Dezswap DEX
Security Audit

ChainLight

revision 1.1

Prepared for
DEZSWAP

Prepared by
ChainLight
(Theori)

March 2nd, 2023

Table of Contents

Executive Summary

Starting on October 17th, ChainLight of Theori and DreamAcademy assessed the smart contracts for Dezswap code base. We focused on identifying issues that result in a loss of funds and any lack of security mitigations which protect the end-users by comparison analysis with UniswapV2.

We evaluated the correctness, security, and code maturity of the Dezswap code base. We wrote testcases with high code coverage and checked the correctness of the contracts. Additionally, we have conducted comparison analysis with UniswapV2. This process helped us recognize security issues efficiently by pointing out regions within the code that should be looked for especially carefully.

We have discovered a total of 6 security relevant issues, and 3 of them were evaluated to be of high impact. However, all of them are exploitable only under limited circumstances, and thus we believe the probability of observing an in-the-wild attack is minute. We also made recommendations for improving code maturity improvement.

Contact Info

ChainLight, chainlight@theori.io
https://chainlight.io/

ChainLight's mission is to make the Web3 ecosystem more secure and enable our customers and users to grow without fear of security threats. We proactively counteract bad actors that steal funds from Web3 projects and protect users by finding vulnerabilities in Web3 applications quickly and efficiently.

We offer manual security auditing services from experienced auditors as well as automated analysis tools that enable customers to discover and remediate critical security issues ahead of time. Customers can integrate our product into their CI infrastructure to continuously scan for security vulnerabilities.

We will develop and employ state-of-the-art program analysis techniques as well as financial modeling techniques to ensure security and future solvency of Web3 projects. We seek to share our discovery and techniques with the wider open-source community.

DreamAcademy
https://dream-academy.io/

DreamAcademy is an educational program operated by ChainLight of Theori and HanWhaLife. Its mission is to nurture Web3 security talents through a well-organized curriculum and real-world, hands-on experiences. The whole process was done under the guidance of ChainLight, and the following people conducted this assessment; Woosun Song (Lead), Woosung Jung, Eunyeong Ahn, and Jihyun Yoo.

Scope

We reviewed the smart contracts for DELIGHT LABS's Dezswap Protocol.
The code was retrieved on October 17th, 2022, from:

- Dezswap
    - commit 9929b079e3172650cf4b0f9a64732411c8c402f6

Code revision

The ability to `provide_liquidity` execute with `create_pair` has revision. The revision code review was completed, and it was confirmed that there was no security problem.

- https://github.com/dezswap/dezswap-contracts/tree/feature/create-with-provide commit 746e441fcfb03feace53dd8aebf6b0ae4823501f

Overview

The assessment focused on the following items.

Correctness

Check that the contracts strictly adhere to the specifications. We do not perform any sort of formal verification; we inductively reason the correctness via thorough and extensive tests.

Security

Devise exploit scenarios that may result in the backend contract / end user / liquidity provider to suffer loss. Also, DoS attacks and undesired locking of assets are also put into consideration.

Adherence to Precedent Protocols

Similarities with precedent DEX protocols such as UniswapV2 is a desirable property for multiple reasons. First, it eases the entry of users who have experience with those protocols. Second, such protocols have been thoroughly audited over a relatively long time, making them trustworthy references.

Code Maturity

We looked for code that can be refactored in terms of: gas consumption optimization / syntactic coherence / readability improvement.

Comparison Analysis Against UniswapV2

We chose UniswapV2 as the pivot of evaluation for the following reasons:

- The official documentation of Dezswap mentions Uniswap as a source of inspiration.
- UniswapV2 has been audited by multiple parties.
- Both contracts have the components factory/router/pair and their end-to-end functionalities are equivalent.
- Although there exist next-generation protocols such as UniswapV3 or Curve Finance, comparing them with Dezswap failed to yield meaningful conclusions. For example, UniswapV3 implements concentrated liquidity, which does not exist at all in Dezswap.

Analysis

	UniswapV2	Dezswap
Asset	Must use wrapped tokens (ex. WETH) to exchange on-chain currencies.	Can handle on-chain currencies directly without any sort of wrapping.
Liquidity Provision	Router has API for liquidity provision with slippage mitigations.	`withdraw_liquidity` is susceptible to sandwich attacks
	Pair has API for liquidity provision, but without any slippage mitigations.	Pair has API for liquidity provision with slippage mitigations.
	Pair inherits the IERC20 interface in order to act as the liquidity token.	The pair and its liquidity token are located at separate addresses.
	Fees are taken upon liquidity provision.	No fees taken upon liquidity provision.
Swap	Router has API for swap, with slippage mitigations. Slippage mitigation methods are identical: swap transaction(message) reverts if minimum output amount is unfulfilled.	
	Optimistic swap: Provides the output asset(s) prior to confirming the input assets are given.	Pessimistic swap: Does not provide the output asset(s) until it is confirmed that the input assets are given.
	Fees are taken from both the assets of the pool.	Fees are taken only from the ask asset.
Liquidity Withdrawal	Router has API for liquidity withdrawal, with slippage mitigations.	Router has no API for liquidity withdrawal.
	A certain amount (`MINIMUM_LIQUIDITY`) of liquidity tokens are permanently locked.	* It is possible for LPs to remove all liquidity.
	Fees are taken upon liquidity withdrawal.	No fees taken upon liquidity withdrawal.

* As a result of the comparison, it is recommended to add `MINIMUM_LIQUIDITY` because `MINIMUM_LIQUIDITY` does not exist, which can lead to a DoS attack by a malicious attacker when liquidity does not exist.

Findings

These are the potential issues that may have correctness and/or security impacts.

Summary

#	ID	Title	Severity
1	THE-DEZSWAP-001	provide_liquidity is susceptible to sandwich attacks	**Fixed** (Critical)
2	THE-DEZSWAP-002	withdraw_liquidity is susceptible to sandwich attacks	**Fixed** (High)
3	THE-DEZSWAP-003	Pool size can become zero via a huge swap	**Fixed** (High)
4	THE-DEZSWAP-004	Router should adopt the deadline argument	**Fixed** (Medium)
5	THE-DEZSWAP-005	Commission is rounded down in compute_swap	**Fixed** (Low)
6	THE-DEZSWAP-006	Insufficient integer overflow handling in provide_liquidity	**Fixed** (Informational)

Issue #1: provide_liquidity is susceptible to sandwich attacks

ID	Summary	Severity
THE-DEZSWAP-001	The `provide_liquidity` function lacks a proper slippage tolerance mitigation, exposing it to sandwich attacks.	**Fixed** (Critical)

Root Cause

Slippage loss mitigation in `provide_liquidity` is necessary because an end-user will provide two assets according to the current pool ratio, which can differ from the pool ratio at the time of transaction execution. This difference results in the overprovision of one of the assets.

In UniswapV2, both the pair and the router implement APIs for liquidity provision. The API in the pair is `mint()`, and has no slippage mitigations. On the other hand, Router02 provides a safe API for providing liquidity, called `addLiquidity()`, which mitigates slippage loss by calculating the amounts of asset0 and asset1 that maximizes the returned LP token amount and transferring only those amounts. This flow effectively eliminates all loss due to slippage.

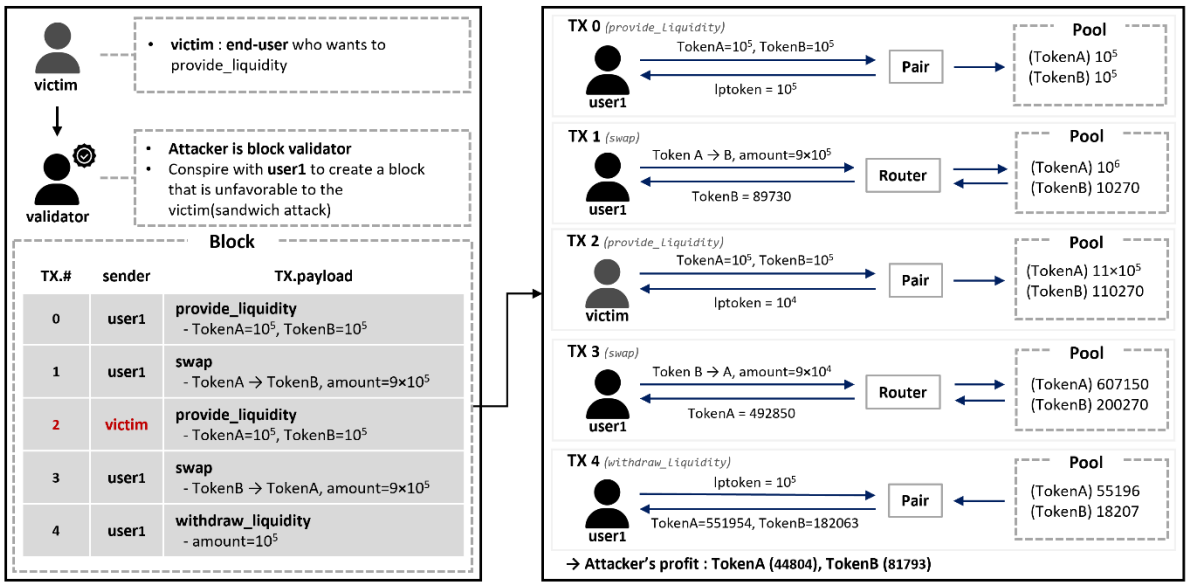
Unlike UniswapV2's implementation, Dezswap's pair has slippage loss mitigations. Dezswap mitigates slippage loss by reverting the transaction if the pool ratio and the deposit ratio differs more than a certain threshold. The threshold is calculated by the formula `1 - slippage_tolerance`, where `slippage_tolerance` is an optional argument provided by the contract caller.

However, this implementation is problematic because liquidity provision is not a process where slippage loss is inevitable, as the implementation of UniswapV2's `addLiquidity()` API is completely loss-free. Dezswap forces users to tolerate some amount of loss even if it is completely avoidable. If we assume an attacker conspiring with a block validator, it even becomes possible to maximize other users' loss continuously and profit from this loss.

Threat Model and Exploit Scenario

In order for the attack to take place, the following conditions must be satisfied: First, the attacker must be a block validator or a third-party colluding with a block validator. The perks of being a block validator is that it is possible to reorder transactions within a single block. Second, the attacker must possess capital that can drastically change the exchange rate of the pair.

We present a toy scenario that describes such an attack. In this scenario, three parties are involved: an innocent LP, a malicious LP, and a malicious block validator that conspires with the malicious LP. The pair holds two assets, token A and token B, whose pool sizes have a ratio of 1:1. The attack begins as the innocent LP issues a transaction containing a `provide_liquidity` message. Because the innocent LP sees a pool ratio of 1:1, it will deposit the same amount of token A and token B to the pair.



(1) The innocent LP sends a `provide_liquidity` transaction that deposits 10^5 token A and 10^5 token B. For simplicity, we assumed that `slippage_tolerance` is set to `None`. As the malicious block validator receives the innocent LP's transaction, it constructs 4 additional transactions and places them in an appropriate order within the block so that the following sequence of events take place.
(2) The malicious user deposits 10^5 token A and 10^5 token B and becomes a LP. It receives 10^5 LP tokens, which can be redeemed to 10^5 token A and 10^5 token B.
(3) Through the second transaction, the malicious LP swaps a large amount of token A into token B. This results in the pool's 1:1 balance to shift to approximately 10:1.
(4) The innocent LP's `provide_liquidity` transaction is executed. The current pool ratio is approximately 10:1, which results in the overprovision of Token B.
(5) Afterwards, the malicious LP restores the pool ratio by swapping large amounts of Token B to Token A.
(6) The malicious LP removes liquidity and redeems Token A and Token B. It receives more tokens than 10^5, because of the overprovision in 3. Thus, it is possible to say that the malicious LP took advantage of the innocent LP.

Fixes and Recommendations

(1) Create a safe API for liquidity provision in the router. The slippage loss mitigation method must be identical to UniswapV2's router: only transferring assets that contribute to according to the pool ratio, and return the rest.
(2) Remove all safety checks (The `slippage_tolerance` argument and all code dependent on it) from Dezswap pair. This is done to reduce the gap between Dezswap pair and UniswapV2's pair as well as reducing gas consumption caused by redundant checks when using the router.

Fix

The issue was fixed in commit https://github.com/dezswap/dezswap-contracts/commit/82131ac4acc3a48d159f74502280d38db71ae733

Issue #2: withdraw_liquidity is susceptible to sandwich attacks

ID	Summary	Severity
THE-DEZSWAP-002	The `withdraw_liquidity` function lacks the proper slippage tolerance mitigation, exposing it to sandwich attacks. The market conditions could limit the impact.	**Fixed** (High)

Root Cause

UniswapV2 Router02's `removeLiquidity` API takes the `amountMin` argument, which is compared with the yields of each assets redeemed and reverts the transaction if `amountMin` is not fulfilled.

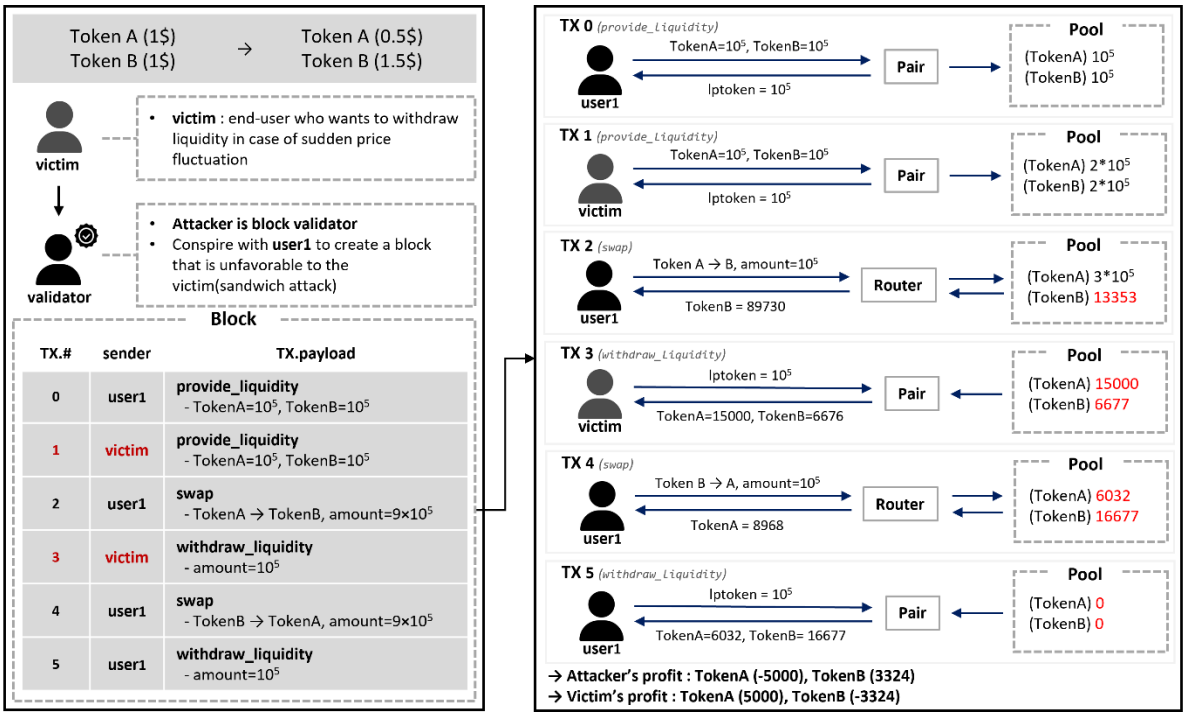
In Dezswap, the `withdraw_liquidity` API exists for removing liquidity. However, there are two differences with UniswapV2's API: First, the API for withdrawing liquidity only exists in the pair, and the router does not provide an API for withdrawing liquidity. Second, there is no argument analogous to `amountMin` and no slippage loss mitigations are performed in `withdraw_liquidity`. The second poses a nontrivial security implication.

If an innocent LP tries to withdraw liquidity under inclement circumstances, he/she may face loss. Such circumstances are described in the next section, Threat Model and Exploit Scenario.

Threat Model and Exploit Scenario

The condition for this attack to be successful is equivalent to the conditions described in 5.2.2. However, an additional condition must also be present in this case: The difference in value between the two tokens in the pair must be amplified within a short period of time.

The setting for our scenario is as follows: The pair consists of pools of Token A and Token B. Initially, Token A and Token B have the same value: both tokens can be bought with 1$ on commodity exchanges. The following sequence of events lead to an attacker making profit from an innocent user's loss:



(1) User1 is a malicious user colluding with a block validator. User1 provides liquidity to the pair by depositing 10^5 Token A and 10^5 Token B, and gets 10^5 LP token in return.

(2) The victim also provides liquidity to the pair by depositing 10^5 Token A, 10^5 Token B, and receives 10^5 LP token in return. Currently, the same amount of Token A and Token B exists in the pool.

(3) Due to external influences, the price of Token A drops to 0.5$ and the price of Token B increases to 1.5$.

(4) Due to this price fluctuation, liquidity providers will be motivated to withdraw liquidity. Because the victim is currently a liquidity provider, he/she will also attempt to withdraw liquidity as soon as possible, and thus issues a transaction holding the *withdraw_liquidity* message. The victim expects to redeem the same amount of Token A and Token B because at the point of issuing, the amount of both assets held by the pool are same.

(5) The malicious validator receives victim's transaction, and instead of executing it right away, it reorders the transaction to make profit. First, a swap transaction that swaps a large amount of Token A into Token B. This results in the pool having more Token A than Token B. Afterwards, the victim's withdraw liquidity transaction is executed. Then, user1 restores the balance within the pool and redeems its LP tokens.

(5) Until (2), the total asset value held by user1 and victim were equal, because they both held the same amount of LP tokens. However, user1 received more Token B than Token A, which resulted in a loss for the victim and a win for user1. We can see this as a form of 'forced' impermanent loss for the victim.

Fixes and Recommendations

(1) There is no API for liquidity withdrawal on Dezswap's router. To reduce the gap between UniswapV2 and Dezswap, we recommend implementing one.
(2) As in UniswapV2 Router02's `removeLiquidity` API, slippage loss mitigations must be implemented. We recommend implementing slippage loss mitigation in the same fashion. (Taking the `amountMin` argument)

Fix

The issue was fixed in commit https://github.com/dezswap/dezswap-contracts/commit/99a92a17b88e13e9f63afcb764d2d1870091e354

Issue #3: Pool can be emptied via a huge swap

ID	Summary	Severity
THE-DEZSWAP-003	Truncated division breaks the CPMM invariant leading to the free swap.	**Fixed** (High)

Root Cause

During swap, the value *return_amount* is computed prior to performing actual asset movement. *return_amount* is the amount of ask asset that is given to the user excluding commission. If *return_amount* is equal to the ask pool size, the ask pool size becomes 0 which breaks the CPMM invariant. Theoretically, it is impossible to make the ask pool size to become 0 as x and y cannot become 0 in the equation $xy = k > 0$. However, the following code makes it is possible to make x or y become 0 as integer divisions result in truncations.

```
    // offer => ask
    // ask_amount = (ask_pool - cp / (offer_pool + offer_amount)) * (1 -
commission_rate)
    let cp: Uint256 = offer_pool * ask_pool;
    let return_amount: Uint256 = (Decimal256::from_uint256(ask_pool)
        - Decimal256::from_ratio(cp, offer_pool + offer_amount))
        * Uint256::one();
```

It substitutes from the *ask_pool* the amount of ask assets that should remain after the swap. In the world of real numbers, the second operand of the substitution can never become zero, as division of two positive real numbers can never become zero. However, in the world of *Uint256*, as the remainder of division is discarded, it is possible for the second operand to become zero.

Threat Model and Exploit Scenario

In order to execute a swap that causes the `return_amount` to become 0, two conditions must be satisfied: first, `offer_pool + offer_amount` must be substantially larger than `offer_pool * ask_pool`. Second, `return_amount` must be small so that computed commission is rounded down to zero. It is nearly impossible to satisfy both conditions at once. Thus, the attack must be performed in two stages. The first stage requires shrinking the ask pool size to a small but nonzero value. The second stage performs a small swap that empties the ask pool.

The feasibility of the first stage depends on the initial pool size. The amount of offer asset required for this stage is linearly proportional to the size of the ask pool. Thus, in most cases the attack will be infeasible as it requires a substantial amount of capital. However, under vulnerable settings such as a small liquidity pool or a large difference between asset decimals, there exists a possibility.

The impact of this attack is that it renders the target pool dysfunctional. Once a pool size becomes 0, `provide_liquidity` and `swap` functionalities are inaccessible. In this situation, the owner of the pair can make one of two choices, each of which results in the same amount of loss.

The first choice is to leave the pair dysfunctional and do nothing. Then, it becomes possible for an attacker to drain all funds from the pair. It can be done by first pushing a small amount of funds to make the pool nonzero again, and make swaps in the reverse direction. For example, let's assume that a pair has 0 Token A and 1000 Token B. Then, an attacker sends 1 Token A to the pair to make it have 1 Token A and 1000 Token B. Afterwards, the attacker swaps 1 Token A and gets 500 Token B in return, which is approximately half of the pair's TVL. Repeating this process will drain the pair.

The second choice is to push an adequate number of assets to restore the balance between the pools. However, this results in an identical cost as in the first case. Let's assume the same situation where a pair has 0 Token A and 1000 Token B, and the value ratio of Token A and Token B in another DEX is 1:1. Then, to restore balance the pair owner must push 1000 Token A, which has the same value as 1000 Token B. In other words, to recover a pair

from such conditions, it requires funds commensurate to the current TVL of the pair. Thus, restoring the pair is as costly as abandoning it.

Fixes and Recommendations

We recommend modifying the method of computing *return_amount* in *compute_swap*. Although the fundamental equation is equivalent with the original code, its integer behavior is different.

```
     // offer => ask
     // ask_amount = (ask_pool - cp / (offer_pool + offer_amount)) * (1 -
commission_rate)
     -    let cp: Uint256 = offer_pool * ask_pool;
     -    let return_amount: Uint256 = (Decimal256::from_uint256(ask_pool)
     -        - Decimal256::from_ratio(cp, offer_pool + offer_amount))
     -        * Uint256::one();
     +    let return_amount: Uint256 = (ask_pool * offer_amount) / (offer_pool +
offer_amount);
```

In the original method, it is possible for the second operand of substitution to become zero which results in a 'round-up' like behavior for the entire expression. However, for the proposed method, *return_amount* can never be *ask_amount* unless *offer_pool* is zero due to a 'round-down' like behavior for the entire expression. In other words, the equation is hardened by reordering the necessary computations.

Fix

The issue was fixed in commit https://github.com/dezswap/dezswap-contracts/pull/8/commits/219243e1c1458dbfc4e17cfcfbee970ec4bc7f28

Issue #4: Router should adopt the deadline argument

ID	Summary	Severity
THE-DEZSWAP-004	Lack of deadline parameter allows the block producer to arbitrarily delay the victim user's transaction.	**Fixed** (Medium)

Root Cause

If the execution of an end-user's transaction is delayed for a long period of time, it can result in undesirable effects such as loss of funds. Thus, UniswapV2's router prevents a transaction from being delayed to a certain extent by taking an argument called `deadline`. In contrast, Dezswap does not take any safety measures similar to this.

Threat Model and Exploit Scenario

An example of a delayed contract resulting in a loss of funds is as follows. Due to external influences, the values of assets within a pair can fluctuate. If a liquidity provider's request to withdraw liquidity is delayed, the pair may transition into a state that is not in favor of the liquidity provider. Such delays may be amplified by malicious block validators. Such block validators will try to convert the loss of innocent users into their profit, which is discussed extensively in Issue#2 and Issue#3.

Fixes and Recommendations

A fix to this issue would be to implement the `deadline` argument and relevant checks. This would also result in reducing the gap between UniswapV2 and Dezswap. An alternative solution would be to use the timeout block height field in the transaction. This method is superior when compared to the former in terms of gas consumption. However, it has two downsides: First, the timeout block height is a per-transaction field. Thus, it would be insufficient to a user who wishes to include multiple messages with different timeouts within a single transaction. Second, the concept of block numbers is less intuitive than time to an end-user who is not familiar of blockchain internals.

Fix

The issue was fixed in commit https://github.com/dezswap/dezswap-contracts/commit/6e9eaf3f2e83b29620617d14c40be0130b87784b

Issue #5: Commission is rounded down in compute_swap

ID	Summary	Severity
THE-DEZSWAP-005	Truncated division in swap commission calculation leads to minuscule loss for the liquidity providers.	**Fixed** (Low)

Root Cause

To overcome the absence floating point data types in WASM, the contract uses fixed point decimals. `COMMISION_RATE` is fixed to 0.3% in Dezswap pair, and thus when calculating `commission_amount` in the `compute_swap` function, `return_amount` is multiplied by 3 and 1000 to replace the decimal operation. If the expression `return_amount * 3` has a nonzero remainder (when divided by 1000), truncation occurs, resulting in a small loss of value less than 1 for the pair. This results in a small profit for the swap user.

Threat Model and Exploit Scenario

The amount of loss accrued per transaction is timid and is guaranteed to be less than 1. Thus, we have evaluated the severity of this issue as low. One may claim that attackers can elide commission for the entire swap by splitting the entire swap amount to multiple small amounts. However, although this causes a loss to the pair, it would not benefit the attacker due to increased gas price, making it an unrealistic exploit scenario.

Fixes and Recommendations

When computing `commission_amount`, it must be increased by 1 if a nonzero remainder is left during division. This is equivalent to rounding-up the commission amount, whereas the current implementation computes in a round-down fashion.

Fix

The issue was fixed in commit https://github.com/dezswap/dezswap-contracts/commit/fd5574d65542fcc9c2b3360ac92e7f8bc88c81d4

Issue #6: Insufficient integer overflow handling in provide_liquidity

ID	Summary	Severity
THE-DEZSWAP-006	The *provide_liquidity* function does not handle the integer overflow properly.	**Fixed** (Informational)

Root Cause

In Dezswap pair's *provide_liquidity* API, during the very first liquidity provision, an integer overflow can occur if the product of the two assets' amounts exceeds $2^{256} - 1$.

```
let share = if total_share == Uint128::zero() {
    // Initial share = collateral amount
    Uint128::from((deposits[0].u128() * deposits[1].u128()).integer_sqrt())
    // integer over flow
}
```

Threat Model and Exploit Scenario

This problem is difficult to consider as a security issue because an asset flow with such magnitudes is unrealistic. Also, a liquidity provider can avoid the panic by splitting the assets into smaller units that does not result in overflows. However, we recommend fixing this issue nonetheless in consideration of programmatic correctness and exceptional situations (such as the case where the amount of tokens minted is abnormally high due to price inflations).

Fixes and Recommendations

```
fn compute_swap(
    offer_pool: Uint128,
    ask_pool: Uint128,
    offer_amount: Uint128,
) -> (Uint128, Uint128, Uint128) {
    let offer_pool: Uint256 = Uint256::from(offer_pool);
    let ask_pool: Uint256 = ask_pool.into();
    let offer_amount: Uint256 = offer_amount.into();
```

```
    let commission_rate = Decimal256::from_str(COMMISSION_RATE).unwrap();

    let cp: Uint256 = offer_pool * ask_pool;
}
```

In the *compute_swap* function of the Dezswap pair, integer overflow handling is implemented adequately. In order to prevent an integer overflow from occurring in the expression *offer_pool * ask_pool*, the operands are casted to *Uint256* prior to calculation.

It is possible to prevent integer overflows in the same manner. However, because *integer_sqrt* is only implemented for primitive integer types, it must be newly implemented for *Uint256*. The following is an example of a working implementation. However, safety validation and gas cost optimization are not performed, so it is recommended to exercise extreme caution upon using it.

```
fn integer_sqrt_for_uint256(num: Uint256) -> Uint128 {
    use std::ops::Shl;

    // Compute bit, the largest power of 4 <= n
    let max_shift: u32 = 255;
    let one: Uint256 = Uint128::new(1).into();
    let two: Uint256 = Uint128::new(2).into();

    // Compute leadning_zeros by first computing leading zeros of top 128 bits and
then the low 128 bits
    let mut leading_zeros;
    let modulus: Uint256 = two.pow(128);
    let top: Uint128 = num.checked_div(modulus).unwrap().try_into().unwrap();
    let low: Uint128 = num.checked_rem(modulus).unwrap().try_into().unwrap();
    leading_zeros = top.u128().leading_zeros();
    if leading_zeros == 128 {
        leading_zeros += low.u128().leading_zeros();
    }
    let shift: u32 = (max_shift - leading_zeros) & !1;

    let mut bit = one.shl(shift);

    // Algorithm based on the implementation in:
    //
https://en.wikipedia.org/wiki/Methods_of_computing_square_roots#Binary_numeral_system_
(base_2)
```

```
    // Note that result/bit are logically unsigned (even if T is signed).
    let mut n = num;
    let mut result = Uint256::zero();
    while bit != Uint256::zero() {
        if n >= (result + bit) {
            n = n - (result + bit);
            result = result.shr(1) + bit;
        } else {
            result = result.shr(1);
        }
        bit = bit.shr(2);
    }
    let result: Uint128 = result.try_into().unwrap();
    result
}
```

Fix

The issue was fixed in commit https://github.com/dezswap/dezswap-contracts/commit/ad54614d219fafdc5c74a019a46bd4c4ef70d545

Recommendations

These are the recommendations to improve the code maturity for better readability, optimization, and security. They do not impose any immediate security impacts.

Summary

#	Title	Type	Importance
1	Typo in assert_minium_receive	code maturity	Minor
2	Redundant usage of Decimal::from_str in compute_swap	optimization	Minor
3	Remove dead code	optimization	Minor

Recommendation #1: Typo in assert_minium_receive

The *assert_minium_receive* function implemented in the router takes *minium_receive* as its fourth argument.

We speculate that this naming is a typo, because in the *execute_swap_operations* function and *ExecuteMsg::AssertMinimumReceive* enum variant, a variable which serves the same purpose is named *minimum_receive*. We recommend fixing this typo for code maturity maintenance.

Fix

The issue was fixed in commit https://github.com/dezswap/dezswap-contracts/commit/b7ccbd4860d92794a9952fa85c9ef7925a0ab7f0

Recommendation #2: Redundant usage of Decimal::from_str in compute_swap

The pair has a fixed commission rate, which is stored as a constant string. Thus, when it is converted to a decimal data type, the *Decimal256::from_str* API is called for every time. Because string processing is an avoidable process when constructing numbers, we recommend refraining from using *from_str*.

```
const COMMISSION_RATE: &str = "0.003";
let commission_rate = Decimal256::from_str(COMMISSION_RATE)?;
```

For example, using the *from_atomics* API to construct the decimal avoids string processing, which is better in terms of gas consumption.

```
let commission_rate = Decimal256::from_atomics(3u64, 3).unwrap();
```

Fix

The issue was fixed in commit https://github.com/dezswap/dezswap-contracts/commit/eb2d108a43455e5440fa3e125fc355a15f73e155

Recommendation #3: Remove dead code

Dead code increases contract code size, a factor that increases deployment cost. The dead code we found is a total of three. One of them is in the pair, *amount_of* is never called and should be subject to removal. Note that this example was not detected by the compiler or code analyzers (ex. Clippy) because it is declared as a public function. Thus, such instances must be manually found.

```
// pair.rs
pub fn amount_of(coins: &[Coin], denom: String) -> Uint128 {
    match coins.iter().find(|x| x.denom == denom) {
        Some(coin) => coin.amount,
        None => Uint128::zero(),
    }
}
```

```
// pair error.rs
#[error("Too small offer amount")]
TooSmallOfferAmount {},
```

```
// pair error.rs
#[error("Too small offer amount")]
TooSmallOfferAmount {},
```

Fix

The issue was fixed in commit https://github.com/dezswap/dezswap-contracts/commit/784e59d5a5d31bcdb0a74432d0eddd652339af66

Revisions

Revision	Date	History
1.0	2022.12.17	Original writing
1.1	2023.03.02	Code revision

Appendix: Test Methodologies

Testcases

We made a collection of all the test code as well as code coverage reports. Because the repository contains proof-of-concepts for security issues, we have made it private for now.

- https://github.com/dream-academy/dezswap-tc

Methods

The testcases in the repository above mostly consists of python code, which may raise an eyebrow for cosmwasm developers. One previous method for testing a wasm smart contract is by writing unit-tests, which are compiled to x86. However, we felt that this method of writing tests is both insufficient and inefficient. It is insufficient because unit-tests cannot capture all of the semantics of the cosmwasm architecture. It is inefficient because it increases the LoC of test code due to calls to `mock_*` functions and requires an extensive knowledge of the cosmwasm semantics to write tests in the first place.

An alternative testing method is to deploy contracts on a testnet. Testing on a testnet captures more semantics than unit-tests and does handles the cosmwasm semantics on behalf of the caller, but has its downsides nonetheless. First, it is not quick enough due to block creation time. Second, testcases are hard to reproduce, as reproduction requires re-starting from contract instantiation. Third, it is impossible to freely modify parameters such as a user's funds, block number, or a value within a contract's storage.

Thus, we created a new testing method that solves all of these problems. This method has three major novelties: First, contract storage is constructed based on its on-chain current state, which is fetched via RPC(or LCD) queries. Second, the stack-like submessage passing semantics is implemented so that invocation of other contracts are done automatically. Third, it is possible to freely modify states and tx metadata such as bank balances, storage, and message info. Due to these three characteristics, it is possible to create a lightweight, local 'fork' of an existing chain and execute/instantiate contracts on top of it.

It also has Python bindings which enables uses to write tests in Python.