Security Assessment Report

Keel Solana PSM

September 24, 2025

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the Keel Solana PSM smart contract.

The artifact was the source code of the following programs, excluding tests, in https://github.com/keel-fi/solana-psm/.

The initial audit focused on the following versions and revealed 13 issues or questions.

program	type	commit
solana-psm	Solana	de3823a2d2f2ecbbe61eecdd4d7341c373f8ab96

The post-audit was conducted on version 35f9c63c1046c9c9677b482ccb0728e5182670c1, which concludes this audit.

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

Table of Contents

Result Overview

Issue	Impact	Status
SOLANA-PSM		
[M-01] Incomplete `transferFee` handling	Medium	Resolved
[M-02] Incorrect withdraw all token amount calculation	Medium	Resolved
[L-01] Ensure the mints of the pool and tokens are different	Low	Resolved
[L-02] Missing owner fee handling in single sided deposit and withdrawal	Low	Resolved
[L-03] Prevent burning pool initial supply	Low	Resolved
[L-04] Validate token2022 extensions in the initialization	Low	Resolved
[I-01] Inconsistent signer requirement in the comment and code	Info	Resolved
[I-02] Redundant overflow check	Info	Resolved
[I-03] The `ray` is not validated in `RedemptionRateCurve`	Info	Resolved
[I-04] Incorrect `source_token_mint_info.owner` parameter	Info	Resolved
[I-05] Duplicated owner fee in pool token amount calculation	Info	Resolved
[I-06] Potential fee overcharge during token swap	Info	Resolved
[I-07] Profitable sandwich attack on rate update transaction	Info	Acknowledged

Findings in Detail

SOLANA-PSM

[M-01] Incomplete `transferFee` handling

Identified in commit `de3823a`.

The PSM program supports creating pools with Token-2022 standard tokens. Some Token-2022 standard tokens may have extensions installed, which may impact the program's business logic.

For example, if a token has the `TransferFees` extension enabled, the actual amount received by the destination account will be less than the amount transferred from the source account.

Currently, only the `process_swap` has special handling for the `TransferFees` extension, while the remaining four instructions do not consider transfer fees.

```
/* program/src/processor.rs */
415 | pub fn process_swap(
491 | if let Ok(transfer_fee_config) = source_mint.get_extension::<TransferFeeConfig>() {
492 |     amount_in.saturating_sub(
493 |         transfer_fee_config
494 |             .calculate_epoch_fee(Clock::get()?.epoch, amount_in)
495 |             .ok_or(SwapError::FeeCalculationFailure)?,
496 |     )
497 | } else {
498 |     amount_in
499 | }
```

Impact

Failure to check and handle the `transferFee` extension may mess up the accounting.

For example, in `process_deposit_all_token_types`, the user is willing to pay up to `maximum_token_a_amount` and `maximum_token_b_amount` to obtain `pool_token_amount` liquidity tokens.

When the transfer fees extension is enabled, after the program transfers the user's `token_a` and `token_b` to the pool, the amounts received will be less, which means the pool is minting more pool tokens compared to the `token_a` and `token_b` received.

Potential repairs

All four instructions need to check whether the token has the `TransferFees` extension installed before the `Self::token_transfer` related logic, in order to calculate the correct transfer amount:

- `process_deposit_all_token_types`
- `process_withdraw_all_token_types`
- `process_deposit_single_token_type_exact_amount_in`
- `process_withdraw_single_token_type_exact_amount_out`

Taking `process_deposit_all_token_types` as an example.

Before performing the slippage check, calculate the actual required `token_a_amount_with_fee` and `token_b_amount_with_fee`.

```
let token_a_mint = Self::unpack_mint(token_a_mint_info, token_swap.token_program_id())?;
let token_b_mint = Self::unpack_mint(token_b_mint_info, token_swap.token_program_id())?;
if let Ok(transfer_fee_config) = token_a_mint.get_extension::<TransferFeeConfig>() {
    token_a_amount_with_fee = token_a_amount.saturating_add(
        transfer_fee_config
            .calculate_inverse_epoch_fee(Clock::get()?.epoch, token_a_amount)
            .ok_or(SwapError::FeeCalculationFailure)?,
    );
} else {
    token_a_amount_with_fee = token_a_amount;
}
if let Ok(transfer_fee_config) = token_b_mint.get_extension::<TransferFeeConfig>() {
    token_b_amount_with_fee = token_b_amount.saturating_add(
        transfer_fee_config
            .calculate_inverse_epoch_fee(Clock::get()?.epoch, token_b_amount)
            .ok_or(SwapError::FeeCalculationFailure)?,
    );
} else {
    token_b_amount_with_fee = token_b_amount;
}
```

In the slippage check, instead of checking the amount after fees:

```
/* program/src/processor.rs */
728 | if token_a_amount > maximum_token_a_amount {
735 | if token_b_amount > maximum_token_b_amount {
```

It should check the amounts transferred from the user:

```
if token_a_amount_with_fee > maximum_token_a_amount {
if token_b_amount_with_fee > maximum_token_b_amount {
```

When subsequently calling the `Self::token_transfer` function, the amount parameter needs to be set to `token_a_amount_with_fee` or `token_b_amount_with_fee`.

Resolution

Fixed by commit b28fe82.

SOLANA-PSM

[M-02] Incorrect withdraw all token amount calculation

Identified in commit `de3823a`.

The current token withdrawal amount calculation in the `redemption_rate` curve and the `constant_price` curve may silently introduce user losses if the slippage is not properly set.

The redemption_rate curve

In the `process_withdraw_all_token_types` function of `redemption_rate` curve, users can withdraw the underlying pool tokens by burning LP tokens.

```
/* program/src/processor.rs */
780 | pub fn process_withdraw_all_token_types(
786 | ) -> ProgramResult {
843 |     let results = calculator
844 |         .pool_tokens_to_trading_tokens(
845 |             pool_token_amount,
846 |             u128::from(pool_mint.supply),
847 |             u128::from(token_a.amount),
848 |             u128::from(token_b.amount),
849 |             RoundDirection::Floor,
850 |             token_swap.get_current_timestamp_opt()?
851 |         )
852 |         .ok_or(SwapError::ZeroTradingTokens)?;
```

Function `process_withdraw_all_token_types` calls function `pool_tokens_to_trading_tokens` to calculate the resulting amounts of token A and token B.

```
/* program/src/curve/redemption_rate.rs */
268 | fn pool_tokens_to_trading_tokens(
269 |     &self,
270 |     pool_tokens: u128,
271 |     pool_token_supply: u128,
272 |     swap_token_a_amount: u128,
273 |     swap_token_b_amount: u128,
274 |     round_direction: super::calculator::RoundDirection,
275 |     timestamp: Option<u128>
276 | ) -> Option<super::calculator::TradingTokenResult> {
284 |     let total_value = U256::from(self
285 |         .normalized_value(swap_token_a_amount, swap_token_b_amount, timestamp)?
286 |         .to_imprecise()?);
288 |     let (token_a_amount, token_b_amount) = match round_direction {
289 |         RoundDirection::Floor => {
291 |             let token_a_amount = pool_tokens
292 |                 .checked_mul(total_value)?
293 |                 .checked_div(pool_token_supply)?
```

```
294 |                 .min(U256::from(swap_token_a_amount));
296 |             let token_b_amount = pool_tokens
297 |                 .checked_mul(total_value)?
298 |                 .checked_mul(ray)?
299 |                 .checked_div(token_b_price)?
300 |                 .checked_div(pool_token_supply)?
301 |                 .min(U256::from(swap_token_b_amount));
303 |             (token_a_amount, token_b_amount)
304 |         }
```

In particular, parameters of `pool_tokens_to_trading_tokens` are:

- `pool_tokens`: The amount of LP tokens the user wants to burn
- `pool_token_supply`: The total supply of LP tokens in the pool
- `swap_token_a_amount`: The pool's reserve of token A
- `swap_token_b_amount`: The pool's reserve of token B
- `round_direction`: Set to Floor in withdraw_all_token_types
- `timestamp`: Used to calculate the price of token B

And, the resulting token A/B amounts are calculated as: `(pool_tokens * total_value / pool_token_supply).min(swap_token_a_amount)`

Here, `total_value` is obtained from the `normalized_value` function, which returns half of the total value of the pool's underlying tokens.

```
/* program/src/curve/redemption_rate.rs */
402 | fn normalized_value(
403 |     &self,
404 |     swap_token_a_amount: u128,
405 |     swap_token_b_amount: u128,
406 |     timestamp: Option<u128>
407 | ) -> Option<spl_math::precise_number::PreciseNumber> {
417 |     let value = if swap_token_b_value.saturating_sub(U256::from(u64::MAX))
418 |         > U256::MAX.saturating_sub(U256::from(u64::MAX))
419 |     {
420 |         swap_token_b_value
421 |             .checked_div(U256::from(2))?
422 |             .checked_add(U256::from(swap_token_a_amount).checked_div(U256::from(2))?)?
423 |     } else {
424 |         U256::from(swap_token_a_amount)
425 |             .checked_add(swap_token_b_value)?
426 |             .checked_div(U256::from(2))?
427 |     };
```

As a result, when the expected value `(pool_tokens * total_value / pool_token_supply)` exceeds `swap_token_a_amount`, the actual amount received will be limited by the `.min()` call. This can cause users to receive less than the value of the LP tokens they burned (if slippage protection is not properly set).

Consider the following example:

- The user burns 10 LP tokens using withdraw_all_token_types
- Total LP token supply: 50
- Pool reserves: Token A: 1000, Token B: 10 (price = 1)
- Slippage parameters `minimum_token_a_amount` and `minimum_token_b_amount` are set to `0`

According to the logic:

- Total normalized value will be `(1000 + 10 * 1) / 2 = 505`
- Token a withdrawal amount: `pool_tokens * total_value / pool_token_supply == 10 * 505 / 50 = 101` , `(101).min(1000) == 101`.
- Token b withdrawal amount: `pool_tokens * total_value / pool_token_supply == 10 * 505 / 50 = 101`, `(101).min(10) == 10`.
- The total value received by the user: `101 + 10 * 1 = 111`.
- The LP token value burned: `10 * (1000 + 10 * 1) / 50 = 202`.

This results in a significant loss to the user.

The constant_price curve

A similar issue also exists in the `constant_price` curve.

It is recommended to update the token withdrawal amount formula in the `withdraw_all_token_types`.

Resolution

Fixed by commit 5b1956b.

SOLANA-PSM

[L-01] Ensure the mints of the pool and tokens are different

Identified in commit `de3823a`.

The `process_initialize` implements several checks among `token_a.mint`, `token_b.mint` and `pool_mint` as follows. It ensures that the `token_a` and `token_b` do not share the same mint.

In addition, the default `validate_supply()` requires that `token_a.amount > 0` and `token_b.amount > 0`. Given `pool_mint.supply` must be `0` (line 326), they together ensure that the `pool_mint` and the mints of `token_a` and `token_b` cannot be the same.

```
/* program/src/processor.rs */
245 | pub fn process_initialize(
246 |     program_id: &Pubkey,
247 |     fees: Fees,
248 |     swap_curve: SwapCurve,
249 |     accounts: &[AccountInfo],
250 |     swap_constraints: &Option<SwapConstraints>,
251 | ) -> ProgramResult {
307 |     if token_a.mint == token_b.mint {
308 |         return Err(SwapError::RepeatedMint.into());
309 |     }
310 |     swap_curve
311 |         .calculator
312 |         .validate_supply(token_a.amount, token_b.amount)?;
326 |     if pool_mint.supply != 0 {
327 |         return Err(SwapError::InvalidSupply.into());
328 |     }

/* program/src/curve/calculator.rs */
088 | pub trait CurveCalculator: Debug + DynPack {
164 |     fn validate_supply(&self, token_a_amount: u64, token_b_amount: u64) -> Result<(), SwapError> {
165 |         if token_a_amount == 0 {
166 |             return Err(SwapError::EmptySupply);
167 |         }
168 |         if token_b_amount == 0 {
169 |             return Err(SwapError::EmptySupply);
170 |         }
171 |         Ok(())
172 |     }
```

However, for curves that implement the `CurveCalculator`, they do not have the same checks:

1. ConstantPriceCurve

```
/* program/src/curve/constant_price.rs */
067 | impl CurveCalculator for ConstantPriceCurve {
203 |     fn validate_supply(&self, token_a_amount: u64, _token_b_amount: u64) -> Result<(), SwapError> {
204 |         if token_a_amount == 0 {
205 |             return Err(SwapError::EmptySupply);
```

```
206 |         }
207 |         Ok(())
208 |     }
```

2. OffsetCurve

```
/* program/src/curve/offset.rs */
035 | impl CurveCalculator for OffsetCurve {
137 |     fn validate_supply(&self, token_a_amount: u64, _token_b_amount: u64) -> Result<(), SwapError> {
138 |         if token_a_amount == 0 {
139 |             return Err(SwapError::EmptySupply);
140 |         }
141 |         Ok(())
142 |     }
```

3. RedemptionRateCurve

```
/* program/src/curve/redemption_rate.rs */
226 | impl CurveCalculator for RedemptionRateCurve {
391 |     fn validate_supply(
392 |         &self,
393 |         token_a_amount: u64,
394 |         _token_b_amount: u64
395 |     ) -> Result<(), SwapError> {
396 |         if token_a_amount == 0 {
397 |             return Err(SwapError::EmptySupply);
398 |         }
399 |         Ok(())
400 |     }
```

Therefore, it's possible that `token_b.mint` may be the same as the `pool_mint` .

A constraint should be added in `process_initialize` requiring that the mint of `token_a` and `token_b` cannot be `pool_mint`.

Consider adding an explicit check that rejects the mints to `process_initialize` if `token_b.mint` and `pool_mint` are the same.

Resolution

Fixed by commit 08f6e88.

SOLANA-PSM

[L-02] Missing owner fee handling in single sided deposit and withdrawal

Identified in commit `de3823a`.

In the `deposit_single_token_type` and `withdraw_single_token_type_exact_out` functions, the `owner_fee` is calculated based on `half_source_amount`.

```
/* program/src/curve/base.rs */
118 | pub fn deposit_single_token_type(
119 |     &self,
120 |     source_amount: u128,
121 |     swap_token_a_amount: u128,
122 |     swap_token_b_amount: u128,
123 |     pool_supply: u128,
124 |     trade_direction: TradeDirection,
125 |     fees: &Fees,
126 |     timestamp: Option<u128>,
127 | ) -> Option<u128> {
134 |     let half_source_amount = std::cmp::max(1, source_amount.checked_div(2)?);
135 |     let trade_fee = fees.trading_fee(half_source_amount)?;
136 |     let owner_fee = fees.owner_trading_fee(half_source_amount)?;
137 |     let total_fees = trade_fee.checked_add(owner_fee)?;
138 |     let source_amount = source_amount.checked_sub(total_fees)?;

/* program/src/curve/base.rs */
150 | pub fn withdraw_single_token_type_exact_out(
151 |     &self,
152 |     source_amount: u128,
153 |     swap_token_a_amount: u128,
154 |     swap_token_b_amount: u128,
155 |     pool_supply: u128,
156 |     trade_direction: TradeDirection,
157 |     fees: &Fees,
158 |     timestamp: Option<u128>,
159 | ) -> Option<u128> {
167 |     let half_source_amount = source_amount.checked_add(1)?.checked_div(2)?; // round up
168 |     let pre_fee_source_amount = fees.pre_trading_fee_amount(half_source_amount)?;
169 |     let source_amount = source_amount
170 |         .checked_sub(half_source_amount)?
171 |         .checked_add(pre_fee_source_amount)?;
```

However, unlike the swap function, there is no logic to transfer or mint the `owner_fee` to the designated fee account.

In the swap function, the `owner_fee` is converted into pool tokens and minted to the `host_fee_account` and `pool_fee_account`.

```
/* program/src/processor.rs */
415 | pub fn process_swap(
416 |     program_id: &Pubkey,
```

```
417 |     amount_in: u64,
418 |     minimum_amount_out: u64,
419 |     accounts: &[AccountInfo],
420 | ) -> ProgramResult {
591 |     if result.owner_fee > 0 {
592 |         let mut pool_token_amount = token_swap
593 |             .swap_curve()
594 |             .calculator
595 |             .withdraw_single_token_type_exact_out(
596 |                 result.owner_fee,
597 |                 swap_token_a_amount,
598 |                 swap_token_b_amount,
599 |                 u128::from(pool_mint.supply),
600 |                 trade_direction,
601 |                 RoundDirection::Floor,
602 |                 token_swap.get_current_timestamp_opt()?
603 |             )
604 |             .ok_or(SwapError::FeeCalculationFailure)?;
              // mint host_fee if there is a host_fee_account
633 |         if token_swap
634 |             .check_pool_fee_info(pool_fee_account_info)
635 |             .is_ok()
636 |         {
637 |             Self::token_mint_to(
638 |                 swap_info.key,
639 |                 pool_token_program_info.clone(),
640 |                 pool_mint_info.clone(),
641 |                 pool_fee_account_info.clone(),
642 |                 authority_info.clone(),
643 |                 token_swap.bump_seed(),
644 |                 to_u64(pool_token_amount)?,
645 |             )?;
646 |         };
647 |     }
```

The `owner_fee` is ultimately deducted and remains in the pool alongside the `trade_fee` as an award to the liquidity provider rather than the pool owner.

It is recommended to properly distribute the `owner_fee` by following the same process used in the swap function.

Resolution

Fixed by commit [f3308e1](f3308e1).

SOLANA-PSM

[L-03] Prevent burning pool initial supply

Identified in commit `de3823a`.

When initializing a new pool using the `process_initialize` function, the protocol mints the initial supply of LP tokens to the `authority_info` token account provided by the pool creator. The creator is not required to deposit the corresponding underlying token A or B for this initial supply.

```
/* program/src/processor.rs */
245 | pub fn process_initialize(
246 |     program_id: &Pubkey,
247 |     fees: Fees,
248 |     swap_curve: SwapCurve,
249 |     accounts: &[AccountInfo],
250 |     swap_constraints: &Option<SwapConstraints>,
251 | ) -> ProgramResult {
259 |     let destination_info = next_account_info(account_info_iter)?;
356 |     let initial_amount = swap_curve.calculator.new_pool_supply();
358 |     Self::token_mint_to(
359 |         swap_info.key,
360 |         pool_token_program_info.clone(),
361 |         pool_mint_info.clone(),
362 |         destination_info.clone(),
363 |         authority_info.clone(),
364 |         bump_seed,
365 |         to_u64(initial_amount)?,
366 |     )?;

/* program/src/curve/calculator.rs */
009 | /// Initial amount of pool tokens for swap contract, hard-coded to something
010 | /// "sensible" given a maximum of u128.
011 | /// Note that on Ethereum, Uniswap uses the geometric mean of all provided
012 | /// input amounts, and Balancer uses 100 * 10 ^ 18.
013 | pub const INITIAL_SWAP_POOL_AMOUNT: u128 = 1_000_000_000;
088 | pub trait CurveCalculator: Debug + DynPack {
102 |     fn new_pool_supply(&self) -> u128 {
103 |         INITIAL_SWAP_POOL_AMOUNT
104 |     }
```

The purpose of the initial supply design is to help protect the liquidity pool from vulnerabilities such as inflation attacks. A similar approach is used in Uniswap V2 LP token minting mechanism.

```
/* contracts/UniswapV2Pair.sol */
119 | if (_totalSupply == 0) {
120 |     liquidity = Math.sqrt(amount0.mul(amount1)).sub(MINIMUM_LIQUIDITY);
121 |    _mint(address(0), MINIMUM_LIQUIDITY); // permanently lock the first MINIMUM_LIQUIDITY tokens
122 | }
```

However, there are no constraints on the pool creator's `authority_info` token account to ensure that the initial supply LP tokens are unburnable. This renders the initial supply mechanism ineffective.

The pool creator could reduce the total LP token supply to 1, then transfer tokens A and B into the pool's reserve accounts, inflating the LP token price and preventing new liquidity providers from joining (except for the offset curve, which does not support user-provided liquidity).

It is recommended to add a check to ensure that the pool's initial supply of LP tokens cannot be burned.

Resolution

Fixed by commit f470108.

SOLANA-PSM

[L-04] Validate token2022 extensions in the initialization

Identified in commit `de3823a`.

Currently, the `process_initialize` ensures that the `pool_mint` does not have the `MintCloseAuthority` extension enabled.

```
/* program/src/processor.rs */
245 | pub fn process_initialize(
246 |     program_id: &Pubkey,
247 |     fees: Fees,
248 |     swap_curve: SwapCurve,
249 |     accounts: &[AccountInfo],
250 |     swap_constraints: &Option<SwapConstraints>,
251 | ) -> ProgramResult {
276 |     let pool_mint = {
277 |         let pool_mint_data = pool_mint_info.data.borrow();
278 |         let pool_mint = Self::unpack_mint_with_extensions(
279 |             &pool_mint_data,
280 |             pool_mint_info.owner,
281 |             &token_program_id,
282 |         )?;
283 |         if let Ok(extension) = pool_mint.get_extension::<MintCloseAuthority>() {
284 |             let close_authority: Option<Pubkey> = extension.close_authority.into();
285 |             if close_authority.is_some() {
286 |                 return Err(SwapError::InvalidCloseAuthority.into());
287 |             }
288 |         }
289 |         pool_mint.base
290 |     };
```

However, more extensions should be rejected during the initialization process.

Tokens A and B

They are incompatible with the `PermanentDelegate` extension, which allows the delegate configured in the extension to transfer or burn the tokens from anyone's wallet, even if their owner is the `swap_info` PDA and their delegates configured in the token account are `None`.

The pool_mint

The pool mint should not have the `NonTransferable` extension installed. It doesn't allow token transfers, while the pool tokens transfers are needed in some operations.

Similarly, the pool mint should not have the `PermanentDelegate` extension.

Considering creating whitelists for `token_A_mint`, `token_B_mint`, and `pool_mint`. Please refer to Kamino Finance KLend VALID_LIQUIDITY_TOKEN_EXTENSIONS as an example for tokens A and B.

Resolution

Fixed by commit 658951e.

SOLANA-PSM

[I-01] Inconsistent signer requirement in the comment and code

Identified in commit `de3823a`.

The token swap, which is the `swap_info` account in the `Initialize` instruction, is required to be a signer.

```
/* program/src/instruction.rs */
142 | ///   Initializes a new swap
143 | ///
144 | ///   0. `[writable, signer]` New Token-swap to create.
145 | ///   1. `[]` swap authority derived from
146 | ///      `create_program_address(&[Token-swap account])`
147 | ///   2. `[]` token_a Account. Must be non zero, owned by swap authority.
148 | ///   3. `[]` token_b Account. Must be non zero, owned by swap authority.
149 | ///   4. `[writable]` Pool Token Mint. Must be empty, owned by swap
150 | ///      authority.
151 | ///   5. `[]` Pool Token Account to deposit trading and withdraw fees. Must
152 | ///      be empty, not owned by swap authority
153 | ///   6. `[writable]` Pool Token Account to deposit the initial pool token
154 | ///      supply. Must be empty, not owned by swap authority.
155 | ///   7. `[]` Pool Token program id
156 | Initialize(Initialize),
```

However, the implementation does not enforce the signer requirement. In fact, the `Initialize` can finish without `swap_info` being a signer.

If it's a design choice that only the party who can make `swap_info` a singer can call this `initialize` instruction, consider adding the singer check in the `process_initialize`.

Resolution

Fixed by commit f76b530.

SOLANA-PSM

[I-02] Redundant overflow check

Identified in commit `de3823a`.

In the `normalized_value()` function of the `ConstantPriceCurve`, the condition in lines `227-228` is to prevent overflows when calculating `swap_token_a_amount + swap_token_b_value`.

```
/* program/src/curve/constant_price.rs */
219 | fn normalized_value(
220 |     &self,
221 |     swap_token_a_amount: u128,
222 |     swap_token_b_amount: u128,
223 |     _timestamp: Option<u128>
224 | ) -> Option<PreciseNumber> {
225 |     let swap_token_b_value = swap_token_b_amount.checked_mul(self.token_b_price as u128)?;
226 |     // special logic in case we're close to the limits, avoid overflowing u128
227 |     let value = if swap_token_b_value.saturating_sub(u64::MAX.into())
228 |         > (u128::MAX.saturating_sub(u64::MAX.into()))
229 |     {
230 |         swap_token_b_value
231 |             .checked_div(2)?
232 |             .checked_add(swap_token_a_amount.checked_div(2)?)?
233 |     } else {
234 |         swap_token_a_amount
235 |             .checked_add(swap_token_b_value)?
236 |             .checked_div(2)?
237 |     };
238 |     PreciseNumber::new(value)
239 | }
```

However, this condition can be simplified to `swap_token_b_value > u128::MAX,` which is always false since the `swap_token_b_value` is of type `u128` and it cannot exceed `u128::MAX`.

Moreover, `swap_token_b_value` is computed as `swap_token_b_amount * token_b_price`.

```
/* spl-token-2022-6.0.0/src/state.rs */
109 | /// The amount of tokens this account holds.
110 | pub amount: u64, // `swap_token_b_amount` is `u64`

/* program/src/curve/constant_price.rs */
062 | pub struct ConstantPriceCurve {
063 |     /// Amount of token A required to get 1 token B
064 |     pub token_b_price: u64,
065 | }
```

Both `swap_token_b_amount` and `token_b_price` are at most `u64::MAX` ($2^64 - 1$), so the maximum possible value for `swap_token_b_value` is: $(2^{64} - 1) * (2^{64} - 1) = 2^{128} - 2^{65} - 1$.

Even in the worst case, `swap_token_b_value + swap_token_a_amount` is: $2^{128} - 2^{65} + 1 + (2^{64} - 1) = 2^{128} -$

2^{64}. It is still smaller than `u128::MAX`.

Therefore, there is no actual risk of overflows in the expression `swap_token_a_amount + swap_token_b_value`.

Consider removing the redundant overflow check for simplicity.

Resolution

Fixed by commit d6bfb82.

SOLANA-PSM

[I-03] The `ray` is not validated in `RedemptionRateCurve`

Identified in commit `de3823a`.

During the initialization process, the processor validates the curve parameters.

However, the `validate` function of the `RedemptionRateCurve` lacks a check to ensure that `ray` is not zero, which could lead to a division by zero error in the calculations later.

```
/* program/src/curve/redemption_rate.rs */
377 | fn validate(&self, timestamp: Option<u128>) -> Result<(), SwapError> {
378 |     let timestamp = timestamp
379 |         .ok_or(SwapError::MissingTimestamp)?;
380 |
381 |     let token_b_price = self.get_conversion_rate(timestamp)
382 |         .ok_or(SwapError::CalculationFailure)?;
383 |
384 |     if token_b_price == U256::zero() {
385 |         Err(SwapError::InvalidCurve)
386 |     } else {
387 |         Ok(())
388 |     }
389 | }
```

In particular, in the function `get_conversion_rate()`, if the `timestamp` equals the `rho`, it returns early at line 87 without computing `token_b_price`. As a result, even if the `self.ray = 0`, it may bypass validation and not trigger an error.

```
/* program/src/curve/redemption_rate.rs */
082 | pub fn get_conversion_rate(
083 |     &self,
084 |     timestamp: u128
085 | ) -> Option<U256> {
086 |     if timestamp == self.rho {
087 |         return Some(U256::from(self.chi))
088 |     }
089 |     let duration = timestamp.checked_sub(self.rho)?;
090 |     let rate = self._rpow(self.ssr, duration)? * U256::from(self.chi) / U256::from(self.ray);
091 |     Some(rate)
092 | }
```

Consider adding a check in the validate function to ensure that `self.ray` is a constant value (e.g., `1e27`).

Resolution

Fixed by commit 146b7e7.

SOLANA-PSM

[I-04] Incorrect `source_token_mint_info.owner` parameter

Identified in commit `de3823a`.

In the `process_swap` function, the protocol invokes the `unpack_mint_with_extensions` function to check the ownership of the user-provided Mint account and to unpack its data.

```
/* program/src/processor.rs */
415 | pub fn process_swap(
416 |     program_id: &Pubkey,
417 |     amount_in: u64,
418 |     minimum_amount_out: u64,
419 |     accounts: &[AccountInfo],
420 | ) -> ProgramResult {
483 |     let actual_amount_in = {
484 |         let source_mint_data = source_token_mint_info.data.borrow();
485 |         let source_mint = Self::unpack_mint_with_extensions(
486 |             &source_mint_data,
487 |             source_token_mint_info.owner,
488 |             token_swap.token_program_id(),
489 |         )?;
500 |     };
520 |     // Re-calculate the source amount swapped based on what the curve says
521 |     let (source_transfer_amount, source_mint_decimals) = {
522 |         let source_amount_swapped = to_u64(result.source_amount_swapped)?;
524 |         let source_mint_data = source_token_mint_info.data.borrow();
525 |         let source_mint = Self::unpack_mint_with_extensions(
526 |             &source_mint_data,
527 |             source_token_mint_info.owner,
528 |             token_swap.token_program_id(),
529 |         )?;
541 |     };
543 |     let (destination_transfer_amount, destination_mint_decimals) = {
544 |         let destination_mint_data = destination_token_mint_info.data.borrow();
545 |         let destination_mint = Self::unpack_mint_with_extensions(
546 |             &destination_mint_data,
547 |             source_token_mint_info.owner,
548 |             token_swap.token_program_id(),
549 |         )?;
```

And `unpack_mint_with_extensions()` ensures that the passed in account owner is either SPL token or SPL token-2022 program.

```
/* program/src/processor.rs */
077 | /// Unpacks a spl_token `Mint` with extension data
078 | pub fn unpack_mint_with_extensions<'a>(
079 |     account_data: &'a [u8],
080 |     owner: &Pubkey,
081 |     token_program_id: &Pubkey,
082 | ) -> Result<StateWithExtensions<'a, Mint>, SwapError> {
083 |     if owner != token_program_id && check_spl_token_program_account(owner).is_err() {
084 |         Err(SwapError::IncorrectTokenProgramId)
```

```
085 |     } else {
086 |         StateWithExtensions::<Mint>::unpack(account_data).map_err(|_| SwapError::ExpectedMint)
087 |     }
088 | }

/* spl-token-2022-6.0.0/src/lib.rs */
107 | pub fn check_spl_token_program_account(spl_token_program_id: &Pubkey) -> ProgramResult {
108 |     if spl_token_program_id != &id() && spl_token_program_id != &spl_token::id() {
109 |         return Err(ProgramError::IncorrectProgramId);
110 |     }
111 |     Ok(())
112 | }
```

However, when validating the `destination_token_mint_info` account at `processor.rs:547`, the function incorrectly passes `source_token_mint_info.owner` as the owner parameter.

It should instead pass `destination_token_mint_info.owner`.

However, since the `destination_token_mint_info` account is used in a token transfer CPI instruction at the end of `process_swap` function, the user should not be able to provide a forged account due to the check by the SPL token program.

```
/* program/src/processor.rs */
415 | pub fn process_swap(
416 |     program_id: &Pubkey,
417 |     amount_in: u64,
418 |     minimum_amount_out: u64,
419 |     accounts: &[AccountInfo],
420 | ) -> ProgramResult {
432 |     let destination_token_mint_info = next_account_info(account_info_iter)?;
543 |     let (destination_transfer_amount, destination_mint_decimals) = {
544 |         let destination_mint_data = destination_token_mint_info.data.borrow();
545 |         let destination_mint = Self::unpack_mint_with_extensions(
546 |             &destination_mint_data,
547 |             source_token_mint_info.owner,
548 |             token_swap.token_program_id(),
549 |         )?;
566 |     };
649 |     Self::token_transfer(
650 |         swap_info.key,
651 |         destination_token_program_info.clone(),
652 |         swap_destination_info.clone(),
653 |         destination_token_mint_info.clone(),
654 |         destination_info.clone(),
655 |         authority_info.clone(),
656 |         token_swap.bump_seed(),
657 |         destination_transfer_amount,
658 |         destination_mint_decimals,
659 |     )?;

/* spl-token-2022-6.0.0/src/processor.rs */
328 | if let Some((mint_info, expected_decimals)) = expected_mint_info {
329 |     if &source_account.base.mint != mint_info.key {
330 |         return Err(TokenError::MintMismatch.into());
331 |     }
```

It is recommended to replace the `source_token_mint_info.owner` with the `destination_token_mint_info.owner`.

Resolution

Fixed by commit 1376a8f.

SOLANA-PSM

[I-05] Duplicated owner fee in pool token amount calculation

Identified in commit `de3823a`.

During the swap process, the `owner_fee` is supposed to be converted into pool tokens of equivalent value and minted to the `pool_fee_account`.

Taking the constant curve as an example.

Assuming a swap in the `AtoB` direction, the expected amount of pool tokens to be minted for the `owner_fee` should be: `pool_token_amount = pool_supply * owner_fee / (x + price_b * y)`, where `x` and `y` are the token A and token B amounts in the pool before the minting.

However, in practice, the variables `new_swap_source_amount` and `new_swap_destination_amount` used for this calculation (L107–L108) already include `owner_fee`. As a result, the actual computation becomes: `pool_token_amount = pool_supply * owner_fee / (x + owner_fee + price_b * y)`

This leads to a slight underestimation of `pool_token_amount`, meaning the pool tokens minted for the `owner_fee` are less than expected.

```
/* program/src/curve/base.rs */
075 | impl SwapCurve {
078 |     pub fn swap(
079 |         &self,
080 |         source_amount: u128,
081 |         swap_source_amount: u128,
082 |         swap_destination_amount: u128,
083 |         trade_direction: TradeDirection,
084 |         fees: &Fees,
085 |         timestamp: Option<u128>
086 |     ) -> Option<SwapResult> {
105 |         let source_amount_swapped = source_amount_swapped.checked_add(total_fees)?;
106 |         Some(SwapResult {
                    // @audit: owner_fee, trade_fee included
107 |             new_swap_source_amount: swap_source_amount.checked_add(source_amount_swapped)?,
108 |             new_swap_destination_amount: swap_destination_amount
109 |                 .checked_sub(destination_amount_swapped)?,
110 |             source_amount_swapped, // @auidt: owner_fee,trade_fee included
111 |             destination_amount_swapped,
112 |             trade_fee,
113 |             owner_fee,
114 |         })

/* program/src/processor.rs */
415 | pub fn process_swap(
416 |     program_id: &Pubkey,
417 |     amount_in: u64,
418 |     minimum_amount_out: u64,
```

```
419 |      accounts: &[AccountInfo],
420 |  ) -> ProgramResult {
          // @audit: new reserve_a, reserve_b  (source with fees)
568 |      let (swap_token_a_amount, swap_token_b_amount) = match trade_direction {
569 |          TradeDirection::AtoB => (
570 |              result.new_swap_source_amount,
571 |              result.new_swap_destination_amount,
572 |          ),
577 |      };
591 |      if result.owner_fee > 0 {
592 |          let mut pool_token_amount = token_swap
593 |              .swap_curve()
594 |              .calculator
595 |              .withdraw_single_token_type_exact_out(
596 |                  result.owner_fee,
597 |                  swap_token_a_amount,
598 |                  swap_token_b_amount,
599 |                  u128::from(pool_mint.supply),
600 |                  trade_direction,
601 |                  RoundDirection::Floor,
602 |                  token_swap.get_current_timestamp_opt()?
603 |              )
604 |              .ok_or(SwapError::FeeCalculationFailure)?;
```

Although this miscalculation does not pose a security risk, it still results in a slight loss for the pool owner. The portion of LP tokens that the owner does not receive is eventually distributed among all existing pool token holders.

Resolution

Fixed by commit [f61959d](f61959d).

SOLANA-PSM

[I-06] Potential fee overcharge during token swap

Identified in commit `de3823a`.

In the `swap` function, the value `source_amount_less_fees` is calculated by subtracting the total fees `owner fee + trade fee` from the original `source_amount`. This value is then passed to the `swap_without_fees` function to determine the `destination_amount_swapped`.

```
/* program/src/curve/base.rs */
078 | pub fn swap(
079 |     &self,
080 |     source_amount: u128,
081 |     swap_source_amount: u128,
082 |     swap_destination_amount: u128,
083 |     trade_direction: TradeDirection,
084 |     fees: &Fees,
085 |     timestamp: Option<u128>
086 | ) -> Option<SwapResult> {
087 |     // debit the fee to calculate the amount swapped
088 |     let trade_fee = fees.trading_fee(source_amount)?;
089 |     let owner_fee = fees.owner_trading_fee(source_amount)?;
090 |
091 |     let total_fees = trade_fee.checked_add(owner_fee)?;
092 |     let source_amount_less_fees = source_amount.checked_sub(total_fees)?;
093 |
094 |     let SwapWithoutFeesResult {
095 |         source_amount_swapped,
096 |         destination_amount_swapped,
097 |     } = self.calculator.swap_without_fees(
098 |         source_amount_less_fees,
```

Consider a scenario of the `AtoB` trade direction using the `reddemption_rate` curve. If the `source_amount * ray / token_b_price` is not evenly divisible, the recalculated `source_amount_used` will be less than the original `source_amount_less_fees`.

```
/* program/src/curve/redemption_rate.rs */
227 | fn swap_without_fees(
228 |     &self,
229 |     source_amount: u128,
230 |     _swap_source_amount: u128,
231 |     _swap_destination_amount: u128,
232 |     trade_direction: TradeDirection,
233 |     timestamp: Option<u128>
234 | ) -> Option<SwapWithoutFeesResult> {
235 |     let token_b_price = self.get_conversion_rate(timestamp?)?;
236 |     let source_amount = U256::from(source_amount);
237 |     let ray = U256::from(self.ray);
241 |         TradeDirection::AtoB => {
242 |             let destination_amount = source_amount
243 |                 .checked_mul(ray)?
```

```
244 |                 .checked_div(token_b_price)?;
245 |
246 |             let (source_amount_used, _) = destination_amount
247 |                 .checked_mul(token_b_price)?
248 |                 .checked_ceil_div(ray)?;
251 |             if source_amount_used > source_amount {
252 |                 return None;
253 |             }
255 |             (source_amount_used, destination_amount)
```

Consider the following setting:

- `source_amount` is `100`
- `token_b_price` is `11`
- `ray` is `10`

The computed `destination_amount` is: `100 * 10 / 11 = 90` (rounded floor)

Then, `source_amount_used` is: `90 * 11 / 10 = 99` (rounded ceil)

So, the `source_amount_used` is smaller than the `source_amount`.

The final `new_swap_source_amount`, which represents the user's actual input token amount, is calculated as: `source_amount_used + total fees`. The `new_swap_source_amount` will be smaller than the `source_amount`.

```
/* program/src/curve/base.rs */
105 | let source_amount_swapped = source_amount_swapped.checked_add(total_fees)?;
106 | Some(SwapResult {
107 |     new_swap_source_amount: swap_source_amount.checked_add(source_amount_swapped)?,
```

However, the owner and trade fees are still calculated based on the original `source_amount`, not the user's actual input token amount. This results in fee overcharging.

The difference between `source_amount_used` and `source_amount` is minimal; this issue does not pose a security risk.

It is recommended to recalculate the owner and trade fees after the `swap_without_fees` recalculates the `source_amount_used`.

Resolution

Fixed by commit c38f213.

SOLANA-PSM

[I-07] Profitable sandwich attack on rate update transaction

Identified in commit `de3823a`.

A permissioned user can update the redemption rate curve state using the `process_curve_update` function. In particular, the curve's `chi` and `ssr` will be updated.

```
/* program/src/redemption_rate_processor.rs */
026 | pub fn process_curve_update(
027 |     program_id: &Pubkey,
028 |     accounts: &[AccountInfo],
029 |     ssr: u128,
030 |     rho: u128,
031 |     chi: u128
032 | ) -> Result<(), ProgramError> {
054 |     let new_swap_state = create_new_swap_state(
055 |         ssr,
056 |         rho,
057 |         chi,
058 |         curve,
059 |         swap
060 |     )?;
062 |     SwapVersion::pack(new_swap_state, &mut swap_data)?;
```

According to the `get_conversion_rate` function, the `token_b` price is derived from these `chi` and `ssr` values. As a result, updating the curve effectively changes the `token_b` price.

```
/* program/src/curve/redemption_rate.rs */
082 | pub fn get_conversion_rate(
083 |     &self,
084 |     timestamp: u128
085 | ) -> Option<U256> {
086 |     if timestamp == self.rho {
087 |         return Some(U256::from(self.chi))
088 |     }
089 |     let duration = timestamp.checked_sub(self.rho)?;
090 |     let rate = self._rpow(self.ssr, duration)? * U256::from(self.chi) / U256::from(self.ray);
091 |     Some(rate)
092 | }
```

The `set_rate` function includes a check to ensure that the new chi value does not exceed `max_chi` (the theoretical price ceiling), which prevents the new `token_b` price from significantly deviating from the original.

However, when the pool has sufficient liquidity, even small price deviations can create arbitrage opportunities.

```
/* program/src/curve/redemption_rate.rs */
163 | pub fn set_rates(
164 |      &self,
165 |      ssr: u128,
166 |      rho: u128,
167 |      chi: u128,
168 |      current_timestamp: u128,
169 |  ) -> Result<RedemptionRateCurve, ProgramError> {
180 |      let new_calculator = if self.rho == 0 {
             // ....
188 |      } else {
189 |          if rho < self.rho {
190 |              return Err(SwapError::InvalidRho.into())
191 |          }
192 |          if chi < self.chi {
193 |              return Err(SwapError::InvalidChi.into())
194 |          }
195 |          if self.max_ssr != 0 {
196 |              let duration = rho
197 |                  .checked_sub(self.rho)
198 |                  .ok_or(ProgramError::ArithmeticOverflow)?;
200 |              let chi_max = self._rpow(self.max_ssr, duration)
201 |                  .ok_or(SwapError::CalculationFailure)?
202 |                  .checked_mul(U256::from(self.chi))
203 |                  .ok_or(ProgramError::ArithmeticOverflow)?
204 |                  .checked_div(U256::from(self.ray))
205 |                  .ok_or(ProgramError::ArithmeticOverflow)?;
208 |              if U256::from(chi) > chi_max {
209 |                  return Err(SwapError::InvalidChi.into())
210 |              }
211 |          }
```

An arbitrager could execute transactions in the following orders:

1. First transaction: Swap `token_a` into the pool to acquire a large amount of `token_b`.
2. Second transaction: Submit the permissioned curve update that increases the `token_b` price.
3. Third transaction: Swap the previously acquired `token_b` back into the pool at the higher rate, generating a profit.

It is recommended to limit the price fluctuation range to mitigate this issue.

Resolution

The team acknowledged this finding.

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 Nova (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.