, but this error is not checked before it is overwritten by the subsequent math operation:

```
(vars.mathErr, vars.redeemTokens) =
    divScalarByExpTruncate(redeemAmountIn, Exp({mantissa: vars.
        exchangeRateMantissa}));

(vars.mathErr, vars.redeemAmount) =
    mulScalarTruncate(Exp({mantissa: vars.exchangeRateMantissa}),
         vars.redeemTokens);

if (vars.mathErr != MathError.NO_ERROR) {
    return failOpaque(
        Error.MATH_ERROR, FailureInfo.REDEEM_EXCHANGE_AMOUNT_
            CALCULATION_FAILED, uint256(vars.mathErr)
    );
```

As a result, any math error produced during this division could be silently ignored and leads to possible incorrect calculations and unintended behaviors.

Impact

In the case where the caller is malicious, there is no security impact, because this occurs in a branch where `redeemTokensIn` is zero, and so the first division is used to determine `vars.redeemTokens` automatically based on on-chain values. In the other branch, at this step in the workflow, `vars.redeemTokens` can be arbitrarily set by the caller. So, even if a math error occurs here, the impact would only be to allow the caller to arbitrarily set something they can already intentionally do. The token-balance check happens in `redeemAllowed`, which is called later.

In the case where the caller is not malicious but a third party is malicious, there is still no security impact. This is because a caller that calls this function with a zero `redeemTokensIn` quantity is already expecting the protocol to maximize the amount of tokens they redeem. So, any amount of actually redeemed tokens less than the maximum they are entitled to, due to an ongoing attack, would likely be better than the amount of tokens they can redeem after the attack completes.

Recommendations

Even though there is no security impact, we recommend exiting with an error if this calculation fails, so that unexpected behavior does not silently occur.

Remediation

This issue has been acknowledged by Takara Lend, and a fix was implemented in commit `571c9790` ↗.

3.11. Lack of event emissions for configuration changes

Target	Comptroller, CompositeOracle		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Compared to the upstream, this project adds a few significant admin-only configuration change functions, such as these:

```
function updateLiquidateWhiteList(address user, bool state) public {
    require(msg.sender == admin, "Unauthorized");

    if (state) {
        require(!liquidatorWhiteList[user], "User is already in the white
    list");
    }

    liquidatorWhiteList[user] = state;
    // No event emission
}

function triggerLiquidation(bool state) public returns (bool) {
    require(msg.sender == pauseGuardian || msg.sender == admin, "only pause
    guardian and admin can pause");
    require(msg.sender == admin || state == true, "only admin can unpause");

    liquidatable = state;
    // No event emission
    return state;
}
```

For DeFi protocols, it is common practice to emit events when significant configuration parameters are changed by the protocol admin. Event logging is essential for tracking state changes and maintaining transparency within the protocol.

Impact

The absence of event emission makes it more difficult for off-chain automation and end users to monitor changes in the configuration.

Recommendations

We recommend adding event emissions to the functions that change the protocol's configuration. Specifically, we believe these functions in these contracts should have associated events:

- In Comptroller, `updateLiquidateWhiteList()`, `triggerLiquidation()`, `_setBlackList()`, and `_setProtocolPaused()`
- In CompositeOracle, `setL2Aggregators()`

This will help to maintain transparency and allow off-chain systems to monitor changes.

Remediation

This issue has been acknowledged by Takara Lend, and a fix was implemented in commit `6ef5c868` ↗.

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. Misleadingly named `liquidatable` global flag

In the Comptroller contract, the global boolean state variable `liquidatable` was implemented to control the functionality of the liquidation whitelist.

This variable is named in a misleading way. Users who rely on the common definition of the term "liquidatable" will assume, at the very least, that if the variable is true, then liquidations are more likely or more possible than if the variable were false.

However, what this variable actually controls is whether the liquidation process is whitelisted:

```
function liquidateBorrowAllowed(
    // [...]
) external view override returns (uint256) {
    if (liquidatable && !liquidatorWhiteList[liquidator]) {
        return uint256(Error.UNAUTHORIZED);
    }

    // [...]
```

This means that if the flag is true, then only liquidators in the `liquidatorWhiteList` may liquidate positions, and if the flag is false, then the whitelist is disabled and anyone can liquidate positions. So, `liquidatable` being false strictly increases the possibility of liquidations.

We recommend renaming this variable to `liquidatorWhiteListEnabled` in order to reflect the actual purpose of this flag.

4.2. Unused added constants in TTokenStorage

In the TTokenStorage contract, implemented in the file TTokenInterfaces.sol, the following two constants were added:

```
contract TTokenStorage {
    // [...]

  // @notice Maximum fraction of redemption factor that can be set aside
      for reserves
    uint256 internal constant redemptionReserveFactorMaxMantissa = 0.05 ether;

    uint256 internal constant minRedemptionCashRequire = 1 wei;


    // [...]
}
```

These constants are unused in the rest of the project, so we recommend removing them in order to not make misleading assertions about what restrictions this protocol implements.

4.3. Potential revert griefing in the liquidation flow

The `liquidateBorrowAllowed` function in the Comptroller contract is called when an account may be liquidated, and it checks if the liquidation is allowed:

```
function liquidateBorrowAllowed(
    address tTokenBorrowed,
    address tTokenCollateral,
    address liquidator,
    address borrower,
    uint256 repayAmount
) external view override returns (uint256) {
    // [...]

   /* allow accounts to be liquidated if the market is deprecated */
   if (isDeprecated(TToken(tTokenBorrowed))) {
       require(borrowBalance >= repayAmount, "Can not repay more than the
           total borrow");
   } else {
       // [...]
   }
    return uint256(Error.NO_ERROR);
}
```

This logic determines whether a liquidation should proceed based on several state conditions such as including market status and account liquidity. However, the newly added require check (`require(borrowBalance >= repayAmount)`) in the case of deprecated tTokens introduces a potential griefing risk. This check prevents liquidation of more than the total borrowed amount, but this enables borrowers to bypass liquidation if the borrower executes a self-liquidation or repayment of a smaller amount to decrease `borrowBalance` before the liquidation call. This results in a revert and unnecessary gas costs for the liquidator.

Although this is concerning, we noted this as a discussion point instead of a finding because the upstream code contains the same logic and assumption, where if `liquidateBorrowAllowed` is called with an excess `repayAmount`, it reverts. This is likely acceptable because, in the long run, liquidators would be sophisticated market participants and can be expected to either liquidate from a contract or have off-chain arrangements with block builders to ensure the state is consistent with their expectations before the liquidation is requested.

5. System Design

This provides a description of the additions and modifications to the high-level components of the system, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Component: Comptroller

Description

The Comptroller contract is called by other components of the system to centrally implement restrictions and liquidity calculations. It also manages cross-liquidity, which is the automatic sharing of collateral, along with liquidatability, between all of a user's longs and shorts.

Modifications

- An address blacklist was implemented, and blacklist checks were added to most of the operations. Blacklisted addresses cannot mint, redeem, borrow, or repay. The positions owned by blacklisted addresses should, as they accrue interest, eventually be liquidatable. However, Finding 3.1. ↗ currently makes this impossible.
- A borrow guardian and a redeem guardian were added. Both the `pauseGuardian` role and the `admin` role can separately pause borrowing and redeeming on specific TTokens.
- An optional liquidator whitelist was added. When a flag on the Comptroller is enabled, the whitelist prevents nonwhitelisted liquidators from liquidating any positions.
- The function `enterAllMarkets` that whitelisted markets can call was added, which causes a minter to be added to all of the nondeprecated markets managed by this Comptroller.

Test coverage

Cases covered

- `testBlackList()`
  - Return error in `mint()` if the minter is blacklisted.
  - Return error in `redeem()` if the redeemer is blacklisted.
  - Return error in `borrow()` if the borrower is blacklisted.
  - Successfully mint if the minter is not blacklisted.
- `testRedemptionPaused()`
  - Successfully redeem if the redeem is not paused.
- `enterAllMarkets()`
  - Enter all markets.
  - Revert if the sender is not a TToken contract.

- `testGlobalPaused()`
    - Revert in `mint()` if mint is paused.
    - Revert in `redeem()` if redeem is paused.
    - Revert in `borrow()` if borrow is paused.
- `testProtocolProtectedAccount()`
    - Revert in `triggerLiquidation()` if the caller is neither the admin nor pause guardian.

Cases not covered

- No test for `enterMarkets()` if `borrowGuardianPaused` is true.
- No test for `redeem()` if `redeemAllowedInternal` is true.
- No test for `enterAllMarkets()` if borrowing is paused.
- No test for the liquidation if the market is deprecated.
- No test for the liquidation if `liquidatable` is enabled and the liquidator is whitelisted.
- Incorrect test-result check if `liquidatable` is enabled and the liquidator is not whitelisted.

Attack surface

Blacklist or pause feature may be used to bypass the protocol's functionality.

5.2. Component: ErrorReporter

Description

The ErrorReporter contract contains the error definitions.

Modifications

Additional enum options were added to support functionality related to the additional logic.

Test coverage

N/A.

Attack surface

The attack surface is unchanged from the upstream implementation.

5.3. Components: TErc20Delegator, TErc20, and TToken

Description

The TErc20 contract implements the market, including minting and borrowing, for an ERC-20 token as the underlying asset. It depends on the abstract TToken contract and serves as an implementation contract for the TErc20Delegator contract to delegate to.

Modifications

- In the TErc20 contract, the `sweepToken` function was changed, likely unintentionally. See Finding 3.4. ↗.
- The interest-rate calculation, including the variables `borrowRatePerTimestamp` and `supplyRatePerTimestamp`, were changed to be accounted per block, instead of per second.
- A receive handler was added to the TErc20Delegator contract that allows zero-value empty calldata calls.

The following modifications were all made to the TToken contract:

- Checks were added to `transfer` and `transferFrom` to not allow anyone but the admin to transfer tokens to the token address itself.
- A call was added to the new `Comptroller.enterAllMarkets` function in order to enter all of the markets on behalf of anyone who mints tokens.
- Removes the feature where a redemption of `type(uint256).max` tokens automatically selects the maximum amount of redeemable tokens to redeem, in order to simplify the redemption math.

Test coverage

No tests.

Attack surface

The attack surface is unchanged from the upstream implementation.

5.4. Component: JumpRateModel

Description

The JumpRateModel contract implements an interest-rate model present in the upstream Compound contracts.

Modifications

Calls to the SafeMath math library were replaced with native math operations.

Test coverage

N/A.

Attack surface

The attack surface is unchanged from the upstream implementation.

5.5. Component: Rate

Description

The Rate contract is the contract for the Takara Lend token, with symbol TKL. It implements the ERC-20 interface itself, without any dependencies. It also tracks delegation and voting. The holder of a token can delegate that token to any address, including themselves, and then that address's voting power is increased by the balance. This delegation is reduced or increased seamlessly when tokens are transferred. Checkpoints are also implemented so that binary search can be conducted to determine any voter's past voting power for any arbitrary block.

Invariants

- Typical ERC-20 token invariants should hold, including the correctness of the `totalSupply` across all operations, token-balance conservation across transfers, and so on.
- A token owner can only increase the voting power of a delegatee; they cannot, through any sequence of token transfers, affect or disrupt the delegatee in any way.
- The checkpoint system always correctly reflects the voting-power history of a particular delegatee.

Modifications

No modifications were made to the Rate contract since this is based on the Moonwell's (Well) token.

Test coverage

No tests.

Attack surface

The attack surface is unchanged from the upstream implementation.

5.6. Component: Oracle contracts

Description

Four oracle-related contracts were added. They are the Api3Aggregator, Api3LinkedAggregator, LinkedAssetAggregator, and CompositeOracle. The aggregator contracts all implement the AggregatorV3Interface, which is an interface defined by Chainlink that contains the view function `latestRoundData`, which returns the current price and staleness from a particular data source. The CompositeOracle maintains a list of aggregators and queries fallback aggregators farther down the list if any are down or stale.

Invariants

- The oracle contracts should always return a price that is within a reasonable range of the true price.
- The oracle contracts should not return outdated prices.
- The oracle contracts should not return until a grace period has passed since the sequencer came back online.

Test coverage

No tests.

Attack surface

- The oracle contracts may be manipulated to return a price that is not within a reasonable range of the true price.
- Incorrect freshness checks could lead to the use of outdated price data, potentially causing a loss of funds in the protocol.

5.7. Component: Utility contracts

Description

The AssetDeployer contract is used for the deployment and configuration of markets. The Addresses contract serves as a repository for managing contract addresses across different networks for the deployment. The MarketState contract shows information about all active markets, including current operational status such as LTV, TVL, and interest rate.

Invariants

- AssetDeployer should create and configure markets securely.
- Addresses should manage contract addresses correctly.
- MarketState should show correct information about all active markets.

Test coverage

Cases covered

- Create a market contract.
- Configure a new market and mint tToken.
- Successfully be able to harvest rewards.

Attack surface

The market may be vulnerable to an inflation attack.

6. Assessment Results

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

During our assessment on the scoped Takara Lend contracts, we discovered 11 findings. One critical issue was found. Two were of high impact, one was of medium impact, four were 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.