Zellic

Prepared for

Suilend

Prepared by

Juchang Lee

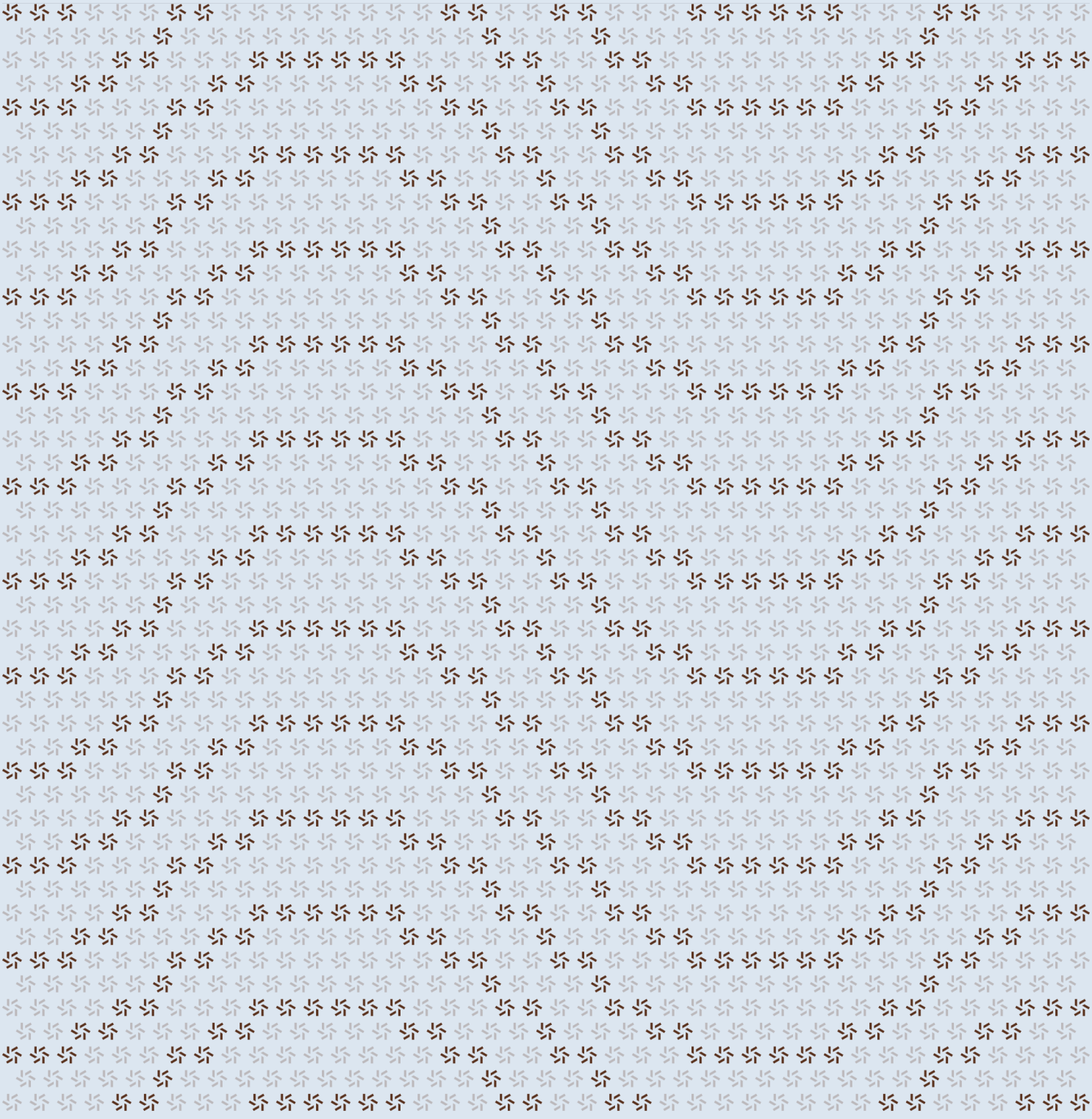
Junyi Wang

Zellic

March 5, 2024

Suilend

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 Suilend 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. No mechanism for debt write-off 11

3.2. Rate limiter can be abused 13

4. Discussion **13**

4.1. Centralization 14

4.2. Use of APR in configuration 14

4.3. Liquidate maximum percentage is weighted 15

4.4. Desync between reserve and obligation interest 16

5. Threat Model **17**

5.1. Module: `lending_market` 18

5.2. Component: Reserve 19

5.3. Component: Obligation 21

6. Assessment Results **22**

6.1. Disclaimer 23

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 Suilend from February 14th to February 26th, 2024. During this engagement, Zellic reviewed Suilend'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 permission-control issues stemming from a misunderstanding of Sui?
- Can an attacker steal funds through a rounding issue?
- Are there issues with share minting/redemption causing loss of funds?
- Are there lending- or liquidation-related bugs leading to bad debt?
- Is there any way to cause a protocol lockup?

1.3. Non-goals and Limitations

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

- Front-end components
- Key custody for the account holding the `LendingMarketCap`
- The reliability of the Pyth oracle

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

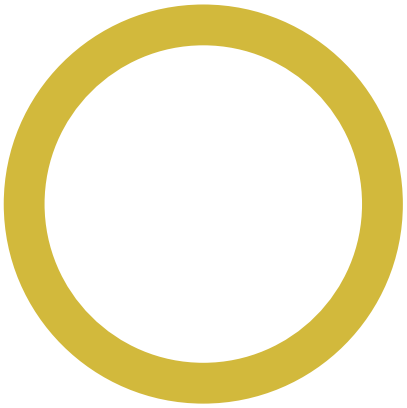
1.4. Results

During our assessment on the scoped Suilend modules, we discovered two findings, both of which were medium impact.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	2
Low	0
Informational	0



2. Introduction

2.1. About Suilend

Suilend is a lending protocol on the Sui blockchain. Users can deposit their funds to earn interest and borrow other assets with interest.

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 modules.
>
> **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 modules 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:

Suilend Modules

Repository	https://github.com/solendprotocol/suilend ↗
Version	suilend: `c6528ffcfd8c4bc678e0edf1ec71893e6f1911e7`
Programs	• contract/sources/cell.move • contract/sources/decimal.move • contract/sources/launch.move • contract/sources/lending_market.move • contract/sources/obligation.move • contract/sources/oracles.move • contract/sources/rate_limiter.move • contract/sources/reserve.move • contract/sources/reserve_config.move
Type	Move
Platform	Sui

2.4. Project Overview

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

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Juchang Lee
Engineer
lee@zellic.io ↗

Junyi Wang
Engineer
Junyi@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 14, 2024	Start of primary review period
February 15, 2024	Kick-off call
February 26, 2024	End of primary review period

3. Detailed Findings

3.1. No mechanism for debt write-off

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

Description

The protocol allows users to take on collateralized debt. Normally, users are required to be over-collateralized. If there is a rapid price change, it is possible for a user to have more debts than assets. When liquidating, the liquidator pays off the user's debt in return for the user's collateral assets. However, if the user runs out of collateral assets, there is no incentive to liquidate. In fact, the liquidate function cannot be used without user assets. There is no incentive for the user to repay the debt either, since the user will not receive collateral back.

Normally, this would simply be a loss absorbed by the protocol. However, there is no deduction of bad debts from the total pool assets (the debt is not forgiven).

The protocol mints cTokens, which represent shares in a pool. The worth of a cToken is based on the amount of assets the pool holds, including both assets currently held and assets lent out. Since the bad debt is not deducted from the assets in the pool, the worth of a cToken does not reflect bad debt. However, there are no assets to back this extra accounted value.

Impact

The bad debts continue to be counted as assets of the pool and continue to accrue interest, even if they may never be repaid. In this situation, cTokens are overvalued. This situation is similar to a bank run, and users redeeming their cTokens too late will not be able to redeem.

For simplicity, interest is 0% and there is no liquidation bonus or fees, and the debt-to-asset ratio can be up to 1 in the following scenario:

- Price of A is 1 USD; price of B is 2 USD.
- User X deposits 100 A.
- User Y deposits 50 B.
- User Z deposits 50 B.
- User A borrows 50 B.
- Price of B increases to 4 USD.
- User X is now liquidatable, with 100 A in assets.
- User L liquidates user X, paying 25 B and taking 100 A from the protocol.
- There is now a total of 0 A and 75 B in the protocol.

- The protocol considers itself to have 75 B available and 25 B lent out, and it does not revalue shares.
- User Y withdraws 50 B using Y's shares, worth 50 B.
- User Z is now unable to withdraw 50 B, despite having shares worth 50 B.

Note that if any other users were in the pool for B at this time, it would make sense for them to immediately withdraw after the creation of bad debt, to not end up in User Z's situation.

The `lending_market` includes a rate limiter which can stop this situation, depending on the parameters set. However, it relies on a centralized entity to set the correct parameters, and also to repay the bad debt in time.

Recommendations

Add a mechanism for forgiving bad debt. Bad debt can be assumed to be never repaid, since it is not in the user's interest to do so. Forgiving bad debt is simply an accounting correction.

Remediation

This issue has been acknowledged by Suilend, and a fix was implemented in commit `e16f24e4` ↗.

3.2. Rate limiter can be abused

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

Description

The protocol uses a single rate limiter for the entire protocol. The rate limiter limits the amount of withdrawn coins in a given time window. This includes all supported coin types but not cTokens. There is no fee for depositing and immediately withdrawing.

Impact

It is possible to halt withdrawals of other users by intentionally filling the rate limit by depositing and immediately withdrawing repeatedly. The attack can potentially be made less costly through flash loans. The exact cost of the attack would depend on the gas cost at the time of the attack and the cheapest available flash loans.

However, we note that the administrator can manually reset the rate limit if it is filled and potentially even completely disable it by setting the rate limit to a very large value.

Recommendations

We recommend that the rate-limiter mechanism be redesigned.

Remediation

This issue has been acknowledged by Suilend.

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. Centralization

The protocol is centralized in design; `LendingMarketCap` has absolute authority over the lending market. It is able to perform many actions to the detriment of the users, including the following:

- Reset the rate limit at any time
- Set the APR of a coin to a very large value, force-liquidating values through the added interest
- Set the borrow fee to a very high value, making it very uneconomical to borrow
- Set the spread fee to a large value, stealing all the profits from interest
- Create a reserve with a controlled coin type, allowing the admin to mint coins to use as collateral and drain the protocol
- Set the liquidation threshold to be very low, instantly liquidating users

This list is not exhaustive. Thus, the security and access control of the `LendingMarketCap` is paramount to the security of the protocol.

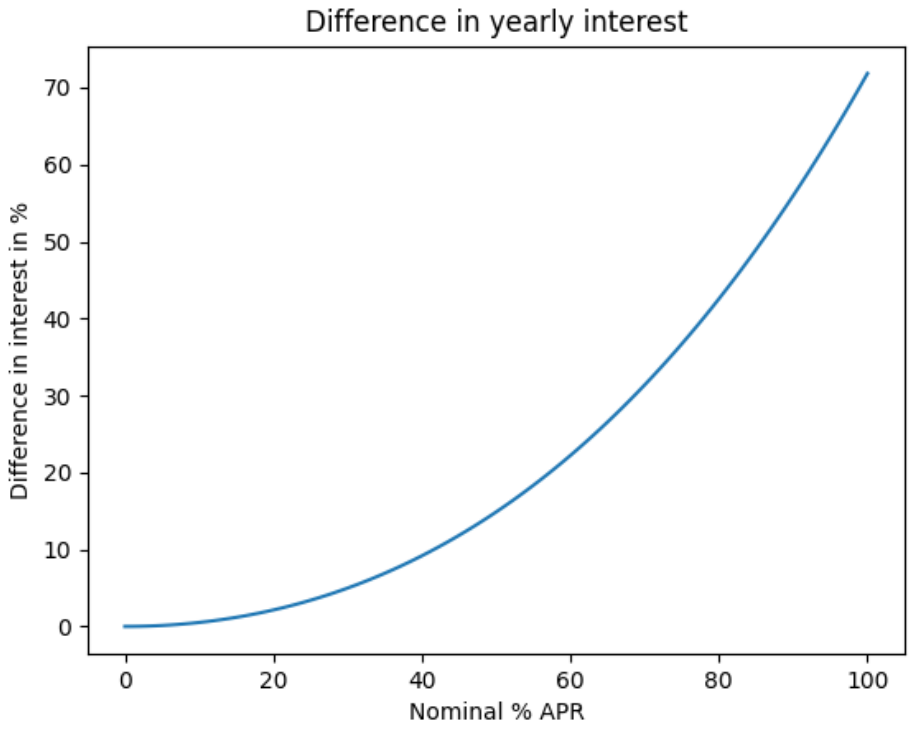
4.2. Use of APR in configuration

The following formula is used for the per-second APR.

$$p = 1 + \frac{A}{365 * 24 * 60 * 60}$$

The per-second interest rate is calculated by dividing the yearly APR by the number of seconds in the year and adding one. The formula for an exact per-second interest rate from an annual interest rate uses a k-th root where *k* is the number of seconds in a year. This can lead to discrepancy between APR and the effective interest paid (i.e. the APY).

In the below graph, we have graphed the discrepancy over a year in the exact interest rate and APR. For clarification, the difference is the extra interest paid using the APR formula over an exact interest rate formula.



This doesn't cause any issues, as it is mainly how the APR is represented and will be used in the protocol. We would recommend the suilend team to display both the APR (and expected APY calculated based on that) in the frontend so users are aware of this difference.

4.3. Liquidate maximum percentage is weighted

In the below liquidation code, there is a limit to the percent of an underwater user's debts liquidated in a single call to liquidate. The hardcoded constant percent is 20%.

```
public(friend) fun liquidate<P>(
    // [..]
    // we can liquidate up to 20% of the obligation's market value
    let max_repay_value = min(
        mul(
            obligation.weighted_borrowed_value_usd,
```

```
            decimal::from_percent(CLOSE_FACTOR_PCT)
        ),
        borrow.market_value
    );
```

However, as can be seen from the code above, the actual amount liquidatable in a single call is actually based on the weighted value of the debt. This is merely unusual and not a defect of the protocol.

4.4. Desync between reserve and obligation interest

In the protocol interest is tracked in two places. The per-pool interest calculations are performed in `reserve`. Per-user accounting happens in `obligation`.

The interest for `obligation` is calculated every time `compound_debt()` is called. The function is shown below.

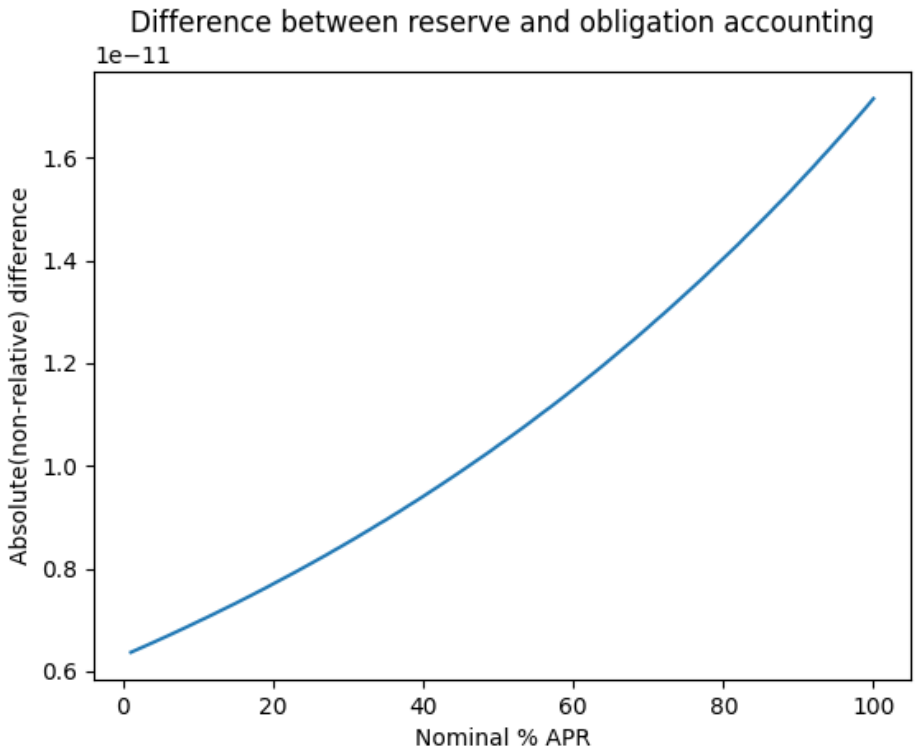
```
/// Compound the debt on a borrow object
fun compound_debt<P>(borrow: &mut Borrow, reserve: &Reserve<P>) {
    let new_cumulative_borrow_rate = reserve::cumulative_borrow_rate(reserve);

    let compounded_interest_rate = div(
        new_cumulative_borrow_rate,
        borrow.cumulative_borrow_rate
    );

    borrow.borrowed_amount = mul(
        borrow.borrowed_amount,
        compounded_interest_rate
    );

    borrow.cumulative_borrow_rate = new_cumulative_borrow_rate;
}
```

The function saves the cumulative interest rate to calculate the amount of interest that has accrued since the last time `compound_debt()` was called. The cumulative interest rate is updated every time `compound_interest()` is called in `reserve`.

The functions are called with different frequencies, and in most cases, `compound_interest()` in `reserve` will be called more often. Depending on the length of time between calls, the rounding error will be different. Since `compound_interest()` is called more often, the accounting error is in favor of the protocol. The user ends up paying slightly more interest than is accounted for in the `reserve`.

Assuming that every five seconds `compound_interest()` is called, and that `compound_debt()` is

called yearly, the difference in the interest rates is graphed below. Note that this is an extreme case, and that the scale of the y-axis is very small.



5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the modules 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: `lending_market`

Basic function

The `lending_market` does no accounting on its own but rather calls into `reserve` for pool-level accounting and into `obligation` for user-level accounting. The `lending_market` is meant to be the entry point to the protocol for users. All public functions are on this object. The public functions also perform checks on the arguments, such as ensuring the type arguments match the arguments. The rate limiter is also enforced here.

The `lending_market` object is intended to be created once and have a `reserve` added by the administrator for each supported coin. The `lending_market` creation function takes a one type witness as the type argument to ensure it is only created once per deployment of the creating contract. All functions that call into the `lending_market` take that type argument to ensure that the correct `lending_market` is called. The type argument is used to distinguish objects that belong to this particular `lending_market` from any others that may be created. For example, it is used to ensure the `LendingMarketCap` used for admin operations belongs to this `LendingMarket` and not one created by a potential attacker. This makes it so that the `lending_market_id` field no longer needs to be checked.

Flow: Earning interest

To earn interest on deposits, the user must have coins to deposit into the `LendingMarket` that matches the type of one of the `reserve`s. It is not necessary to have an `obligation` to earn interest. It is sufficient to simply hold cTokens.

cTokens effectively represent shares in the pool of a particular coin. While the cToken is held, any increases in the total value held by the pool is reflected in the cToken if it is redeemed.

To obtain cTokens, the user needs to call `deposit_liquidity_and_mint_ctokens` with coins of a type that corresponds to one of the reserves. To withdraw the interest, it is necessary to redeem the cTokens for the original coin type by calling `redeem_ctokens_and_withdraw_liquidity`. There is a risk that the pool is lent out to such an extent that it is no longer possible to redeem the cTokens.

Flow: Borrowing coins

In order to borrow coins, the user must first obtain cTokens, described in the earning interest flow. Then the user must create an `obligation` and deposit the cTokens into the obligation through the

functions `create_obligation` and `deposit_ctokens_into_obligation`. It is possible to use any type of coin as collateral for borrowing any other type of coin. All loans are required to be overcollateralized. Some coins are weighted more than others in terms of borrow capacity consumed for borrowing that coin. This weight is configurable by the administrator.

Once the coins are deposited into an `obligation`, the `obligation`'s capability object can be used by the user to borrow coins using the function `borrow`. The user accepts a debt larger than the amount of coins lent out; there is a fee for borrowing, which is paid immediately with the borrowed funds.

The rate limiter

The rate limiter is called from the `LendingMarket` to limit the total rate of outflow of (real) funds. cTokens are not limited by the rate limiter. The limit is configurable by the administrator.

Rate limiting works by only allowing a certain amount of funds to leave during a time period. The time periods are discrete and fixed windows (and not a moving window). The amount of funds allowed out in a given period is based on a fixed limit and the amount of funds expected to leave the protocol in the remainder of the window. The expected amount of funds leaving is based on the amount of funds leaving during the previous period. An assumption is made that in the remainder of the current window, the average amount of funds leaving per unit time matches the average of the previous time period.

5.2. Component: Reserve

Basic function

The reserve accounts for a specific asset owned by the protocol. It tracks the amount borrowed, the amount deposited and not borrowed, and two types of fees. The reserve object also tracks information for the oracle price source and issues cTokens to represent shares of ownership that allow holders to earn interest in the underlying coin.

Function `update_reserve_config`

There is a function that updates the configuration parameters of the reserve. There are no capability checks in this function, but it is marked as a friend function and is called from functions that do perform capability checks. The reserve config can be updated here without further checks.

Function `update_price`

This function is intended to be called from the lending market to refresh the price. The function checks that the `price_info` object being passed in is of the right type, so it is not possible to pass in the price info object of another asset to manipulate the price. The function updates the saved prices and the last updated timestamp on the reserve object.

Function `compound_interest`

This function calculates the compound interest at the current time. This works by maintaining an internal cumulative interest rate tracker. The function converts the interest into the interest rate by the second, then compounds it based on the number of seconds that has passed since the last update to the cumulative interest rate. The change in the cumulative interest rate is then applied to the assets in the reserve. This function also calculates the amount of new debt, and a portion of that new debt is turned into spread fees for the protocol based on the configured rate.

Function `claim_fees`

This function returns the spread fees and the borrow fees as token balances. There are no checks on this function on the caller, but it is a friend function that is only called from `lending_market`, where the caller is checked.

The borrow fees are always kept in a separate account on the `reserve` object, so the borrow fees are always completely withdrawn. On the other hand, the spread fees are kept in the pool and can even be lent out. Before any spread fees are withdrawn, the function calculates the maximum withdraw amount that does not put the available balance below the minimum value and withdraws up to that amount instead. If no funds can be withdrawn without breaching the minimum, then no funds are withdrawn from the spread fees.

Function `deposit_liquidity_and_mint_ctokens`

This function accepts coins of the same type as the reserve and mints cTokens proportional to the deposit, such that the newly minted cTokens represent a share of the pool exactly equal to the deposited coins.

There is a check that the maximum amount of coins in the pool is not exceeded. This check only applies for deposits and therefore does not impact withdrawals or repayments.

Function `redeem_ctokens`

This function takes some amount of cTokens and redeems for the coin type in the pool. Each cToken corresponds to a share of the pool, and a corresponding amount of coins are redeemed out.

This function will never allow tokens to be redeemed out if the amount left over is smaller than the minimum token amount. It is possible that so many tokens are lent out that there is not enough tokens physically in the pool to fulfill a redeem request at the moment, even if the pool is solvent.

Function `borrow_liquidity`

The function is called to borrow liquidity out of the `reserve`. This function only handles the removal of the coins from the `reserve` and not the accounting against the individual user. That is handled in `lending_market`.

The function deducts a borrow fee in cTokens, which is added to the special fee store of the current `reserve`. The fee is paid immediately and is added to the total debt, which is returned.

There is a check that the total amount borrowed in this `reserve` does not exceed a configuration limit and that the remaining amount in the `reserve` is above the hardcoded minimum.

Function `repay_liquidity`

The function adds the given coins into the balance of the current `reserve` and performs appropriate accounting.

Function `deposit_ctokens`

The function takes a deposit of cTokens for the purposes of using as collateral. The user continues to earn interest on the deposited tokens.

Function `withdraw_ctokens`

The function withdraws tokens that are previously used for collateral. In order to ensure the user does not withdraw collateral for an active loan, checks are performed in `lending_market`.

5.3. Component: Obligation

Basic function

The `obligation` object tracks the debts and collateral assets of a single user. It calculates compound interest on the debts and assets of this user and keeps track of how much the user can borrow based on the amount of assets the user owns.

Function `refresh`

Refreshes the calculated total deposit values and borrows lower and upper thresholds.

When borrowing from the protocol, users are required to be overcollateralized. However, there is some amount of grace where even if a user falls below the collateralization requirement for the initial borrow, the user is not considered eligible for liquidation. This leads to two separate thresholds for borrowing, the maximum amount of value the user can actively borrow and the maximum amount the user could have borrowed before being liquidated (this second threshold can be reached through price fluctuations of either the collateral or the borrowed assets). Both thresholds are tracked in `obligation` per user.

The `refresh` function adds up the total value of all deposited assets and adds to the aforementioned two thresholds a portion of the total deposited value. The effect on the borrow thresholds of each asset is calculated separately, since each asset has a different price.

For the debts of the current `obligation`, the current function compounds it and sums together the total value of the debts. This is done separately for each asset since each asset has a different price.

This function updates prices for all relevant `reserves` and reverts if the price is not fresh enough.

This function does not directly liquidate or check the thresholds are met. Instead, functions calling this base function perform these actions.

Function `deposit`

This function tracks the deposit of cTokens into the pool for the purpose of using as collateral for loans. Since depositing should not cause loans to less collateralized, there are no checks for borrow thresholds in this function.

There is a recalculation of the thresholds and summary statistics for the `obligation` in this function, but the recalculated values are not used for any security-sensitive purpose. The values can potentially be incorrect but will be recalculated before every sensitive operation anyway.

Function `borrow`

The function adds a borrowed amount into the accounting of the `obligation`. Then it updates the borrowed amount totals and checks that the debt does not exceed the maximum allowed amount. Since calls to borrow must be preceded by a `refresh` call, the updated amounts here are correct for the check.

Function `repay`

This function decreases the debt of the user by the passed-in amount. The total debt numbers are then updated based on potentially stale price data. The updated values, however, are not used on chain before they are updated again.

Function `liquidate`

This function is called in order to liquidate a user's assets. Liquidate in this context means repaying a portion of the user's debt in exchange for an equal value amount of the collateral, plus a bonus for the liquidator.

At most a fifth of the user's debts may be liquidated in a single call. Any excess value is returned to the user.

The maximum amount repaid is also bounded by the amount of assets available as collateral to give to the liquidator.

6. Assessment Results

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

During our assessment on the scoped Suilend modules, we discovered two findings, both of which were medium impact. Suilend acknowledged all findings and implemented fixes.

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.