Security Assessment Report

Flipcash Program

January 08, 2026

Summary

The Sec3 team was engaged to conduct a thorough security analysis of the Flipcash Program.

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 9 issues or questions.

#	Task	Type	Commit
P1	flipcash api	Solana	`6c985fd` (Oct 08, 2025)
P2	flipcash program	Solana	`6c985fd` (Oct 08, 2025)

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

Table of Contents

Result Overview

Issue	Impact	Status
FLIPCASH API		
[P1-H-01] Unsafe rounding caused by `UnsignedNumeric`	High	Resolved
[P1-L-01] Potential DoS if accounts have lamports before initialization	Low	Resolved
[P1-I-01] Unnecessary signer	Info	Resolved
[P1-I-02] Minor boundary handling issue in `DiscreteExponentialCurve`	Info	Acknowledged
FLIPCASH PROGRAM		
[P2-L-01] Inconsistent writable specifications	Low	Resolved
[P2-L-02] User may lose funds on overpayment	Low	Resolved
[P2-L-03] Missing decimal check on base_mint	Low	Resolved
[P2-I-01] Unnecessary bump passed in args	Info	Acknowledged
[P2-I-02] Misleading ATA variable names	Info	Resolved

Findings in Detail

FLIPCASH API

[P1-H-01] Unsafe rounding caused by `UnsignedNumeric`

> *Identified in commit* `6c985fd`.

The program involves complex calculations, such as logarithms and exponents, and uses the `UnsignedNumeric` type for fixed-point decimal arithmetic.

The `from_numeric` function is responsible for converting the final fixed-point number back into an integer. This process uses the `to_imprecise` function, which rounds to the nearest value. This type of rounding can lead to users receiving more tokens than they should and the pool collecting less in fees.

```
/* api/src/utils.rs */
096 | pub fn from_numeric(value: UnsignedNumeric, decimal_places: u8) -> Result<u64, ProgramError> {
107 |     let result = value.checked_mul(&multiplier)
108 |         .and_then(|r| r.to_imprecise())
109 |         .ok_or(ProgramError::InvalidArgument)?;
110 |
111 |     u64::try_from(result).map_err(|_| ProgramError::InvalidArgument)
112 | }

/* https://github.com/zfedoran/brine-fp/blob/188e023b273a8f97361198648614137cb2045822/src/unsigned.rs#L139-L161 */
139 | pub fn to_imprecise(&self) -> Option<u128> {
140 |     self.value
141 |         .checked_add(Self::rounding_correction())?
142 |         .checked_div(one())
143 |         .map(|v| v.as_u128())
144 | }

159 | fn rounding_correction() -> InnerUint {
160 |     InnerUint::from(ONE / 2)
161 | }
```

Furthermore, the rounding direction issue exists not only when converting from `UnsignedNumeric` to `u64`, its multiplication and division operations round the results to the nearest to handle precision loss. This rounding behavior means the internal precision cannot be controlled, which can cause small errors to accumulate and lead to unexpected final calculations that are rounded up or down.

```
/* https://github.com/zfedoran/brine-fp/blob/188e023b273a8f97361198648614137cb2045822/src/unsigned.rs#L209-L251 */
205 | pub fn checked_div(&self, rhs: &Self) -> Option<Self> {
206 |     if *rhs == Self::zero() {
207 |         return None;
208 |     }
209 |     match self.value.checked_mul(one()) {
```

```
210 |         Some(v) => {
211 |             let value = v
212 |                 .checked_add(Self::rounding_correction())?
213 |                 .checked_div(rhs.value)?;
214 |             Some(Self { value })
215 |         }
216 |         None => {
217 |             let value = self
218 |                 .value
219 |                 .checked_add(Self::rounding_correction())?
220 |                 .checked_div(rhs.value)?
221 |                 .checked_mul(one())?;
222 |             Some(Self { value })
223 |         }
224 |     }
225 | }

230 | pub fn checked_mul(&self, rhs: &Self) -> Option<Self> {
231 |     match self.value.checked_mul(rhs.value) {
232 |         Some(v) => {
233 |             let value = v
234 |                 .checked_add(Self::rounding_correction())?
235 |                 .checked_div(one())?;
236 |             Some(Self { value })
237 |         }
238 |         None => {
239 |             let value = if self.value >= rhs.value {
240 |                 self.value.checked_div(one())?.checked_mul(rhs.value)?
241 |             } else {
242 |                 rhs.value.checked_div(one())?.checked_mul(self.value)?
243 |             };
244 |             Some(Self { value })
245 |         }
246 |     }
247 | }
```

This rounding issue has the following specific effects:

1. The collected fees may be lower than intended because `fee_amount_raw` is also subject to rounding when using `from_numeric`. This could also block some small transactions if their calculated fees are rounded down to zero.

```
/* program/src/instruction/buy.rs */
004 | pub fn process_buy_tokens(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
095 |     let fee_amount_raw = from_numeric(fee_amount.clone(), mint_a_decimals)?;
102 |     if pool.buy_fee > 0 {
103 |         check_condition(
104 |             fee_amount_raw > 0,
105 |             "No fees generated"
106 |         )?;
107 |     }
135 |     if fee_amount_raw > 0 {
136 |         transfer_signed_with_bump(
141 |             fee_amount_raw,
148 |         )?;
149 |     }
```

```
152 | }
```

2. An attacker could exploit the rounding to gain tokens from the pool.

Based on the test case, an attacker can gain 50 `token_a` for free through a pair of buy and sell operations. This problem can still occur even if the fees are not zero. In the example, the attacker took half of the 100 `token_a` from the swap. This means the attack is profitable as long as the `buy_fee` is less than 50 percent (without considering network fees).

```
#[test]
fn test_rounding_issue() {
    use crate::prelude::{to_numeric, from_numeric};
    let a = UnsignedNumeric::from_scaled_u128(CURVE_A);
    let b = UnsignedNumeric::from_scaled_u128(CURVE_B);
    let c = UnsignedNumeric::from_scaled_u128(CURVE_C);

    let curve = ExponentialCurve {
        a: a.clone(),
        b: b.clone(),
        c: c.clone(),
    };

    let token_a_decimal = TOKEN_DECIMALS;
    let token_b_decimal = TOKEN_DECIMALS;
    let steal_token_a_amount = 50u64;

    let mut tokens_left = MAX_TOKEN_SUPPLY * QUARKS_PER_TOKEN;
    let mut value_left = 0u64;

    let token_b_in: u64 = 1;

    let supply_from_bonding = (MAX_TOKEN_SUPPLY * QUARKS_PER_TOKEN) - tokens_left;
    let token_a_out_fp = curve.value_to_tokens(
        &to_numeric(supply_from_bonding, token_a_decimal).unwrap(),
        &to_numeric(token_b_in, token_b_decimal).unwrap()
    ).unwrap();

    let token_a_out = from_numeric(token_a_out_fp.clone(), token_a_decimal).unwrap();
    println!("token_b_in: {}, token_a_out_fp: {}, token_a_out: {}", token_b_in, token_a_out_fp.to_string(),
    ↪  token_a_out);

    value_left += token_b_in;
    tokens_left -= token_a_out;


    let token_a_in = token_a_out - steal_token_a_amount;

    let token_b_out_2_fp = curve.tokens_to_value_from_current_value(
        &to_numeric(value_left, token_b_decimal).unwrap(),
        &to_numeric(token_a_in, token_a_decimal).unwrap()
    ).unwrap();

    let token_b_out_2 = from_numeric(token_b_out_2_fp.clone(), token_b_decimal).unwrap();
    println!("token_a_in: {}, token_b_out_2_fp: {}, token_b_out_2: {}", token_a_in, token_b_out_2_fp.to_string(),
    ↪  token_b_out_2);
```

```
    value_left -= token_b_out_2;
    tokens_left += token_a_in;

    assert!(value_left == 0);
    assert!(tokens_left + steal_token_a_amount == MAX_TOKEN_SUPPLY * QUARKS_PER_TOKEN);

    println!("stolen: {}", steal_token_a_amount);
}
```

test output:

```
---- curve::tests::test_rounding_issue stdout ----
token_b_in: 1, token_a_out_fp: 0.000000009998571853, token_a_out: 100
token_a_in: 50, token_b_out_2_fp: 0.000000000050001409, token_b_out_2: 1
stolen: 50
```

So the `UnsignedNumeric` type used in the program is not suitable for secure on-chain DeFi math, it is recommended to replace `UnsignedNumeric` with a fixed-point number implementation that supports selectable rounding directions.When rounding, the direction chosen should always be favorable to the pool to protect its assets.

Resolution

The rounding issues in the curve logic have been fully resolved in commit `096d8be`.

While the rounding direction when converting the final calculated decimal values into integer token amounts is not in favor of the pool, the current method calculates the difference between the total cumulative state (from zero to the present) and the pool's recorded state. This ensures that rounding errors do not accumulate across multiple transactions.

FLIPCASH API

[P1-L-01] Potential DoS if accounts have lamports before initialization

Identified in commit `6c985fd`.

The program's custom `create_token_account` and `create_mint_account` functions use the system program's `create_account` instruction.

```
/* api/src/cpis.rs */
008 | pub fn create_token_account<'info>(
015 | ) -> ProgramResult {
025 |     // Create the account with system program
026 |     solana_program::program::invoke_signed(
027 |         &system_instruction::create_account(
028 |             payer.key,
029 |             target.key,
030 |             required_lamports,
031 |             spl_token::state::Account::LEN as u64,
032 |             &spl_token::id(),
033 |         ),
040 |     )?;
058 | }

060 | pub fn create_mint_account<'info>(
069 | ) -> ProgramResult {
079 |     // Create the account with system program
080 |     solana_program::program::invoke_signed(
081 |         &system_instruction::create_account(
082 |             payer.key,
083 |             mint.key,
084 |             required_lamports,
085 |             spl_token::state::Mint::LEN as u64,
086 |             &spl_token::id(),
087 |         ),
094 |     )?;
113 | }
```

The `system_instruction::create_account` instruction fails if the target account already has lamports. This means that if someone sends lamports to an account address before it is created, the creation will fail, preventing the program from working correctly.

For the mint account, the chance of this DoS attack is low. It is created in `process_initialize_currency`, and its seeds include a 32-byte random value provided by the user, making its address difficult to predict. Even if predicted, a new seed can be used.

However, the token accounts created in `process_initialize_pool` have easily predictable addresses. For example, the seeds for `target_vault_info` include the pool address and the target mint address. Both of these are known values, making it possible for an attacker to send lamports to the address and prevent

the pool from being created.

A better way to create accounts can be found in the `steel` library, which this program already depends on: If the account has no lamports, it uses `create_account`. If the account already has lamports, it uses a combination of `transfer`, `allocate`, and `assign` instructions. This approach prevents the DoS attack where someone might pre-fund an account before it is created.

```
/* https://docs.rs/crate/steel/4.0.2/source/src/account/cpi.rs#137-212 */
139 | pub fn allocate_account_with_bump<'a, 'info>(
147 | ) -> ProgramResult {
155 |     // Allocate space for account
156 |     let rent = Rent::get()?;
157 |     if target_account.lamports().eq(&0) {
158 |         // If balance is zero, create account
159 |         solana_program::program::invoke_signed(
160 |             &solana_program::system_instruction::create_account(
161 |                 payer.key,
162 |                 target_account.key,
163 |                 rent.minimum_balance(space),
164 |                 space as u64,
165 |                 owner,
166 |             ),
167 |             &[
168 |                 payer.clone(),
169 |                 target_account.clone(),
170 |                 system_program.clone(),
171 |             ],
172 |             &[seeds],
173 |         )?;
174 |     } else {
175 |         // Otherwise, if balance is nonzero:
176 |
177 |         // 1) transfer sufficient lamports for rent exemption
178 |         let rent_exempt_balance = rent
179 |             .minimum_balance(space)
180 |             .saturating_sub(target_account.lamports());
181 |         if rent_exempt_balance.gt(&0) {
182 |             solana_program::program::invoke(
183 |                 &solana_program::system_instruction::transfer(
184 |                     payer.key,
185 |                     target_account.key,
186 |                     rent_exempt_balance,
187 |                 ),
188 |                 &[
189 |                     payer.clone(),
190 |                     target_account.clone(),
191 |                     system_program.clone(),
192 |                 ],
193 |             )?;
194 |         }
195 |
196 |         // 2) allocate space for the account
197 |         solana_program::program::invoke_signed(
198 |             &solana_program::system_instruction::allocate(target_account.key, space as u64),
199 |             &[target_account.clone(), system_program.clone()],
200 |             &[seeds],
201 |         )?;
```

```
202 |
203 |         // 3) assign our program as the owner
204 |         solana_program::program::invoke_signed(
205 |             &solana_program::system_instruction::assign(target_account.key, owner),
206 |             &[target_account.clone(), system_program.clone()],
207 |             &[seeds],
208 |         )?;
209 |     }
210 |
211 |     Ok(())
212 | }
```

Resolution

Fixed by commit `9a0e4b3`.

FLIPCASH API

[P1-I-01] Unnecessary signer

Identified in commit `6c985fd`.

The program passes unnecessary seeds when calling `initialize_account` and `initialize_mint` instructions in the `spl_token` program.

The `initialize_account` instruction does not require the `target` account to be a signer, and `initialize_mint` does not require the `mint` account to be a signer. It is recommended to remove these unnecessary seeds and switch to a simple `invoke` call.

```
/* api/src/cpis.rs */
008 | pub fn create_token_account<'info>(
015 | ) -> ProgramResult {
042 |     // Initialize the PDA.
043 |     solana_program::program::invoke_signed(
044 |         &spl_token::instruction::initialize_account(
045 |             &spl_token::id(),
046 |             target.key,
047 |             mint.key,
048 |             target.key,
049 |         ).unwrap(),
050 |         &[
051 |             target.clone(),
052 |             mint.clone(),
053 |             target.clone(),
054 |             rent_sysvar.clone(),
055 |         ],
056 |         &[seeds],
057 |     )
058 | }

060 | pub fn create_mint_account<'info>(
069 | ) -> ProgramResult {
096 |     // Initialize the mint
097 |     solana_program::program::invoke_signed(
098 |         &spl_token::instruction::initialize_mint(
099 |             &spl_token::id(),
100 |             mint.key,
101 |             mint_authority,
102 |             freeze_authority,
103 |             decimals,
104 |         )?,
105 |         &[
106 |             mint.clone(),
107 |             rent_sysvar.clone(),
108 |         ],
109 |         &[seeds],
110 |     )?;
111 |
112 |     Ok(())
113 | }
```

```
/*
↪ https://github.com/solana-program/token/blob/858ef63cb06c863fa3f943df56df37282047e93e/interface/src/instruction.rs#L24-L63
↪ */
024 | /// Initializes a new mint and optionally deposits all the newly minted
025 | /// tokens in an account.
026 | ///
027 | /// The `InitializeMint` instruction requires no signers and MUST be
028 | /// included within the same Transaction as the system program's
029 | /// `CreateAccount` instruction that creates the account being initialized.
030 | /// Otherwise another party can acquire ownership of the uninitialized
031 | /// account.
032 | ///
033 | /// Accounts expected by this instruction:
034 | ///
035 | ///   0. `[writable]` The mint to initialize.
036 | ///   1. `[]` Rent sysvar
037 | InitializeMint {
038 |     /// Number of base 10 digits to the right of the decimal place.
039 |     decimals: u8,
040 |     /// The authority/multisignature to mint tokens.
041 |     mint_authority: Pubkey,
042 |     /// The freeze authority/multisignature of the mint.
043 |     freeze_authority: COption<Pubkey>,
044 | },
045 | /// Initializes a new account to hold tokens.  If this account is associated
046 | /// with the native mint then the token balance of the initialized account
047 | /// will be equal to the amount of SOL in the account. If this account is
048 | /// associated with another mint, that mint must be initialized before this
049 | /// command can succeed.
050 | ///
051 | /// The `InitializeAccount` instruction requires no signers and MUST be
052 | /// included within the same Transaction as the system program's
053 | /// `CreateAccount` instruction that creates the account being initialized.
054 | /// Otherwise another party can acquire ownership of the uninitialized
055 | /// account.
056 | ///
057 | /// Accounts expected by this instruction:
058 | ///
059 | ///   0. `[writable]`  The account to initialize.
060 | ///   1. `[]` The mint this account will be associated with.
061 | ///   2. `[]` The new account's owner/multisignature.
062 | ///   3. `[]` Rent sysvar
063 | InitializeAccount,
```

Resolution

Fixed by commit `5f4ad38`.

FLIPCASH API

[P1-I-02] Minor boundary handling issue in `DiscreteExponentialCurve`

Identified in commit `096d8be`.

The current `DiscreteExponentialCurve` implementation contains a minor issue when processing tokens or values that reach the upper limit. It allows the final result to exceed the permitted max token/value by one step size.

First, the final entries in both `DISCRETE_CUMULATIVE_VALUE_TABLE` and `DISCRETE_PRICING_TABLE` correspond to the price and cumulative value at the exact upper limit. So, only this specific point is valid, actually. Any values or tokens within the `DISCRETE_PRICING_STEP_SIZE` range following this step should be invalid.

```
/* api/src/table.rs */
210012 | pub static DISCRETE_CUMULATIVE_VALUE_TABLE: &[u128] = &[
420013 |     113997300431503234258171693950O,  // Supply: 21000000
```

But in the current `tokens_to_value` and `value_to_tokens` functions, the logic only restricts the point corresponding to the new supply to be within the `end_step` range. It treats the final step as a standard step, allowing the end point to fall within the `DISCRETE_PRICING_STEP_SIZE` range following it.

The correct constraint should ensure that when `end_step == DISCRETE_PRICING_TABLE.len() - 1`, the remaining value or token amount must be exactly zero.

```
/* api/src/curve.rs */
104 | pub fn tokens_to_value(
108 | ) -> Option<UnsignedNumeric> {
116 |     let end_supply = current_supply.checked_add(tokens)?;
122 |     let end_step = end_supply
123 |         .checked_div(&step_size)?
124 |         .floor()?
125 |         .to_imprecise()? as usize;
126 |
127 |     if end_step >= DISCRETE_PRICING_TABLE.len() {
128 |         return None;
129 |     }
165 | }

167 | pub fn value_to_tokens(
171 | ) -> Option<UnsignedNumeric> {
225 |     // low is now the last step where cumulative <= target
226 |     let end_step = low;
227 |
228 |     if end_step >= DISCRETE_PRICING_TABLE.len() {
229 |         return None;
230 |     }
249 | }
```

However, the callers `buy_common` and `sell_common` enforce additional restrictions on value and token amounts that exceed these boundaries, so this issue does not occur in practice and poses no actual risk.

The potential impact is limited to off-chain applications that directly utilize these API functions, which may encounter unexpected results.

```
/* program/src/instruction/sell.rs */
139 | fn sell_common<'info>(
151 | ) -> Result<u64, ProgramError>{
182 |     let supply_from_bonding = MAX_TOKEN_SUPPLY
183 |         .checked_mul(QUARKS_PER_TOKEN)
184 |         .ok_or(ProgramError::InvalidArgument)?
185 |         .checked_sub(tokens_left_raw)
186 |         .ok_or(ProgramError::InvalidArgument)?;
          // @audit: supply_from_bonding is always smaller than MAX_TOKEN_SUPPLY, so the new supply pass into
          ↪  tokens_to_value is also lower than MAX_TOKEN_SUPPLY.
198 |     let new_supply = to_numeric(supply_from_bonding, mint_a_decimals)?
199 |         .checked_sub(&in_amount)
200 |         .unwrap();
206 |     let new_value = curve.tokens_to_value(&zero, &new_supply)
207 |         .ok_or(ProgramError::InvalidArgument)?;
257 | }

/* program/src/instruction/buy.rs */
136 | fn buy_common<'info>(
148 | ) -> Result<u64, ProgramError>{
          // @audit: capped under SCALED_MAX_CUMULATIVE_VALUE
202 |     let max_cumulative_value = UnsignedNumeric::from_scaled_u128(SCALED_MAX_CUMULATIVE_VALUE);
203 |     let capped_new_value = if uncapped_new_value.greater_than(&max_cumulative_value) {
204 |         max_cumulative_value
205 |     } else {
206 |         uncapped_new_value
207 |     };
214 |     let new_supply = curve.value_to_tokens(&zero, &capped_new_value)
215 |         .ok_or(ProgramError::InvalidArgument)?;
247 | }
```

Below are two unit tests demonstrating the issue:

```
#[test]
fn test_discrete_tokens_to_value_respects_max_supply_from_tables() {
    let curve = DiscreteExponentialCurve::default();
    let zero_supply = UnsignedNumeric::zero();

    // Maximum supply that the tables encode at whole steps
    let max_step = DISCRETE_CUMULATIVE_VALUE_TABLE.len() - 1;
    let max_supply_u128 = (max_step as u128) * DISCRETE_PRICING_STEP_SIZE;
    let max_tokens = UnsignedNumeric::new(max_supply_u128).unwrap();

    assert!(
        (MAX_TOKEN_SUPPLY as u128) == max_supply_u128,
        "MAX_TOKEN_SUPPLY incorrect"
    );

    // Up to max_supply should be representable
    let value_at_max = curve.tokens_to_value(&zero_supply, &max_tokens);
```

```
    assert!(
        value_at_max.is_some(),
        "tokens_to_value should be defined up to max_supply encoded by the tables"
    );

    // Going one token beyond max_supply should be considered out of range
    let too_many_tokens = UnsignedNumeric::new(max_supply_u128 + 1).unwrap();
    assert!(
        curve.tokens_to_value(&zero_supply, &too_many_tokens).is_none(),
        "tokens_to_value should return None when requested tokens exceed max_supply encoded by the tables"
    );
}

#[test]
fn test_discrete_value_to_tokens_respects_max_cumulative_from_tables() {
    let curve = DiscreteExponentialCurve::default();
    let zero_supply = UnsignedNumeric::zero();

    // Maximum cumulative value encoded by the tables
    let max_step = DISCRETE_CUMULATIVE_VALUE_TABLE.len() - 1;
    let max_supply_u128 = (max_step as u128) * DISCRETE_PRICING_STEP_SIZE;
    let max_supply_numeric = UnsignedNumeric::new(max_supply_u128).unwrap();

    let max_cumulative_raw = DISCRETE_CUMULATIVE_VALUE_TABLE[max_step];
    let max_cumulative = UnsignedNumeric::from_scaled_u128(max_cumulative_raw);

    // At max cumulative value, the number of tokens should not exceed max_supply
    let tokens_at_max = curve
        .value_to_tokens(&zero_supply, &max_cumulative)
        .expect("value_to_tokens should be defined up to max cumulative value encoded by the tables");
    assert!(
        tokens_at_max.less_than_or_equal(&max_supply_numeric),
        "value_to_tokens at max cumulative should not exceed max_supply encoded by the tables"
    );

    // Going beyond the maximum cumulative value should be considered out of range
    let too_much_value = UnsignedNumeric::from_scaled_u128(max_cumulative_raw + 1);
    assert!(
        curve.value_to_tokens(&zero_supply, &too_much_value).is_none(),
        "value_to_tokens should return None when value exceeds max cumulative encoded by the tables"
    );
}
```

Resolution

The team acknowledged this finding.

FLIPCASH PROGRAM

[P2-L-01] Inconsistent writable specifications

Identified in commit `6c985fd`.

The program uses `check_mut` to verify if an account is writable and uses `as_account_mut` to deserialize modifiable accounts. However, there are inconsistencies where the mutability requirements do not match the actual usage. For example, some accounts are required to be mutable but are not modified, while others are modified without a mutability check.

Accounts Required to be Writable but Not Modified

```
/* program/src/instruction/buy.rs */
004 | pub fn process_buy_tokens(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
008 |     let [
010 |         pool_info,
011 |         currency_info,
021 |     ] = accounts else {
022 |         return Err(ProgramError::NotEnoughAccountKeys);
023 |     };
029 |     check_mut(pool_info)?;
030 |     check_mut(currency_info)?;
048 |     let pool = pool_info.as_account_mut::<LiquidityPool>(&flipcash_api::ID)?;
152 | }

/* program/src/instruction/sell.rs */
004 | pub fn process_sell_tokens(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
008 |     let [
010 |         pool_info,
011 |         currency_info,
021 |     ] = accounts else {
022 |         return Err(ProgramError::NotEnoughAccountKeys);
023 |     };
029 |     check_mut(pool_info)?;
030 |     check_mut(currency_info)?;
146 | }

/* program/src/instruction/pool.rs */
004 | pub fn process_initialize_pool(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
005 |     let raw_args = InitializePoolIx::try_from_bytes(data)?;
006 |     let args = raw_args.to_struct()?;
007 |
008 |     let [
010 |         currency_info,
021 |     ] = accounts else {
022 |         return Err(ProgramError::NotEnoughAccountKeys);
023 |     };
028 |     check_mut(currency_info)?;
082 |     let currency = currency_info.as_account_mut::<CurrencyConfig>(&flipcash_api::ID)?;
180 | }
```

Account Deserialized as Mutable Without Being Writable or Modified

```
/* program/src/instruction/metadata.rs */
004 | pub fn process_initialize_metadata(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
008 |     let [
010 |         currency_info,
018 |     ] = accounts else {
019 |         return Err(ProgramError::NotEnoughAccountKeys);
020 |     };
033 |     let currency = currency_info.as_account_mut::<CurrencyConfig>(&flipcash_api::ID)?;
105 | }
```

Accounts Modified Without a Writable Check

```
/* program/src/instruction/buy.rs */
004 | pub fn process_buy_tokens(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
008 |     let [
018 |         fee_target_info,
021 |     ] = accounts else {
022 |         return Err(ProgramError::NotEnoughAccountKeys);
023 |     };
          // @audit: fee_target_info will be written to in a CPI but is not checked as mutable.
135 |     if fee_amount_raw > 0 {
136 |         transfer_signed_with_bump(
137 |             target_vault_info,
138 |             target_vault_info,
139 |             fee_target_info,
140 |             token_program_info,
141 |             fee_amount_raw,
142 |             &[
143 |                 TREASURY,
144 |                 pool_info.key.as_ref(),
145 |                 target_mint_info.key.as_ref()
146 |             ],
147 |             pool.vault_a_bump,
148 |         )?;
149 |     }
152 | }

/* program/src/instruction/sell.rs */
004 | pub fn process_sell_tokens(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
008 |     let [
019 |         fee_base_info,
021 |     ] = accounts else {
022 |         return Err(ProgramError::NotEnoughAccountKeys);
023 |     };
          // @audit: fee_base_info will be written to in a CPI but is not checked as mutable.
129 |     if fee_amount_raw > 0 {
130 |         transfer_signed_with_bump(
131 |             base_vault_info,
132 |             base_vault_info,
133 |             fee_base_info,
134 |             token_program_info,
135 |             fee_amount_raw,
136 |             &[
137 |                 TREASURY,
138 |                 pool_info.key.as_ref(),
139 |                 base_mint_info.key.as_ref()
140 |             ],
141 |             pool.vault_b_bump,
```

```
142 |         )?;
143 |     }
146 | }
```

It is recommended to remove unnecessary `check_mut` calls and replace `as_account_mut` with `as_account` for read-only accounts. For accounts that are actually modified, a `check_mut` should be added.

Resolution

Fixed by commit `c52b931`.

FLIPCASH PROGRAM

[P2-L-02] User may lose funds on overpayment

Identified in commit `6c985fd`.

The program uses an exponential price curve, where the price gets higher as it approaches the total supply limit.

The curve used does not stop at the maximum supply, instead, it is designed to reach $1,000,000 at the maximum supply and continue to grow exponentially afterward.

This means if a user accidentally provides too many base tokens, the `curve.value_to_tokens` function can return an amount that is greater than the remaining supply of the target token.

```
/* api/src/curve.rs */
021 | /// Calculate token price at a given supply
022 | pub fn spot_price_at_supply(&self, current_supply: &UnsignedNumeric) -> Option<UnsignedNumeric> {
023 |     // R'(S) = a * b * e^(c * s)
024 |
025 |     let c_times_s = self.c.checked_mul(current_supply)?;
026 |     let exp = c_times_s.signed().exp()?;
027 |     self.a.checked_mul(&self.b)?.checked_mul(&exp)
028 | }

/* api/src/consts.rs */
016 | // Constants for the default curve from $0.01 to $1_000_000 over 21_000_000 tokens
017 | pub const CURVE_A: u128         = 11400_230149967394933471;
018 | pub const CURVE_B: u128         = 0_000000877175273521;
019 | pub const CURVE_C: u128         = CURVE_B;
```

In this situation, the `process_buy_tokens` function limits the user to receiving only the remaining supply of the target token. The excess base tokens are not refunded.

```
/* program/src/instruction/buy.rs */
004 | pub fn process_buy_tokens(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
080 |     let mut total_tokens = curve.value_to_tokens(&supply, &in_amount)
081 |         .ok_or(ProgramError::InvalidArgument)?;
082 |     if tokens_left.less_than(&total_tokens) {
083 |         total_tokens = tokens_left
084 |     }
087 |     let tokens_after_fee = total_tokens.checked_sub(&fee_amount)
088 |         .ok_or(ProgramError::InvalidArgument)?;
096 |     let tokens_after_fee_raw = from_numeric(tokens_after_fee.clone(), mint_a_decimals)?;
112 |
113 |     transfer(
114 |         buyer_info,
115 |         buyer_base_ata_info,
116 |         base_vault_info,
117 |         token_program_info,
118 |         args.in_amount,
```

```
119 |     )?;
120 |
121 |     transfer_signed_with_bump(
122 |         target_vault_info,
123 |         target_vault_info,
124 |         buyer_target_ata_info,
125 |         token_program_info,
126 |         tokens_after_fee_raw,
133 |     )?;
152 | }
```

But since there is a slippage check in the buy instruction, allowing users to set their desired minimum token output before executing a trade. This ensures they have a clear expectation of the outcome and are protected from receiving an unexpectedly low amount, so the impact of the issue is limited.

Resolution

Fixed by commit `88b063a`.

FLIPCASH PROGRAM

[P2-L-03] Missing decimal check on base_mint

Identified in commit `6c985fd`.

In the `buy` and `sell` instructions, the program calls `to_numeric` and `from_numeric` to convert the `base_mint` or `target_mint` amount between `UnsignedNumeric` and `u64`. These two functions require that the decimal be less than or equal to 18.

```
/* api/src/utils.rs */
077 | pub fn to_numeric(amount: u64, decimal_places: u8) -> Result<UnsignedNumeric, ProgramError> {
078 |     if decimal_places > 18 {
079 |         return Err(ProgramError::InvalidArgument);
080 |     }
093 | }

096 | pub fn from_numeric(value: UnsignedNumeric, decimal_places: u8) -> Result<u64, ProgramError> {
097 |     if decimal_places > 18 {
098 |         return Err(ProgramError::InvalidArgument);
099 |     }
112 | }
```

The `target_mint` is created by the program in `process_initialize_currency` with a constant decimal value of 10, which meets the requirement.

However, the `base_mint` is an arbitrary mint specified by the pool creator in `process_initialize_pool`. The number of decimals for this mint could be greater than 18, and `process_initialize_pool` is missing a check for this.

```
/* program/src/instruction/pool.rs */
004 | pub fn process_initialize_pool(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
008 |     let [
            // @audit: The decimal value of this mint account is not checked.
012 |         base_mint_info,
021 |     ] = accounts else {
022 |         return Err(ProgramError::NotEnoughAccountKeys);
023 |     };
180 | }
```

Currently, there is no instruction to modify a pool's configuration, and each pool and currency account has a one-to-one relationship.

That means if an incorrect configuration is set, the `buy` and `sell` functions will become completely unusable, and will also make the created currency and pool permanently unusable.

It is recommended to add a decimal check in `process_initialize_pool` to prevent a denial of service

caused by an administrator's incorrect configuration.

Resolution

Fixed by commit `7f4a2c1`.

FLIPCASH PROGRAM

[P2-I-01] Unnecessary bump passed in args

Identified in commit `6c985fd`.

When creating the accounts, the bump for each new PDA is passed in the `args`. However, this value is actually not arbitrary, but a fixed one.

All newly created PDAs are verified by the `check_uninitialized_pda` function. This function does not accept a custom bump and instead uses the default one. Therefore, passing the bump as a parameter is unnecessary.

We can directly call `Pubkey::find_program_address(seeds, program_id)` to get both the expected PDA address and the default bump. This would reduce unnecessary parameter passing.

```
/* program/src/instruction/currency.rs */
004 | pub fn process_initialize_currency(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
030 |     check_uninitialized_pda(
031 |         mint_info,
032 |         &[
033 |             MINT,
034 |             authority_info.key.as_ref(),
035 |             raw_args.name.as_ref(),
036 |             args.seed.as_ref()
037 |         ],
038 |         &flipcash_api::id()
039 |     )?;
040 |
041 |     check_uninitialized_pda(
042 |         currency_info,
043 |         &[ CURRENCY, mint_info.key.as_ref() ],
044 |         &flipcash_api::id()
045 |     )?;
085 |     currency.bump = args.bump;
086 |     currency.mint_bump = args.mint_bump;
089 | }

/* program/src/instruction/pool.rs */
004 | pub fn process_initialize_pool(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
064 |     check_uninitialized_pda(
065 |         pool_info,
066 |         &[ POOL, currency_info.key.as_ref() ],
067 |         &flipcash_api::id()
068 |     )?;
069 |
070 |     check_uninitialized_pda(
071 |         target_vault_info,
072 |         &[ TREASURY, pool_info.key.as_ref(), target_mint_info.key.as_ref() ],
073 |         &flipcash_api::id()
074 |     )?;
075 |
076 |     check_uninitialized_pda(
```

```
077 |         base_vault_info,
078 |         &[ TREASURY, pool_info.key.as_ref(), base_mint_info.key.as_ref() ],
079 |         &flipcash_api::id()
080 |     )?;
175 |     pool.bump = args.bump;
176 |     pool.vault_a_bump = args.vault_a_bump;
177 |     pool.vault_b_bump = args.vault_b_bump;
178 |
179 |     Ok(())
180 | }
```

Resolution

The team acknowledged this finding.

FLIPCASH PROGRAM

[P2-I-02] Misleading ATA variable names

Identified in commit `6c985fd`.

In `process_buy_tokens` and `process_sell_tokens`, several account variables end with `ata_info`. However, the program does not actually require these accounts to be Associated Token Accounts (ATAs). It only checks that they are token accounts owned by the correct user.

```
/* program/src/instruction/buy.rs */
004 | pub fn process_buy_tokens(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
008 |     let [
016 |         buyer_target_ata_info,
017 |         buyer_base_ata_info,
021 |     ] = accounts else {
022 |         return Err(ProgramError::NotEnoughAccountKeys);
023 |     };
040 |     buyer_target_ata_info.as_token_account()?
041 |         .assert(|t| t.owner().eq(buyer_info.key))?
042 |         .assert(|t| t.mint().eq(target_mint_info.key))?;
044 |     buyer_base_ata_info.as_token_account()?
045 |         .assert(|t| t.owner().eq(buyer_info.key))?
046 |         .assert(|t| t.mint().eq(base_mint_info.key))?;
152 | }

/* program/src/instruction/sell.rs */
004 | pub fn process_sell_tokens(accounts: &[AccountInfo], data: &[u8]) -> ProgramResult {
008 |     let [
016 |         seller_target_ata_info,
017 |         seller_base_ata_info,
021 |     ] = accounts else {
022 |         return Err(ProgramError::NotEnoughAccountKeys);
023 |     };
040 |     seller_target_ata_info.as_token_account()?
041 |         .assert(|t| t.owner().eq(seller_info.key))?
042 |         .assert(|t| t.mint().eq(target_mint_info.key))?;
044 |     seller_base_ata_info.as_token_account()?
045 |         .assert(|t| t.owner().eq(seller_info.key))?
046 |         .assert(|t| t.mint().eq(base_mint_info.key))?;
146 | }
```

While this does not pose a security risk, it could confuse users and make them believe that only ATAs are allowed. Consider either adding a check to ensure the accounts are ATAs or changing the variable names for clarity.

Resolution

Fixed by commit `317491b`.

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 Flipcash Inc. (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.