Security Assessment Report

DefiTuna

March 14, 2025

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the DefiTuna smart contracts.

The artifact of the audit was the source code of the following programs, excluding tests, in a private repository.

The initial audit focused on the following versions and revealed 11 issues or questions.

program	type	commit
DefiTuna	Solana	`7ced6e0b11d1bcd4126fdc2fd2592dc52f6868b7`

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
DEFITUNA		
[M-01] Potential DoS due to uncapped repay amount	Medium	Resolved
[L-01] Repay may fail in corner cases	Low	Resolved
[I-01] Unnecessary signer seeds in "UpdateFeesAndRewards" CPI	Info	Resolved
[I-02] Missing checks for "mint_a" and "mint_b" in "liquidate_position_orca"	Info	Resolved
[I-03] Incorrect protocol fee calculation in "collect_and_compound_fees_orca"	Info	Resolved
[I-04] Incorrect "fixed_x64_to_f64" implementation	Info	Resolved
[I-05] Unnecessary overflow handling in "get_liquidity_for_amount_a"	Info	Resolved
[I-06] Inaccurate swap equation	Info	Resolved
[I-07] Missing "auto_compound" check in "collect_and_compound_fees_orca"	Info	Resolved
[Q-01] Questions on several unused field management	Question	Resolved
[Q-02] Question on state checks in "collect_and_compound_fees_orca"	Question	Resolved

Findings in Detail

DEFITUNA

[M-01] Potential DoS due to uncapped repay amount

In the `repay_debt` instruction, the current implementation does not enforce any restrictions on the quantity of tokens a user may repay. Both the `TunaPosition::decrease_debt_a` and `decrease_debt_b` functions update loan shares and loan funds using `saturating_sub`, which allows the repaid token amount to potentially exceed the user's debt.

This oversight not only exposes users to potential losses but, in combination with issue L-01, could enable a malicious actor to prevent other users from making proper repayments. Although an attacker would have no incentive to carry out such an attack—given the substantial token expenditure required with no prospect of profit—addressing this issue is recommended to protect the users.

```
/* programs/tuna/src/instructions/repay_debt.rs */
118 | if tuna_position_ata_a.amount > 0 {
119 |     vault_a.accrue_interest(timestamp)?;
120 |     let (_, shares) = vault_a.repay(tuna_position_ata_a.amount, 0)?;
121 |     tuna_position.decrease_debt_a(shares);
122 | }
123 |
124 | if tuna_position_ata_b.amount > 0 {
125 |     vault_b.accrue_interest(timestamp)?;
126 |     let (_, shares) = vault_b.repay(tuna_position_ata_b.amount, 0)?;
127 |     tuna_position.decrease_debt_b(shares);
128 | }

/* programs/tuna/src/state/tuna_position.rs */
171 | pub fn decrease_debt_a(&mut self, shares: u64) {
172 |     if self.loan_shares_a > 0 {
173 |         // Don't throw an error on overflow, as loan_funds is only used to compute the interest a
↪  user owes.
174 |         // Precision errors are acceptable in this case.
175 |         match mul_div_64(shares, self.loan_funds_a, self.loan_shares_a, Rounding::Up) {
176 |             Ok(funds) => {
177 |                 self.loan_funds_a = self.loan_funds_a.saturating_sub(funds);
178 |             }
179 |             Err(_) => {
180 |                 self.loan_funds_a = 0;
181 |             }
182 |         };
183 |     }
```

```
184 |
185 |     self.loan_shares_a = self.loan_shares_a.saturating_sub(shares);
186 | }
```

Resolution

Fixed by commit `1b74d03`.

DEFITUNA

[L-01] Repay may fail in corner cases

In the protocol, debt is managed using a classic shares-based model. Each mint has a corresponding vault that maintains `borrowed_funds` and `borrowed_shares`, with the ratio between these two values determining the value of each share. Within the `repay` function, the protocol calculates the corresponding `funds` or `shares` amount based on user input, rounding in a manner that is favorable to the protocol.

However, the update to `borrowed_funds` is currently performed using `checked_sub`. Since rounding is biased in favor of the protocol, users may end up repaying a slightly higher value in `funds` compared to the value of their `shares`. In certain edge cases, this can cause `checked_sub` to underflow, leading to a transaction revert and preventing successful repayment. To mitigate this issue, it is recommended to replace `checked_sub` with `saturating_sub`.

```
/* programs/tuna/src/state/vault.rs */
189 | self.borrowed_funds = self.borrowed_funds.checked_sub(funds).ok_or(ErrorCode::MathUnderflow)?;
```

Resolution

Fixed by commit `0b94cd0`.

DEFITUNA

[I-01] Unnecessary signer seeds in "UpdateFeesAndRewards" CPI

In the `collect_fees` and `collect_reward` functions, the current implementation first invokes Orca's `UpdateFeesAndRewards` instruction before proceeding with subsequent operations. Additionally, the CPI includes the tuna position PDA as a signer.

```
/* programs/tuna/src/cpi/orca/amm_orca.rs */
182 | let ctx = CpiContext::new_with_signer(whirlpool_program.to_account_info(), accounts, seeds);
183 | cpi::update_fees_and_rewards(ctx)?;

/* programs/tuna/src/cpi/orca/amm_orca.rs */
213 | let cpi_ctx = CpiContext::new_with_signer(whirlpool_program.clone(), accounts, seeds);
214 | cpi::update_fees_and_rewards(cpi_ctx)?;
```

However, since Orca's `UpdateFeesAndRewards` instruction is publicly callable, the inclusion of the tuna position PDA as a signer is unnecessary.

```
/* programs/whirlpool/src/instructions/update_fees_and_rewards.rs */
008 | pub struct UpdateFeesAndRewards<'info> {
009 |     #[account(mut)]
010 |     pub whirlpool: Account<'info, Whirlpool>,
011 |
012 |     #[account(mut, has_one = whirlpool)]
013 |     pub position: Account<'info, Position>,
014 |
015 |     #[account(has_one = whirlpool)]
016 |     pub tick_array_lower: AccountLoader<'info, TickArray>,
017 |     #[account(has_one = whirlpool)]
018 |     pub tick_array_upper: AccountLoader<'info, TickArray>,
019 | }
```

Resolution

Fixed by commit `9cbc2fd`.

DEFITUNA

[I-02] Missing checks for "mint_a" and "mint_b" in "liquidate_position_orca"

In the `LiquidatePositionOrca` instruction, the program accepts `mint_a` and `mint_b` accounts without verifying that they correspond to `whirlpool.token_mint_a` and `whirlpool.token_mint_b`, respectively.

```
/* programs/tuna/src/instructions/orca/liquidate_position_orca.rs */
031 | #[account()]
032 | pub mint_a: Box<InterfaceAccount<'info, Mint>>,
033 |
034 | #[account()]
035 | pub mint_b: Box<InterfaceAccount<'info, Mint>>,
```

As a result, if incorrect `mint_a` and `mint_b` accounts are provided, erroneous decimal values may be used within the `check_oracle_price` function, leading to incorrect validation results.

```
/* programs/tuna/src/instructions/orca/liquidate_position_orca.rs */
184 | if let Err(err) = check_oracle_price(
185 |     &clock,
186 |     ctx.accounts.whirlpool.sqrt_price,
187 |     &vault_a.pyth_oracle_feed_id.to_bytes(),
188 |     &ctx.accounts.pyth_oracle_price_feed_a,
189 |     ctx.accounts.mint_a.decimals,
190 |     &vault_b.pyth_oracle_feed_id.to_bytes(),
191 |     &ctx.accounts.pyth_oracle_price_feed_b,
192 |     ctx.accounts.mint_b.decimals,
193 |     market.oracle_price_deviation_threshold,
194 | ) {
195 |     if err == ErrorCode::OracleStalePrice {
196 |         // It's not fine, but we can't disable liquidations totally if the price feed account is
↪  not updated.
197 |         msg!("Ignoring oracle price as it's stale!")
198 |     } else {
199 |         return Err(err.into());
200 |     }
201 | }

/* programs/tuna/src/utils/pyth.rs */
030 | let oracle_price = (price_a as f64) / (price_b as f64) * 10_f64.powi(price_a_exp - price_b_exp +
↪  decimals_b as i32 - decimals_a as i32);
```

Resolution

Fixed by commit `bb8cb19`.

DEFITUNA

[I-03] Incorrect protocol fee calculation in "collect_and_compound_fees_orca"

In the `collect_and_compound_fees_orca` instruction, the collected fees are permitted to be reinvested for liquidity provisioning. According to the fee structure defined in the market, this portion of funds should incur fees. However, since these funds are more appropriately classified as collateral rather than debt, they should be subject to the `protocol_fee_on_collateral` rate rather than the `protocol_fee` rate.

```
/* programs/tuna/src/instructions/orca/collect_and_compound_fees_orca.rs */
154 | let fee_a = mul_div_64(collected_yield_a, ctx.accounts.market.protocol_fee as u64, HUNDRED_PERCENT
↪  as u64, Rounding::Down)?;
155 | let fee_b = mul_div_64(collected_yield_b, ctx.accounts.market.protocol_fee as u64, HUNDRED_PERCENT
↪  as u64, Rounding::Down)?;
156 | msg!("Protocol fees: [{}; {}]", fee_a, fee_b);
```

Resolution

Fixed by commit `9f73f79`.

DEFITUNA

[I-04] Incorrect "fixed_x64_to_f64" implementation

In the Orca pool, the square root price is stored using an x64 format. To facilitate certain computations, a conversion function `fixed_x64_to_f64` was implemented. However, the current implementation mistakenly uses `FRAC_MASK` as the denominator, whereas the correct denominator should be `Q64` (i.e., `FRAC_MASK + 1`). Despite this discrepancy, the resulting error is minimal, and given that this function inherently leads to precision loss in the lower bits, the overall impact of this issue is negligible.

```
/* programs/tuna/src/math/fixed.rs */
011 | pub fn fixed_x64_to_f64(value: u128) -> f64 {
012 |     const FRAC_MASK: u128 = u64::MAX as u128;
013 |     (value >> 64) as f64 + ((value & FRAC_MASK) as f64) / (FRAC_MASK as f64)
014 | }
```

Resolution

Fixed by commit `336a81b`.

DEFITUNA

[I-05] Unnecessary overflow handling in "get_liquidity_for_amount_a"

The implementation of `get_liquidity_for_amount_a` anticipates potential overflow during computation. Specifically, if the product of `intermediate` and `wide_amount` exceeds the U256 range, the current approach divides `wide_amount` by `delta_sqrt_price` before multiplying by `intermediate`.

However, since `amount` is defined as a u64 and `intermediate` is computed by multiplying two u128 `sqrt_price` values followed by a right shift of 64 bits, their multiplication will not exceed the U256 range. Furthermore, if overflow handling were indeed required, the larger number should be used as the dividend; otherwise, dividing the u64 `amount` by `delta_sqrt_price` could yield zero in such scenarios.

```
/* programs/tuna/src/math/orca/liquidity.rs */
053 | fn get_liquidity_for_amount_a(amount: u64, sqrt_price_lower: u128, sqrt_price_upper: u128) ->
↪  Result<u128> {
054 |     let wide_amount = U256::from(amount);
055 |     let intermediate = U256::from(sqrt_price_upper).mul(U256::from(sqrt_price_lower)).shr(64);
056 |     let delta_sqrt_price = U256::from(sqrt_price_upper - sqrt_price_lower);
057 |
058 |     let liquidity: U256 = match intermediate.checked_mul(wide_amount) {
059 |         // If the previous equation overflows, try another one that does a division first
060 |         None => wide_amount
061 |             .div(delta_sqrt_price)
062 |             .checked_mul(U256::from(intermediate))
063 |             .ok_or(ErrorCode::MathOverflow)?,
064 |         Some(r) => r.div(delta_sqrt_price),
065 |     };
```

Resolution

Fixed by commit `070c406`.

DEFITUNA

[I-06] Inaccurate swap equation

To enhance user convenience, the current design permits users to supply liquidity using any arbitrary ratio of token A and token B. Subsequently, the protocol invokes Orca's swap function to adjust the quantities of both tokens based on the Orca pool's square root price, thereby achieving the appropriate ratio for liquidity provision. However, because the swap itself affects the pool's square root price, the computation becomes considerably more complex. The current implementation employs the bisection method to approximate the root of the following equation, determining the expected post-swap square root price, which is then used to calculate the necessary token swap amount.

However, in Orca's swap implementation, the fee is always collected in the form of the input token. That is, if the swap direction is from token A to token B, the fee is charged in token A, and vice versa. This behavior is not properly reflected in the equation.

```
/* programs/tuna/src/manager/swap.rs */
087 | /// # Arguments
088 | ///
089 | /// * `p` - sqrt price after a swap
090 | /// * `p0` - current pool sqrt price
091 | /// * `pl` - lower sqrt price
092 | /// * `pu` - upper sqrt price
093 | /// * `x` - amount of x tokens
094 | /// * `y` - amount of y tokens
095 | /// * `liquidity` - current liquidity
096 | /// * `f` - (1.0 - swap_fee)
097 | ///
098 | /// # Description
099 | ///
100 | /// Deposit ratio = x/y = (√Pu - √P) / (√P   √Pu   (√P - √Pl))
101 | ///
102 | /// Deposit ratio = x/y = (x + L (1/√P0 - 1/√P)) / (y + L (√P0 - √P) (1-fee))
103 | ///
104 | /// Using above we can write
105 | ///
106 | /// => (x + L (1/√P0 - 1/√P)) / (y + L (√P0 - √P) (1-fee)) = (√Pu - √P) / (√P   √Pu   (√P - √Pl))
107 | ///
108 | /// => (x + L/√P0 - L/√P)   (√P   √Pu   (√P - √Pl)) - (y + L (√P0 - √P) (1-fee))   (√Pu - √P) = 0
109 | fn swap_equation(price: f64, current_price: f64, lower_price: f64, upper_price: f64, x: f64, y:
↪  f64, liquidity: f64, one_minus_fee: f64) -> f64 {
110 |     (x + liquidity / current_price - liquidity / price) * (price * upper_price * (price -
↪  lower_price))
111 |         - (y + liquidity * (current_price - price) * one_minus_fee) * (upper_price - price)
```

```
112 | }
```

Resolution

Fixed by commit `56df6d2`.

DEFITUNA

[I-07] Missing "auto_compound" check in "collect_and_compound_fees_orca"

In the `TunaPosition` account, there is an `auto_compound` boolean field that indicates whether a position should automatically compound its yield.

However, in the `collect_and_compound_fees_orca` instruction, this field is not being verified, allowing a liquidator to perform compounding on any position regardless of the user's auto-compounding setting. Although no third-party liquidators currently exist—and any necessary checks might be performed off-chain—it is recommended that this verification be implemented within the program.

```
/* programs/tuna/src/state/tuna_position.rs */
078 | /// Set to true for yield auto compounding.
079 | pub auto_compound: bool,

/* programs/tuna/src/instructions/orca/collect_and_compound_fees_orca.rs */
037 | #[account(
038 |     mut,
041 |     constraint = tuna_position.authority == authority.key() || tuna_config.liquidator_authority ==
↪  authority.key(),
043 | )]
044 | pub tuna_position: Box<Account<'info, TunaPosition>>,
```

Resolution

Fixed by commit `9f73f79`.

DEFITUNA

[Q-01] Questions on several unused field management

In the current implementation, several fields are maintained within the program but are not actively utilized—potentially because they are intended for use only by the frontend or other off-chain keeper programs. We have the following questions regarding the management of these fields:

1. **TunaPosition::compounded_yield_a & TunaPosition::compounded_yield_b**
   According to the documentation, these fields represent the yield amount in token A/B that has been collected and compounded into the position. In practice, these values are increased by the amount of collected yield minus the fee in the `collect_and_compound_fees_orca` instruction. Additionally, in the `remove_liquidity_orca` instruction, a proportional amount of the compounded yield—corresponding to the liquidity removed—is deducted from the total compounded yield.
   However, in the `liquidate_position_orca` instruction, when a position is completely closed, the yield available for collection is added to the compounded yield. Could you please clarify the rationale behind this approach? Furthermore, in scenarios where the position is not fully closed, should the compounded yield be reduced proportionally, similar to the behavior in the `remove_liquidity_orca` instruction?

```
/* programs/tuna/src/instructions/orca/collect_and_compound_fees_orca.rs */
158 | tuna_position.compounded_yield_a += collected_yield_a - fee_a;
159 | tuna_position.compounded_yield_b += collected_yield_b - fee_b;

/* programs/tuna/src/instructions/orca/remove_liquidity_orca.rs */
294 | // If the position is decreased, the compounded yield amount is also decreased proportionally.
295 | tuna_position.compounded_yield_a -= mul_div_64(tuna_position.compounded_yield_a,
↪  withdraw_percent as u64, HUNDRED_PERCENT as u64, Rounding::Down)?;
296 | tuna_position.compounded_yield_b -= mul_div_64(tuna_position.compounded_yield_b,
↪  withdraw_percent as u64, HUNDRED_PERCENT as u64, Rounding::Down)?;
297 |

/* programs/tuna/src/instructions/orca/liquidate_position_orca.rs */
247 | if full_position_close {
248 |     // Collect yield
249 |     collect_fees(
250 |         tuna_position,
251 |         ctx.accounts.whirlpool.to_account_info(),
```

```
252 |         orca_position.to_account_info(),
253 |         ctx.accounts.tuna_position_ata.to_account_info(),
254 |         ctx.remaining_accounts[POOL_VAULT_ATA_A_RA_INDEX].to_account_info(),
255 |         ctx.remaining_accounts[POOL_VAULT_ATA_B_RA_INDEX].to_account_info(),
256 |         tuna_position_ata_a.to_account_info(),
257 |         tuna_position_ata_b.to_account_info(),
258 |         ctx.accounts.token_program.to_account_info(),
259 |         ctx.accounts.whirlpool_program.to_account_info(),
260 |         ctx.remaining_accounts[TICK_ARRAY_LOWER_RA_INDEX].to_account_info(),
261 |         ctx.remaining_accounts[TICK_ARRAY_UPPER_RA_INDEX].to_account_info(),
262 |     )?;
263 |
264 |     tuna_position_ata_a.reload()?;
265 |     tuna_position_ata_b.reload()?;
266 |
267 |     let collected_yield_a = tuna_position_ata_a.amount - tuna_position_amount_before_a;
268 |     let collected_yield_b = tuna_position_ata_b.amount - tuna_position_amount_before_b;
269 |     tuna_position.compounded_yield_a += collected_yield_a;
270 |     tuna_position.compounded_yield_b += collected_yield_b;
271 |     msg!("Collected yield: [{}; {}]", collected_yield_a, collected_yield_b);
272 | }
```

2. **Market::liquidity_provider**

   The Market account contains a `liquidity_provider` field intended to indicate whether the current market is associated with Orca or, potentially in the future, Raydium. However, subsequent code that uses the Market account does not perform any checks on this field. Given that the Market account already includes a pool field, which is sufficient to prevent potential type confusion issues, is this omission of a check on `liquidity_provider` intentional?

Resolution

1. **TunaPosition::compounded_yield_a & TunaPosition::compounded_yield_b**

   Fixed by commit `71a4f55`.

2. **Market::liquidity_provider**

   The team clarified that this field is not used anywhere.

DEFITUNA

[Q-02] Question on state checks in "collect_and_compound_fees_orca"

In the `collect_and_compound_fees_orca` instruction, the only program/market state check currently implemented is for `tuna_config.suspend_remove_liquidity`.

However, since this instruction actually adds liquidity, should the check be switched to `tuna_config.suspend_add_liquidity` instead of `suspend_remove_liquidity`, and should an additional check for `market.disabled` also be incorporated?

```
/* programs/tuna/src/instructions/orca/collect_and_compound_fees_orca.rs */
117 | if ctx.accounts.tuna_config.suspend_remove_liquidity {
118 |     return Err(ErrorCode::Suspended.into());
119 | }
```

Resolution

Fixed by commit `095b765`.

Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and CrypticDot dba DefiTuna (the "Client''). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification. We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.