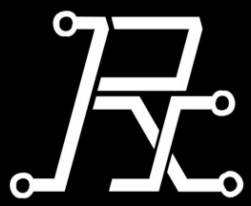

Security Assessment Report
Kamino Lending

July 03, 2023

Table of Content

Summary

From July 03, 2023 through July 23, 2023, *Kamino Finance* engaged *Ronny Xing* to review the security of its

- Kamino-Lending smart contract on Solana
    - commit hash **419de5ceb00e8b39ca0b0b6df756f754f7b4f9bb**

We conducted this assessment working with the code above and the security researcher mentioned in the title. The Kamino-Lending project is a non-custodial liquidity lending market to earn interest on supplying and borrowing assets.Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an overcollateralized (perpetually) or undercollateralized (one-block liquidity) fashion.

Throughout the engagement, we identified 10 issues, ranging in severity from low to critical, and 2 issues with undetermined severity.We discovered:

- Issue related to disabled price oracle manipulation;
- 3 issues related to the accounts verification about the obligation and farm program;
- Issue related to auto-deleverage liquidation;
- Issue related to the liquidation DOS due to elevation group update;
- Issue related to mixing assets from different elevation groups;
- Issue related to ownership update;
- 2 issues related to lack of refresh before config changes or state sync;
- Other informational and undetermined issues.

Disclaimer

This audit report should not be used as investment advice.

We make best effort at finding security issues in smart contracts, however we do not provide any guarantees on eliminating all possible security issues. As a single audit-based assessment cannot and should not be considered comprehensive, we always recommend proceeding with several other independent audits and a public bug bounty program to ensure the security of smart contracts.

Audit Scope

The assessment scope contains mainly the smart contracts of the kamino-lending program for the Kamino-Lending project committed by July 02, 2023 (**419de5ceb00e8b39ca0b0b6df756f754f7b4f9bb**). We listed the files we have audited below.

Table 1 Smart Contract Audit Scope

Contract path
programs/kamino-lending/src/state/liquidation_operations.rs
programs/kamino-lending/src/state/token_info.rs
programs/kamino-lending/src/state/mod.rs
programs/kamino-lending/src/state/lending_market.rs
programs/kamino-lending/src/state/last_update.rs
programs/kamino-lending/src/state/reserve.rs
programs/kamino-lending/src/state/nested_accounts.rs
programs/kamino-lending/src/state/types.rs
programs/kamino-lending/src/state/obligation.rs
programs/kamino-lending/src/handlers/handler_init_farms_for_reserve.rs
programs/kamino-lending/src/handlers/handler_refresh_obligation_farms_for_reserve.rs
programs/kamino-lending/src/handlers/handler_refresh_reserve.rs
programs/kamino-lending/src/handlers/handler_repay_obligation_liquidity.rs
programs/kamino-lending/src/handlers/handler_flash_borrow_reserve_liquidity.rs
programs/kamino-lending/src/handlers/handler_init_obligation.rs
programs/kamino-lending/src/handlers/handler_socialize_loss.rs
programs/kamino-lending/src/handlers/handler_deposit_reserve_liquidity_and_obligation_collateral.rs
programs/kamino-lending/src/handlers/handler_redeem_reserve_collateral.rs
programs/kamino-lending/src/handlers/handler_withdraw_obligation_collateral_and_redeem_reserve_collateral.rs
programs/kamino-lending/src/handlers/handler_deposit_reserve_liquidity.rs
programs/kamino-lending/src/handlers/handler_flash_repay_reserve_liquidity.rs
programs/kamino-lending/src/handlers/handler_request_elevation_group.rs
programs/kamino-lending/src/handlers/mod.rs
programs/kamino-lending/src/handlers/handler_liquidate_obligation_and_redeem_reserve_

collateral.rs
programs/kamino-lending/src/handlers/handler_withdraw_obligation_collateral.rs
programs/kamino-lending/src/handlers/handler_init_reserve.rs
programs/kamino-lending/src/handlers/handler_init_lending_market.rs
programs/kamino-lending/src/handlers/handler_deposit_obligation_collateral.rs
programs/kamino-lending/src/handlers/handler_redeem_fees.rs
programs/kamino-lending/src/handlers/handler_update_reserve_config.rs
programs/kamino-lending/src/handlers/handler_init_obligation_farms_for_reserve.rs
programs/kamino-lending/src/handlers/handler_refresh_obligation.rs
programs/kamino-lending/src/handlers/handler_borrow_obligation_liquidity.rs
programs/kamino-lending/src/handlers/handler_update_lending_market.rs
programs/kamino-lending/src/utils/token_transfer.rs
programs/kamino-lending/src/utils/consts.rs
programs/kamino-lending/src/utils/account_loader_trait.rs
programs/kamino-lending/src/utils/spltoken.rs
programs/kamino-lending/src/utils/slots.rs
programs/kamino-lending/src/utils/refresh_ix_utils.rs
programs/kamino-lending/src/utils/constraints.rs
programs/kamino-lending/src/utils/mod.rs
programs/kamino-lending/src/utils/macros.rs
programs/kamino-lending/src/utils/seeds.rs
programs/kamino-lending/src/utils/account_ops.rs
programs/kamino-lending/src/utils/prices/checks.rs
programs/kamino-lending/src/utils/prices/tests_price.rs
programs/kamino-lending/src/utils/prices/utils.rs
programs/kamino-lending/src/utils/prices/mod.rs
programs/kamino-lending/src/utils/prices/scope.rs
programs/kamino-lending/src/utils/prices/pyth.rs
programs/kamino-lending/src/utils/prices/types.rs
programs/kamino-lending/src/utils/prices/switchboard.rs
programs/kamino-lending/src/utils/borrow_rate_curve.rs
programs/kamino-lending/src/lib.rs

programs/kamino-lending/src/lending_market/tests_borrow_factor.rs
programs/kamino-lending/src/lending_market/tests_size.rs
programs/kamino-lending/src/lending_market/test_utils.rs
programs/kamino-lending/src/lending_market/withdrawal_cap_operations.rs
programs/kamino-lending/src/lending_market/ix_utils.rs
programs/kamino-lending/src/lending_market/mod.rs
programs/kamino-lending/src/lending_market/flash_ixs.rs
programs/kamino-lending/src/lending_market/tests_withdrawal_cap_operations.rs
programs/kamino-lending/src/lending_market/farms_ixs.rs
programs/kamino-lending/src/lending_market/lending_operations.rs
programs/kamino-lending/src/lending_market/tests_socialize_loss.rs
programs/kamino-lending/src/lending_market/lending_checks.rs

Vulnerability Dashboard

Vulnerability Summary

Table 3 Vulnerability Summary

Severity	# of Findings
Critical	1
High	3
Medium	3
Low	3
Informational	2
Undetermined	0
Total	12

Category Summary

Table 4 Category Summary

Category	# of Findings
Improper Input Validation	3
Insufficient Verification of Data Authenticity	2
Business Logic Errors	5
Insufficient Precision or Accuracy of a Real Number	1

Key Findings and Recommendations

Table 5 Key Audit Findings

ID	Title	Severity	Type
01	Improper validation for disabled oracles	**Critical**	Insufficient Verification of Data Authenticity
02	Lack of lending_market check for reserve farm initialization will lead to DOS	**High**	Improper Input Validation
03	Lack of check for obligation_farm in the refresh instruction	**High**	Improper Input Validation
04	Low activity assets can be auto-deleverage liquidated without cost of capital	**High**	Insufficient Precision or Accuracy of a Real Number & Business Logic Errors
05	Lack of ownership check for obligation farmer initialization	**Medium**	Improper Input Validation
06	Obligations with discarded elevation_group are unable to be liquidated	**Medium**	Business Logic Errors
07	Elevation_group can be used by assets outside the group for borrowing	**Medium**	Business Logic Errors
08	Validity of the new market owner should be verified	**Low**	Insufficient Verification of Data Authenticity
09	Interest should be updated before reserve config update	**Low**	Business Logic Errors
10	Obligation should be refreshed before refreshing obligation farmer	**Low**	Business Logic Errors

Improper validation for disabled oracles

Severity: **Critical**
Target: Smart Contract

Difficulty: **Low**
Type: CWE-345

Description

The instruction `refresh_reserve` can get the best market price from the three oracles. The accounts of the oracles are input as the code:

```
#[derive(Accounts)]
pub struct RefreshReserve<'info> {
    #[account(mut,
        constraint = reserve.load()?.config.token_info.is_valid() @
LendingError::InvalidOracleConfig,
        constraint =
reserve.load()?.config.token_info.is_twap_config_valid() @
LendingError::InvalidTwapConfig,
        constraint =
check_pyth_acc_matches(&reserve.load()?.config.token_info, &pyth_oracle) @
LendingError::InvalidPythPriceAccount,
        ...
    )]
    pub reserve: AccountLoader<'info, Reserve>,

    /// CHECK: Verified above
    pub pyth_oracle: Option<AccountInfo<'info>>,
```

And the validation of the oracle account info is checked in the function like `check_pyth_acc_matches`:

```
fn check_pyth_acc_matches(token_info: &TokenInfo, pyth_info:
&Option<AccountInfo>) -> bool {
    // Check need to be done if pyth is enabled in the configuration
    !token_info.pyth_configuration.is_enabled()
    // If enabled, the pyth account should be present and the key should
match the one in the configuration
        || matches!(pyth_info, Some(a) if *a.key ==
token_info.pyth_configuration.price)
}
```

But if the `pyth_configuration` is not enabled, the first condition will be true. In this case, the check function will return true regardless of the input pubkey. But in the `utils::prices::get_price` function, the `pyth_price` is wrapped as an `Option` from `{input pyth account}.and_then()` to check if it's None.

```
let pyth_price = pyth_price_account_info.and_then(|a|
get_pyth_price_and_twap(a).ok());
```

So when any oracle is disabled, the attacker can use a fake price account to set reserves with an untrue market price.

Proof of Concept

POC test file:
https://github.com/5z1punch/Kamino-Lending-POC/blob/main/programs/kamino-lending/tests/tests_rx_refresh_price_from_fake_oracle.rs

In the poc, the pyth oracle is disabled. And the attacker sets up a fake pyth oracle account to refresh the reserve as the untrue price.

Impact

The attacker can manipulate the collateral prices and borrow more value than normal debt with a real collateral rate close to zero. The entire liquidity of the reserve will be taken away.

Recommendation

If an oracle is disabled, the input account must be `Pubkey::default()` or `None`.

Lack of lending_market check for reserve farm initialization will lead to DOS

Severity: **High**
Target: Smart Contract

Difficulty: **Low**
Type: CWE-20

Description

In the instruction `init_farms_for_reserve`, source code `programs/kamino-lending/src/handlers/handler_init_farms_for_reserve.rs#L31-L46`:

```
#[derive(Accounts)]
pub struct InitFarmsForReserve<'info> {
    #[account(mut)]
    pub lending_market_owner: Signer<'info>,
    #[account(has_one = lending_market_owner)]
    pub lending_market: AccountLoader<'info, LendingMarket>,
    /// CHECK: Checked through create_program_address
    #[account(
        mut,
        seeds = [seeds::LENDING_MARKET_AUTH,
lending_market.key().as_ref()],
        bump = lending_market.load()?.bump_seed as u8,
    )]
    pub lending_market_authority: AccountInfo<'info>,

    #[account(mut)]
    pub reserve: AccountLoader<'info, Reserve>,
```

Although there is a signer check of the lending_market_owner for the lending_market, the account constraint of reserve doesn't check if the reserve belongs to the input lending_market. So the attacker can init a secondary market owned by himself by `init_lending_market` instruction. Then the attacker can use it to init or update a farm for any reserve from the primary market.

Proof of Concept

POC test file:
https://github.com/5z1punch/Kamino-Lending-POC/blob/main/programs/kamino-lending/tests/tests_rx_hijack_reserve_farms.rs
In the poc:

1. The attacker creates a new market account(fake market) by the `init_lending_market` instruction;
2. The attacker injects a farm from the fake market to the primary market;
3. The normal users try to deposit collateral into the primary market, but the transaction throws an `AuthorityFarmDelegateMissmatch` error.

Impact

Replacing the reserve farm with a new one which has a different `farm_delegate` will make the `refresh_obligation_farms_for_reserve` instruction DOS. The `farm_delegate` is a PDA account who, as a program, can manage the farm. It's from `lending_market_authority`:

```
let accounts = farms::accounts::InitializeFarmDelegated {
    farm_admin: ctx.accounts.lending_market_owner.to_account_info().key(),
    farm_state: farm_state_key,
    farm_delegate: ctx
        .accounts
        .lending_market_authority
        .to_account_info()
        .key(),
```

So if the attacker replaces the reserve farm, the `farm_delegate` will be different from the primary authority.

But as a normal user, it always uses the accounts and authority from the primary market to interact with the program. And all instructions that can change the state must follow the refresh instructions, which is checked in the `check_refresh_ixs!`. And once the reserve farm is not empty, the `refresh_obligation_farms_for_reserve` instruction must be called.

```
if reserve.get_farm(farm_type) != Pubkey::default() {
    let required_ix = RequiredIx {
        kind: RequiredIxType::RefreshFarmsForObligationForReserve,
        accounts: vec![(*reserve_address, 3), (*obligation_address, 1)],
    };
    required_pre_ixs.push(required_ix.clone());
    required_post_ixs.push(required_ix);
}
```

Because of the mismatch between the authority, the refresh will fail in the farm instruction `set_stake_delegated`.

The whole market will DOS for the normal users.

Recommendation

Add constraints

```
    #[account(mut,
        has_one = lending_market
    )]
    pub reserve: AccountLoader<'info, Reserve>,
```

to the accounts.

Lack of check for obligation_farm in the refresh instruction

Severity: **High**
Target: Smart Contract

Difficulty: **Low**
Type: CWE-20

Description

In the instruction `refresh_obligation_farms_for_reserve`, source code `programs/kamino-lending/src/handlers/handler_refresh_obligation_farms_for_reserve.rs#L69-L96`:

```
#[derive(Accounts)]
pub struct RefreshObligationFarmsForReserve<'info> {
    #[account(mut)]
    pub crank: Signer<'info>,
    /// CHECK: Obligation is checked against the lending market in lending_checks
    #[account(mut)]
    pub obligation: AccountInfo<'info>,

    ...

    /// CHECK: We initialize this in the farms program
    #[account(mut)]
    pub obligation_farm: AccountInfo<'info>,
```

There is not a check for the derived relation between the `obligation` and the `obligation_farm`. So the attacker can init an obligation and farm user with zero deposit, and then swap the farm user state with every normal user by calling the instruction `refresh_obligation_farms_for_reserve` with uncorrelated obligations and obligation_farms.

Proof of Concept

POC test file:
https://github.com/5z1punch/Kamino-Lending-POC/blob/main/programs/kamino-lending/tests/tests_rx_horizontal_refresh_farms.rs

Impact

The poc test above shows an effective attack vector:

1. The attacker uses the collateral amount of the obligation from the normal user(victim) to refresh the farm stake amount of himself.
2. The attacker uses his own collateral amount, which is 0, to refresh the farm stake amount of the victim.
3. The attacker gets all the farm rewards. Because he is the only one who has the non zero stake amount in the farm.
4. The victims lose all the rewards until they refresh manually.

Recommendation

Check the owner of the `obligation_farm` is the owner of the `obligation`.

Low activity assets can be auto-deleverage liquidated without cost of capital

Severity: **High**
Target: Smart Contract

Difficulty: **Low**
Type: CWE-840 & CWE-1339

Description

There are two valid liquidation state that loan value exceeds the liquidation threshold, or the `debt_reserve` or `collateral_reserve` can be auto-deleveraged.

If a reserve is auto-deleverage-able, the auto-deleverage liquidation threshold decreases by 1 bps per hour. It means that the normal loans can be liquidated as the auto-deleverage state continues. And there is a config `reserve.config.deleveraging_margin_call_period_secs` to ensure a minimum duration about auto-deleverage state before the normal loans can be liquidated.

The deleveraging started slot is saved in `deposit_limit_crossed_slot` and `borrow_limit_crossed_slot` when the reserve is refreshed manually in the `refresh_reserve` instruction:

```
// programs/kamino-lending/src/handlers/handler_refresh_reserve.rs
lending_operations::refresh_reserve_limit_timestamps(reserve,
clock.slot)?;
// programs/kamino-lending/src/lending_market/lending_operations.rs
pub fn refresh_reserve_limit_timestamps(reserve: &mut Reserve, slot: Slot)
-> Result<()> {
    reserve.update_deposit_limit_crossed_slot(slot)?;
    reserve.update_borrow_limit_crossed_slot(slot)?;
    Ok(())
}
```

And the `crossed` flag (deleveraging flag) is set by the code:

```
let crossed = self.liquidity.total_borrow()? >
Decimal::from(self.config.borrow_limit);

pub fn total_borrow(&self) -> Result<Decimal> {
    Ok(Decimal::from_scaled_val(self.borrowed_amount_wads))
}
```

It uses the `Decimal` compare between `borrowed_amount_wads` and `borrow_limit`. But for the check about deposit/borrow limit in the deposit/borrow instructions, it uses `u64` compare and floor scaled decimal converted to `u64`, e.g. function `lending_operations::flash_borrow_reserve_liquidity`:

```
if Decimal::from(liquidity_amount)
    .add(decimal_borrowed_amount)
    .try_floor::<u64>()
    .unwrap()
    > reserve.config.borrow_limit
{
    return err!(LendingError::InvalidAmount);
}
```

So deposit/borrow instructions can use the accuracy error above to pass the limit check and set the `crossed` flag at the same time.

And combining the above context issues, the instructions, which can deposit/borrow to/from the reserve but don't require the reserve must have been refreshed in the same slot, can be used to manipulate the auto-deleverage duration without cost of capital.

There are 4 instructions that satisfy the above conditions:

- deposit_reserve_liquidity
- redeem_reserve_collateral
- flash_borrow_reserve_liquidity
- flash_repay_reserve_liquidity

We use the instructions about flash loans in the next POC section to show how the reserve is auto-deleverage liquidated without cost of capital.

Proof of Concept

The following attack aims at low activity assets, because it requires that the reserve is not refreshed by others before liquidation starts.

1. The SOL reserve config: LTV 70%, liquidation_threshold 73%
2. The borrower(victim) deposited some SOL as collateral and borrowed 70% SOL from the market. It's a kind of leverage and should never be liquidated due to price fluctuation.
3. The attacker sends a transaction with these ordered ixs:
    - flash_borrow_reserve_liquidity: The borrow liquidity amount:u64 is `Decimal::from(reserve_config.borrow_limit)`

`.sub(Decimal::from_scaled_val(sol_reserve.liquidity.borrowed_amount_wads)).try_ceil::<u64>().unwrap()`
    - refresh_reserve: the `borrow_limit_crossed_slot` is set to the current slot.
    - flash_borrow_reserve_liquidity: the only cost is the flash loan fee.
4. Wait 310 hours, the liquidation_threshold will be down to 69.9%
5. The attacker sends a transaction with these ordered ixs to liquidate the victim's obligation:
    - flash_borrow_reserve_liquidity: The borrow liquidity amount should be updated to cross the borrow limit again.
    - refresh_reserve
    - refresh_obligation
    - liquidate_obligation_and_redeem_reserve_collateral
    - flash_borrow_reserve_liquidity
6. The reserve is still auto-deleverage-able after the liquidation. Repeat the above steps(no need to wait for any slot) until all obligations are liquidated.

POC test file:
https://github.com/5z1punch/Kamino-Lending-POC/blob/main/programs/kamino-lending/tests/tests_rx_flashloan_autodeleverage.rs

Impact
Once the reserve can wait enough time before being refreshed, most of the normal obligations can be liquidated maliciously.

Recommendation
Ensure the reserve is refreshed before the ixs:

- deposit_reserve_liquidity
- flash_borrow_reserve_liquidity

Or, update the `crossed` flag after the ixs:

- redeem_reserve_collateral
- flash_repay_reserve_liquidity

And use `try_ceil::<u64>()` or `Decimal` instead of `try_floor::<u64>()` for the deposit/borrow limit check.

Lack of ownership check for obligation farmer initialization

Severity: Medium
Target: Smart Contract

Difficulty: Low
Type: CWE-20

Description

In the instruction `init_obligation_farms_for_reserve`, source code `programs/kamino-lending/src/handlers/handler_init_obligation_farms_for_reserve.rs#L27-L32`:

```
#[derive(Accounts)]
pub struct InitObligationFarmsForReserve<'info> {
    #[account(mut)]
    pub owner: Signer<'info>,

    #[account(mut)]
    pub obligation: AccountLoader<'info, Obligation>,
```

There is not an ownership check of the owner account for the obligation account. So anyone as a signer can init a farm user account for any obligation which does not have a farm user account yet. And the owner of the farmer user will be the attacker instead of the real owner of the obligation.

Proof of Concept

POC test file:
https://github.com/5z1punch/Kamino-Lending-POC/blob/main/programs/kamino-lending/tests/tests_rx_hijack_farmer.rs

Impact

The poc test above shows an effective attack vector:

1. A depositor/user(victim) initiates his obligation but hasn't initialized the farm user account for it.
2. The attacker initializes the farm user for the obligation of the victim. The owner of the farm user will be the attacker.
3. The victim stakes his collateral/debt on the farm.
4. Only the attacker can harvest the reward from the farm because he is the owner. The victim will lose all the rewards and he can't take back the ownership.

Recommendation

Add constraints

```
    #[account(mut,
        has_one = lending_market,
        has_one = owner
    )]
    pub obligation: AccountLoader<'info, Obligation>,
```

to the accounts.

Obligations with discarded elevation_group are unable to be liquidated

Severity: **Medium**
Target: Smart Contract

Difficulty: **Low**
Type: CWE-840

Description

There is a check about the `elevation_group` in the obligation refresh functions `refresh_obligation_deposits` and `refresh_obligation_borrows`:

```
if elevation_group != ELEVATION_GROUP_NONE
    && !deposit_reserve
        .config
        .elevation_groups
        .contains(&elevation_group)
{
    return err!(LendingError::InconsistentElevationGroup);
}
```

It verifies the elevation_group of the obligation is the same as the reserve config. But if the market owner removes an elevation group from a reserve which is held by the obligation, the obligation can't be refreshed until the owner of the obligation updates the elevation_group.

There is the permission check in the instruction `request_elevation_group`, so only the owner of the obligation can update the elevation_group:

```
pub owner: Signer<'info>,

#[account(mut,
    has_one = lending_market,
    has_one = owner
)]
pub obligation: AccountLoader<'info, Obligation>,
```

However when the obligation defaults and needs to be liquidated, the liquidator can't refresh the default obligation before calling the `liquidate_obligation_and_redeem_reserve_collateral` instruction. The refresh instruction must be called because there is a state check in the `utils::assert_obligation_liquidatable`:

```
if obligation.last_update.is_stale(slot)? {
    msg!("Obligation is stale and must be refreshed in the current slot");
    return err!(LendingError::ObligationStale);
}
```

Proof of Concept

POC test file:
https://github.com/5z1punch/Kamino-Lending-POC/blob/main/programs/kamino-lending/tests/tests_rx_elevation_change_liquidate.rs

In the poc:

1. There is an elevation_group 1 for stablecoins. And the borrower requests the elevation_group 1 for his obligation, which uses USDC as collateral.
2. The market owner removes the elevation_group 1 from the USDC reserve.
3. The borrower above can't be liquidated, because the `refresh_obligation` instruction will throw a `InconsistentElevationGroup` error and revert the whole transaction.

Impact

The default obligation can't be liquidated. The system may run a deficit.

Recommendation

If the elevation_group does not match the reserve config in the liquidation process, the elevation_group should be set to the `ELEVATION_GROUP_NONE`.

Elevation_group can be used by assets outside the group for borrowing

Severity: **Medium**
Target: Smart Contract

Difficulty: **Low**
Type: CWE-840

Description

The elevation group delimits a package of assets which have similar risks(usually less risky, more stable assets, e.g. a group of stablecoins). The obligation only has the collateral/debt assets in a same elevation group can set the elevation group id to raise the ceiling about LTV and liquidation threshold to increase utilization rate.

But in the instruction `request_elevation_group`, source code `programs/kamino-lending/src/handlers/handler_request_elevation_group.rs`, the obligation will not be marked as stale after updating the elevation_group.

So a borrower can deposit some high quality collateral and raise the elevation_group first. And then the borrower sets the elevation_group back to `NONE` and he can borrow the asset, which is out of the previous elevation group, as the LTV of the previous elevation group.

Proof of Concept

The actual attack flow is complicated, because there is a refresh ixs check at the beginning of the `handler_borrow_obligation_liquidity` instruction:

```
check_refresh_ixs!(ctx, borrow_reserve);
```

It ensures that there must be a `refresh_reserve` ix for `borrow_reserve` and a `refresh_obligation` ix before the borrow ix. But the check is based on the instruction introspection from `SysInstructions` and doesn't limit the stack depth for the current instruction. It means the borrow ix can be called as a CPI to bypass the refresh ixs check.

The attack process:

1. The reserves config:
    - USDC and SOL reserves LTV all are 70%
    - USDC is in elevation group 1 and the LTV of elevation group 1 is 90%
    - SOL is not in elevation group 1

2. The attacker deposits USDC as collateral and requests the obligation to elevation group 1.
3. The attacker sends a transaction with these ordered ixs:
    - refresh sol reserve
    - refresh the obligation
    - Two CPI calls from a program deployed by the attacker.
4. CPI calls:
    - set the obligation back to elevation group NONE
    - borrow SOL as elevation group 1, which means the sol loan will be lent as LTV 90% instead of 70%.

POC test file:
https://github.com/5z1punch/Kamino-Lending-POC/blob/main/programs/kamino-lending/tests/tests_rx_elevation_mixing.rs

Impact

It significantly increases the default risk in the lending market. The attacker can only improve capital utilization but can't profit from it directly.

Recommendation

Mark the obligation as stale after `request_elevation_group`.

Validity of the new market owner should be verified

Severity: **Low**
Target: Smart Contract

Difficulty: **High**
Type: CWE-345

Description

The `update_lending_market` instruction can update the lending_market_owner by the current owner.

```
UpdateLendingMarketMode::UpdateOwner => {
    let value: [u8; 32] = value[0..32].try_into().unwrap();
    let value = Pubkey::from(value);
    market.lending_market_owner = value;
    msg!("Value is {:?}", value);
}
```

The public key of the new owner is just from a bytes array from the input data. Lack of validity check for the owner update is dangerous which can lead to loss of admin permission.

Recommendation

The best practice is that the update instruction only caches the public key of the new owner, and another instruction, which can be only called by the new owner, will really update the owner field of the market to the caller.

Interest should be updated before reserve config update

Severity: **Low**
Target: Smart Contract

Difficulty: **High**
Type: CWE-840

Description

The reserve should be updated and accumulated before updating the `reserve.config.protocol_take_rate` and `reserve.borrow_rate_curve` config of the reverse.

```
    /// Compound current borrow rate over elapsed slots
    fn compound_interest(
        ...
    ) -> LendingResult<()> {
        let slot_interest_rate =
current_borrow_rate.try_div(SLOTS_PER_YEAR)?;
        let compounded_interest_rate = Rate::one()
            .try_add(slot_interest_rate)?
            .try_pow(slots_elapsed)?;
        self.cumulative_borrow_rate_wads =
            Decimal::from_scaled_val(self.cumulative_borrow_rate_wads)
                .try_mul(compounded_interest_rate)?
                .into();
        ...
        self.accumulated_protocol_fees_wads = net_new_debt
            .try_mul(take_rate)?
            .try_add(decimal_acc_protocol_fees_wads)?
            .to_scaled_val::<u128>()?;
```

The function above will take interest for protocol at every "write" operation. But there is not a refresh operation before the reserve config update. It results in taking more or less fees from the interest since the last refresh.

The system will also be affected when updating the `reserve.borrow_rate_curve`. Because it changes the interest accrued since the last refresh.

Recommendation

Call `lending_operations::refresh_reserve_interest` before update config.

Obligation should be refreshed before refreshing obligation farmer

Severity: **Low**
Target: Smart Contract

Difficulty: **Low**
Type: CWE-840

Description

In the instruction `refresh_obligation_farms_for_reserve`, The `ReserveFarmKind::Debt` kind of farm uses the `obligation_liquidity.borrowed_amount_wads` as the staked token amount to update the farm user state.

programs/kamino-lending/src/handlers/handler_refresh_obligation_farms_for_reserve.rs :

```
ReserveFarmKind::Debt => {

    let liquidity = obligation.find_liquidity_in_borrows(reserve_address);

    ...
Decimal::from_scaled_val(obligation_liquidity.borrowed_amount_wads);

    ...

}
```

The `obligation_liquidity.borrowed_amount_wads` is accrued and updated in the `ObligationLiquidity::accrue_interest` when the obligation is refreshed. But the `refresh_obligation_farms_for_reserve` instruction doesn't refresh the obligation before calling the `cpi_set_stake_delegated`. So the staked amount of the debt farm will be a little less than the real borrow amount.

Recommendation

refresh the obligation borrowed liquidity before updating the farm user.

Informational and Undetermined Issues Summary

Withdrawal interface for fee vault is not implemented

Severity: **Informational** Target: Smart Contract

All the owner fee and protocol fee will be collected to the `reserve.liquidity.fee_vault` address. Such as the `redeem_fees` instruction:

```
    #[account(mut, address = reserve.load()?.liquidity.fee_vault)]
    pub reserve_liquidity_fee_receiver: Box<Account<'info, TokenAccount>>,
```

But there is not an interface that the market owner can use to transfer these tokens from the fee vault. The tokens are stuck in the current program version.

Confidence interval is too wide

Severity: **Informational** Target: Smart Contract

The MIN_CONFIDENCE_PERCENTAGE is defined in the `programs/kamino-lending/src/utils/prices/mod.rs`. It's hard-coded as 2%:

```
/// validate price confidence - confidence/price ratio should be less than 2%
const MIN_CONFIDENCE_PERCENTAGE: u64 = 2u64;
/// Confidence factor is used to scale the confidence value to a value that can be compared to the price.
const CONFIDENCE_FACTOR: u64 = 100 / MIN_CONFIDENCE_PERCENTAGE;
```

And the function `validate_pyth_confidence` uses the `CONFIDENCE_FACTOR` to check if the `pyth_price.conf` is not greater than 2% of the `pyth_price.price`.

But according to the pyth document, the `pyth_price.conf` is the standard deviation. Laplace distribution contains ~95% of the probability mass within ~2.12 standard deviations (~3 times the scale parameter).

So if the max standard deviation is 2%, in the worst case with (5% * 5% = 0.25%) probability, the LTV pct of a debt will be (2 * 2% * 2.12 = 8.48%) below the true value. It means that one out of every 400 loans is already above the liquidation threshold at the time they are lent in the worst case.

Recommendation: According to the pyth document, if markets are behaving normally, then the confidence interval will be tight, typically much less than 1% of the price.

Fix Log

Table 6 Fix Log

ID	Title	Severity	Status
01	Improper validation for disabled oracles	**Critical**	**Confirmed and Fixed**
02	Lack of lending_market check for reserve farm initialization will lead to DOS	**High**	**Confirmed and Fixed**
03	Lack of check for obligation_farm in the refresh instruction	**High**	**Confirmed and Fixed**
04	Low activity assets can be auto-deleverage liquidated without cost of capital	**High**	**Confirmed and Fixed**
05	Lack of ownership check for obligation farmer initialization	**Medium**	**Confirmed and Fixed**
06	Obligations with discarded elevation_group are unable to be liquidated	**Medium**	**Confirmed and Fixed**
07	Elevation_group can be used by assets outside the group for borrowing	**Medium**	**Confirmed and Fixed**
08	Validity of the new market owner should be verified	**Low**	**Confirmed and Fixed**
09	Interest should be updated before reserve config update	**Low**	**Confirmed and Fixed**
10	Obligation should be refreshed before refreshing obligation farmer	**Low**	**Acknowledged**

Confidence interval too wide -- **Skipped**: Our experience with confidence interval is that 2% is a good compromise between availability and security. We want to reduce the risk of invalid price in periods of high market volatility, 2% is still small enough to protect us against too wrong price evaluation.

Appendix

Table i Severity Categories

Severity	Description
Undetermined	The extent of the risk can not be determined during this assessment
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Low	The risk is relatively small or is not a risk that the customer indicated as important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possibly legal implication for client
High	Affects large number of users, very bad for clients reputation or posses possible financial or legal risk
Critical	Affects large number of users, and is capped at 10% of economic damage

Table ii Difficulty Levels

Difficulty	Description
Undetermined	The difficulty of the exploit is undetermined during this assessment
Low	Commonly exploited, existing tools can be leveraged or can be easily scripted
Medium	Attacker must write a dedicated exploit, or need in depth knowledge of a complex system
High	The attacker must have privileged insider access or need to know extremely complicated technical details or must discover other issues to exploit this

Table iii Vulnerability Classifications

Type	CWE ID
Improper Input Validation	CWE-20
Insufficient Verification of Data Authenticity	CWE-345
Business Logic Errors	CWE-840
Insufficient Precision or Accuracy of a Real Number	CWE-1339

About

 Ronny Xing

 ronnyschiatto@gmail.com

 sh3ll.plus

Independent Security Researcher 

Top 5 of Code4rena 

Co-Founder  OFFSIDE Labs

I issue the report in my individual capacity