Razor DEX

Security Audit Report

April 24, 2024

Contents

1 Introduction

1.1 About Razor DEX

`Razor DEX` is a decentralized exchange built on `Uniswap V2`'s `x * y = k` constant product model. Powered by the `Move` Language, `Razor DEX` aims to provide users with a secure, transparent, and an unparalleled trading experience within the Movement ecosystem.

1.2 Source Code

The following source code was reviewed during the audit:

- https://github.com/razorlabsorg/razor-dex-contracts
- CommitID: bff3d37

And this is the final version representing all fixes implemented for the issues identified in the audit:

- https://github.com/razorlabsorg/razor-dex-contracts
- CommitID: 9be4160

2 | Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the `Razor DEX` protocol. Throughout this audit, we identified a total of 6 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	-	-	-	-
High	-	-	-	-
Medium	3	1	-	2
Low	3	-	-	3
Informational	-	-	-	-
Undetermined	-	-	-	-

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	Description
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and remediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3 Vulnerability Details

[M-1] Possible Underflow in m1dex::swap.move

Target	Category	IMPACT	LIKELIHOOD	STATUS
swap.move	Business Logic	Medium	Medium	Addressed

In the `swap` contract, the `mint()` function is used to mint `LP` tokens for liquidity providers. While examining its logic, we notice the current mint logic needs to be improved.

To elaborate, we show below the `mint()` routine. To prevent price manipulation, the first user adding liquidity needs to lock a minimum amount of liquidity into the pool. Therefore, the provided liquidity needs to be greater than the minimum to avoid the risk of underflow.

m1dex::swap::mint()

```
  public fun mint<X, Y>(
    coin_x: Coin<X>,
    coin_y: Coin<Y>
): Coin<LPCoin<X, Y>> acquires LiquidityPool, AdminData, Events {
    assert!(RazorPoolLibrary::compare<X, Y>(), ERR_PAIR_ORDER_ERROR);
    assert!(exists<LiquidityPool<X, Y>>(RESOURCE_ACCOUNT_ADDRESS),
        ERR_PAIR_NOT_EXIST);
    assert_not_paused();
    assert_lp_unlocked<X, Y>();
    let amount_x = coin::value(&coin_x);
    let amount_y = coin::value(&coin_y);
    // feeOn
    let fee_on = mint_fee_interval<X, Y>(lp, admin_data);
    coin::merge(&mut lp.coin_x_reserve, coin_x);
    coin::merge(&mut lp.coin_y_reserve, coin_y);
    let (balance_x, balance_y) = (coin::value(&lp.coin_x_reserve), coin::value(&
        lp.coin_y_reserve));

    let total_supply = RazorPoolLibrary::get_lpcoin_total_supply<LPCoin<X, Y>>()
        ;
    let liquidity;
    if (total_supply == 0) {
        liquidity = RazorPoolLibrary::sqrt(amount_x, amount_y) -
            MINIMUM_LIQUIDITY;
        mint_coin<X, Y>(&get_resource_account_signer(), MINIMUM_LIQUIDITY, &lp.
            lp_mint_cap);
    } else {
        // normal tx should never overflow
        let amount_1 = ((amount_x as u128) * total_supply / (reserve_x as u128)
            as u64);
        let amount_2 = ((amount_y as u128) * total_supply / (reserve_y as u128)
            as u64);
```

```
        liquidity = RazorPoolLibrary::min(amount_1, amount_2);
    };
    assert!(liquidity > 0, ERR_INSUFFICIENT_LIQUIDITY_MINT);
    let coins = coin::mint<LPCoin<X, Y>>(liquidity, &lp.lp_mint_cap);
    // update interval
    update_internal(lp, balance_x, balance_y, reserve_x, reserve_y);
    // feeOn
    if (fee_on) lp.k_last = (balance_x as u128) * (balance_y as u128);
    // event
    let events = borrow_global_mut<Events<X, Y>>(RESOURCE_ACCOUNT_ADDRESS);
    event::emit_event(&mut events.mint_event, MintEvent {
        amount_x,
        amount_y,
        liquidity,
    });
    coins
}
```

The same issue is also applicable for the `m2-dex::swap::mint()` function.

Remediation Add validation in above-mentioned functions.

[M-2] Potential Risks Associated with Centralization

Target	Category	IMPACT	LIKELIHOOD	STATUS
Multiple Contracts	Security	Medium	Medium	Acknowledged

In the `Razor` protocol, the existence of a privileged `owner` account introduces centralization risks, as it holds significant control and authority over critical operations governing the protocol. In the following, we show the representative functions potentially affected by the privileges associated with the privileged account.

Example Privileged Operations in `Razor`

```
  public entry fun pause(
    account: &signer
) acquires AdminData {
    assert_not_paused();
    let admin_data = borrow_global_mut<AdminData>(RESOURCE_ACCOUNT_ADDRESS);
    assert!(signer::address_of(account) == admin_data.admin_address,
        ERR_FORBIDDEN);
    admin_data.is_pause = true;
}
 public entry fun set_swap_fee(
    account: &signer,
    swap_fee: u64
```

```
111 ) acquires AdminData {
112     let admin_data = borrow_global_mut<AdminData>(RESOURCE_ACCOUNT_ADDRESS);
113     assert!(signer::address_of(account) == admin_data.admin_address,
            ERR_FORBIDDEN);
114     assert!(swap_fee <= 1000, ERR_FORBIDDEN);
115     admin_data.swap_fee = swap_fee;
116 }
```

Remediation To mitigate the identified issue, it is recommended to introduce multi-sig mechanism to undertake the role of the privileged account. Moreover, it is advisable to implement timelocks to govern all modifications to the privileged operations.

Response By Team This issue has been confirmed by the team and the multi-sig mechanism will be used to mitigate it.

[M-3] Lack of Resource Account Check in m1dex::swap.move

Target	Category	IMPACT	LIKELIHOOD	STATUS
swap.move	Business Logic	Medium	Medium	Addressed

In `Aptos`, a resource account is an autonomous account without a corresponding private key used by developers to store resources or publish modules on-chain. In the `Razor` protocol, clearly the `swap` module is initialized using an existing resource account, likely denoted as `RESOURCE_ACCOUNT_ADDRESS`. Hence, it's essential to verify in the `init_module()` function if the initializing resource account is `RESOURCE_ACCOUNT_ADDRESS`.

m1dex::swap::init_module()

```
160   fun init_module(admin: &signer) {
161     // init admin data
162     let signer_cap = resource_account::retrieve_resource_account_cap(admin, DEV)
          ;
163     let resource_account = account::create_signer_with_capability(&signer_cap);
164     move_to(&resource_account, AdminData {
165         signer_cap,
166         dao_fee_to: DEPLOYER_ADDRESS,
167         admin_address: DEPLOYER_ADDRESS,
168         dao_fee: 5,          // 1/6 to dao fee
169         swap_fee: 30,        // 0.3%
170         dao_fee_on: true,  // default true
171         is_pause: false,     // default false
172     });
173     // init pair info
174     move_to(&resource_account, PairInfo {
```

```
175         pair_list: vector::empty(),
176     });
177 }
```

Remediation Add resource account validation in `init_module()` function.

[L-1] Revisited Swap Logic in m2dex::swap.move

Target	Category	IMPACT	LIKELIHOOD	STATUS
swap.move	Business Logic	Low	Low	Addressed

In the `Razor` protocol, the function `swap_balance_for_balance()` accepts balance of two coin types as parameters that are used to perform a swap between the two coins. Hence, the resulting balance of the swap-out coin type should be zero, while the other should not be zero. Therefore, when `amount_x_in` is zero, `amount_y_in` should be non-zero (line 209).

m2dex::swap.move

```
200   public fun swap_balance_for_balance<X, Y>(
201     lps: &mut LiquidityPools,
202     clock: &Clock,
203     coins_x_in: Balance<X>,
204     coins_y_in: Balance<Y>,
205 ): (Balance<X>, Balance<Y>) {
206     let (pool, admin_data) = get_pool<X, Y>(lps);
207     let amount_x_in = balance::value(&coins_x_in);
208     let amount_y_in = balance::value(&coins_y_in);
209     assert!((amount_x_in > 0 && amount_y_in == 0)  (amount_x_in == 0
          amount_x_in > 0), ERR_INPUT_VALUE);
210     if (amount_x_in > 0) {
211         let (reserve_in, reserve_out) = get_reserves_size<X, Y>(pool);
212         let amount_out = get_amount_out(amount_x_in, reserve_in, reserve_out,
              admin_data.swap_fee);
213         swap<X, Y>(lps, clock, coins_x_in, 0, coins_y_in, amount_out)
214     } else {
215         let (reserve_out, reserve_in) = get_reserves_size<X, Y>(pool);
216         let amount_out = get_amount_out(amount_y_in, reserve_in, reserve_out,
              admin_data.swap_fee);
217         swap<X, Y>(lps, clock, coins_x_in, amount_out, coins_y_in, 0)
218     }
219 }
```

Remediation Add validation to make sure when `amount_x_in` is zero, `amount_y_in` should be non-zero value.

[L-2] Inconsist Error Code in m1dex::swap.move

Target	Category	IMPACT	LIKELIHOOD	STATUS
swap.move	Business Logic	Low	Low	Addressed

In `Razor` protocol, the `set_swap_fee()` function is used by the admin to modify the swap fee. In the current implementation, an error with the code `ERR_FORBIDDEN` is triggered when the swap fee exceeds 1000 (line 580). However, we notice that the explanation in the comment for this error code is unrelated to the value of the swap fee. Therefore, it is suggested to replace the corresponding error code with a more appropriate one.

m1dex::swap::set_swap_fee()

```
574   public entry fun set_swap_fee(
575     account: &signer,
576     swap_fee: u64
577 ) acquires AdminData {
578     let admin_data = borrow_global_mut<AdminData>(RESOURCE_ACCOUNT_ADDRESS);
579     assert!(signer::address_of(account) == admin_data.admin_address,
            ERR_FORBIDDEN);
580     assert!(swap_fee <= 1000, ERR_FORBIDDEN);
581     admin_data.swap_fee = swap_fee;
582 }
```

Remediation Modify the error code in `set_swap_fee()` function.

[L-3] Lack of Admin Check in m1coins::coins.move

Target	Category	IMPACT	LIKELIHOOD	STATUS
coins.move	Business Logic	N/A	N/A	Addressed

The `initialize()` function is a public function used to initialize coins with their respective names, symbols, decimals, and other properties. Hence, this initialization of coins should be executed by the admin rather than other account. However, when we examining its logic, we notice that the initialization logic needs improvement to ensure that the account calling this function is the admin.

m1coins::coins::initialize()

```
238   public entry fun initialize(admin: &signer) {
239     // Initialize BTC
240     let (btc_b, btc_f, btc_m) =
241         coin::initialize<BTC>(admin,
```

```
            utf8(b"Bitcoin"), utf8(b"BTC"), 8, true);
    // Initialize USDT
    let (usdt_b, usdt_f, usdt_m) =
        coin::initialize<USDT>(admin,
            utf8(b"Tether"), utf8(b"USDT"), 8, true);
    // Initialize USDC
    let (usdc_b, usdc_f, usdc_m) =
        coin::initialize<USDC>(admin,
            utf8(b"Circle USD"), utf8(b"USDC"), 8, true);
    // Initialize ETH
    let (eth_b, eth_f, eth_m) =
        coin::initialize<ETH>(admin,
            utf8(b"Ether"), utf8(b"ETH"), 8, true);
    // Initialize SOL
    let (sol_b, sol_f, sol_m) =
        coin::initialize<SOL>(admin,
            utf8(b"Solana"), utf8(b"SOL"), 8, true);
    // Initialize BNB
    let (bnb_b, bnb_f, bnb_m) =
        coin::initialize<BNB>(admin,
            utf8(b"Binance Coin"), utf8(b"BNB"), 8, true); // daisy decimal??
    // Initialize RAZOR
    let (razor_b, razor_f, razor_m) = coin::initialize<RAZOR>(
        admin,
        utf8(b"Razor Token"),
        utf8(b"RAZOR"),
        8,
        true
    );

    // Store the capabilities for each coin type under the admin account.
    move_to(admin, Caps<BTC> { mint: btc_m, freeze: btc_f, burn: btc_b });
    move_to(admin, Caps<USDT> { mint: usdt_m, freeze: usdt_f, burn: usdt_b });
    move_to(admin, Caps<USDC> { mint: usdc_m, freeze: usdc_f, burn: usdc_b });
    move_to(admin, Caps<ETH> { mint: eth_m, freeze: eth_f, burn: eth_b });
    move_to(admin, Caps<SOL> { mint: sol_m, freeze: sol_f, burn: sol_b });
    move_to(admin, Caps<BNB> { mint: bnb_m, freeze: bnb_f, burn: bnb_b });
    move_to(admin, Caps<RAZOR> { mint: razor_m, freeze: razor_f, burn: razor_b
        });

    // Register all coins for the admin account.
    register_coins_all(admin);
}
```

Remediation Add admin check in `initialize()` function.

4 Conclusion

`Razor DEX` is a decentralized exchange built on `Uniswap V2`'s $x * y = k$ constant product model using `Move` Language. The current code base is well structured and neatly organized. Those identified issues were promptly confirmed and fixed.

5 Appendix

5.1 About AstraSec

AstraSec is a blockchain security company that serves to provide high-quality auditing services for blockchain-based protocols. With a team of blockchain specialists, AstraSec maintains a strong commitment to excellence and client satisfaction. The audit team members have extensive audit experience for various famous DeFi projects. AstraSec's comprehensive approach and deep blockchain understanding make it a trusted partner for the clients.

5.2 Disclaimer

The information provided in this audit report is for reference only and does not constitute any legal, financial, or investment advice. Any views, suggestions, or conclusions in the audit report are based on the limited information and conditions obtained during the audit process and may be subject to unknown risks and uncertainties. While we make every effort to ensure the accuracy and completeness of the audit report, we are not responsible for any errors or omissions in the report.

We recommend users to carefully consider the information in the audit report based on their own independent judgment and professional advice before making any decisions. We are not responsible for the consequences of the use of the audit report, including but not limited to any losses or damages resulting from reliance on the audit report.

This audit report is for reference only and should not be considered a substitute for legal documents or contracts.

5.3 Contact

Phone	+86 176 2267 4194
Email	contact@astrasec.ai
Twitter	https://twitter.com/AstraSecAI