Zellic

April 7, 2025

Prepared for
Grigoriy Kopeliovich
Nikita Kinelovskii
Oleg Malyshev
Stepan Bazrov
Bidask Protocol

Prepared by
Syed Faraz Abrar
Qingying Jie
Zellic

Bidask V2

Smart Contract Security Assessment

Contents

About Zellic **4**

1. Overview **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 6

2. Introduction **6**

2.1. About Bidask V2 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. Potential integer overflow in the function `handle_swap` 11

3.2. The refund operation of the LP account could be partially executed 14

3.3. The native pool does not reserve the protocol fee for native tokens 16

3.4. First depositor controls the pool's initial price and bin 18

3.5. Noncompliant transfer-notification parsing 20

3.6. Incorrect excess-amount calculation 22

3.7. Duplicated gas-amount calculation 24

3.8. Pool can get stuck in the `INITING` state 26

3.9. Incorrect payload passed to the function `pay_add_liquidity_fallback` 27

3.10. The range price could be set to the boundary when there are no active bins 29

4. Discussion 30

4.1. Dormant range contracts 31

4.2. Test suite 32

5. System Design 32

5.1. Flow: Swapping tokens 33

5.2. Flow: Add liquidity 39

6. Assessment Results 42

6.1. Disclaimer 43

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Bidask Protocol from March 10th to March 31st, 2025. During this engagement, Zellic reviewed Bidask V2's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Could an attacker steal liquidity deposited by other users?
- Could an attacker trigger a denial of service that prevents users from using the protocol?
- Could an attacker cause users' tokens to be stuck in a pool?
- Could user tokens get stuck in a pool accidentally?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- The amount of gas required for each operation
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. During this assessment, we primarily focused on the swap and add-liquidity flows, specifically on critical-severity vulnerabilities that could cause users' tokens to get stuck or stolen. We reported one vulnerability related to a gas check that was not sufficient and could cause an operation to fail in the middle of execution after passing the gas check; however, we did not exhaustively ensure that all the gas checks were correct in all the operations. Additionally, we analyzed the swap and deposit / remove-liquidity math, but more time for analysis could have been beneficial. However, this was not possible in the audit's time frame, as we prioritized our time on other important aspects of the codebase.

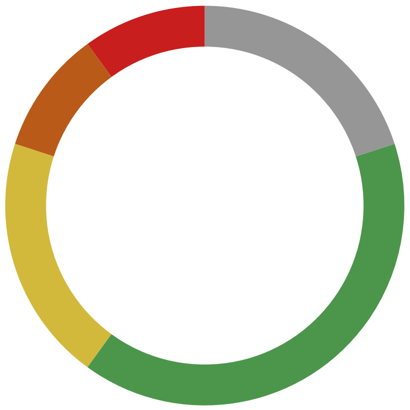
1.4. Results

During our assessment on the scoped Bidask V2 contracts, we discovered 10 findings. One critical issue was found. One was of high impact, two were of medium impact, four were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Bidask Protocol in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	1
🟧 High	1
🟨 Medium	2
🟩 Low	4
⬜ Informational	2



2. Introduction

2.1. About Bidask V2

Bidask Protocol contributed the following description of Bidask V2:

> Bidask is a CLMM DEX architected natively for the TON Blockchain — optimized for speed, scalability, and efficiency. The fundamentally new TON DeFi protocol design brings security challenges, and security is our top priority.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Bidask V2 Contracts

Type	FunC
Platform	TON

Target	Excluding the limit order creation and execution
Repository	https://github.com/bidask-protocol/bidask-v2-core ↗
Version	`6ac271d5f763ca84ddab729470669afbfe3f4592`
Programs	`contracts/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 4.8 person-weeks. The assessment was conducted by two consultants over the course of 16 calendar days.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io ↗

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Syed Faraz Abrar
Engineer
faith@zellic.io ↗

Qingying Jie
Engineer
qingying@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 10, 2025	Kick-off call
March 10, 2025	Start of primary review period
March 31, 2025	End of primary review period

3. Detailed Findings

3.1. Potential integer overflow in the function `handle_swap`

Target	range		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Before calling the function `execute_swap`, the function `handle_swap` will use the function `to_funny` to left shift some input data by 128 bits, converting it into the so-called funny number, which can retain certain precision during the calculation.

```
;; @brief Convert regular int to funny-number
int to_funny(int number) asm "128 LSHIFT#";

() handle_swap(slice in_msg_body, slice sender_address, int msg_value,
    int fwd_fee) impure inline {
    ;; [...]
   amount = to_funny(amount);
   out = to_funny(out);
   exact_out = to_funny(exact_out);

    execute_swap(msg_value, fwd_fee, account?, amount, out, exact_out,
    last_price, is_x, from_user, ref_cell, additional_data, reject_payload,
    forward_payload);
}
```

When executing the function `execute_swap`, the range contract will send an `op::continue_swap` message to itself or an adjacent range contract in certain cases. Since the range contract processes the `op::continue_swap` message using the function `handle_swap`, the `amount`, `out`, and `exact_out` need to be converted back from the funny number before calling the function `send_continue_swap`.

```
() execute_swap (int msg_value, int fwd_fee, int account?, int amount,
    int out, int exact_out, int last_price, int is_x,
            slice user_address, cell ref_cell, cell additional_data,
    cell reject_payload, cell forward_payload) impure inline {
    ;; [...]
    while ((amount > 1) & (no_exact_out | (out < exact_out)) &
    inside_edge_price(last_price, is_x) & (~ got_empty_range)) {
```

```
    (P_a, P_b) = load_price_bounds(storage::current_bin);
    if ((storage::sqrt_p <= P_a) & is_x) {
        ;; [...]
        elseif (left_range_exists()) {

            force_save_liquidity();
            send_continue_swap(storage::left_range_address, account?,
is_x, from_funny(amount), from_funny(out),
                exact_out, last_price, ref_cell, user_address, additional_
                    data, reject_payload, forward_payload);
            save_storage();
            return ();

        }
        ;; [...]
    } elseif ((P_b <= storage::sqrt_p) & (~ is_x)) {
        ;; [...]
        elseif (right_range_exists()) {

            force_save_liquidity();
            send_continue_swap(storage::right_range_address, account?,
is_x, from_funny(amount), from_funny(out),
                exact_out, last_price, ref_cell, user_address, additional_
                    data, reject_payload, forward_payload);
            save_storage();
            return ();

        }
        ;; [...]
    } else {

        ;; [...]

        if (gas_consumed > gas_limit - 100000) { ;; 100000 for all other
instructions
            force_save_liquidity();
            send_continue_swap(my_address(), account?, is_x,
from_funny(amount), from_funny(out),
                exact_out, last_price, ref_cell, user_address, additional_
                    data, reject_payload, forward_payload);
            save_storage();
            return ();
        }
        ;; [...]
    }
```

```
        }
        ;; [...]
    }
```

Impact

When `exact_out` is greater than zero, the continue-swap operation will fail due to integer overflow, and the input tokens will be locked in the pool.

Recommendations

Consider using the function `from_funny` to convert `exact_out` back before calling the function `send_continue_swap`.

Remediation

This issue has been acknowledged by Bidask Protocol, and a fix was implemented in commit `cff58459` ↗.

3.2. The refund operation of the LP account could be partially executed

Target	lp_account		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The LP account is an intermediary management contract used when providing liquidity for two types of jettons simultaneously. It records the amount of jettons currently sent by the user.

The owner of the LP account can use the operation `op::refund_me` to send a request to the pool to withdraw the jettons sent to the pool but not yet deposited. However, the LP account forwards the remaining value to the pool without checking if the `msg_value` is enough to pay the gas for this operation.

```
if(op == op::refund_me) {
    throw_unless(error::NO_LIQUIDITY, (storage::amount0 > 0) |
    (storage::amount1 > 0));

    builder msg = begin_cell()
        .store_uint(op::cb_refund_me, 32)
        ;; [...]
        .store_uint(0, 1);
    send_simple_message(0, storage::pool_address, msg.end_cell(),
    CARRY_REMAINING_GAS);

    storage::amount0 = 0;
    storage::amount1 = 0;

    save_storage();
    return ();
}
```

Impact

It is possible that the caller does not provide sufficient TONs, causing the refund operation to be partially executed. For example, `op::refund_me` may execute successfully in the LP account, but `op::cb_refund_me` in the pool may fail due to running out of gas. In this case, the `storage::amount0` and `storage::amount1` in the LP account have already been updated. The user cannot retry this refund, and their funds are locked in the pool.

Recommendations

Consider checking if the `msg_value` is sufficient to cover the gas required for full execution of the refund operation.

Remediation

This issue has been acknowledged by Bidask Protocol, and a fix was implemented in commit `9f334f9c` ↗.

3.3. The native pool does not reserve the protocol fee for native tokens

Target	pool		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The pool contract collects protocol fees from the returned funds of the swap and accumulates them in the `storage::collected_token1_protocol_fee` or `storage::collected_token2_protocol_fee`.

```
() handle_swap_success(slice in_msg_body, int msg_value, slice sender_address)
    impure inline {

    var (account?, amount1, amount2, is_x, to, has_ref, ref_addr,
    additional_data, reject_payload, forward_payload) =
    in_msg_body~parse_swap_success();
    ;; [...]
    storage::token1_amount -= amount1;
    storage::token2_amount -= amount2;
    ;; [...]
    if (is_x) { ;; is_x declares which of two tokens was swapped.  If is_x is
    true, token1 was swapped to token2
        ;; [...]
        ref_fee2 = amount2~get_fees(is_x, has_ref);
        storage::token2_amount += ref_fee2;
    }
    else {
        ;; [...]
        ref_fee1 = amount1~get_fees(is_x, has_ref);
        storage::token1_amount += ref_fee1;
    }
    ;; [...]
}
```

```
(int, (int)) ~get_fees(int amount, int is_x, int has_ref) impure inline {
    int protocol_fee = amount * storage::protocol_fee / BASE_FEE;
    ;; [...]
    amount -= protocol_fee;
    if (is_x) {
        storage::collected_token2_protocol_fee += protocol_fee;
    }
    else {
        storage::collected_token1_protocol_fee += protocol_fee;
    }
    return (amount, (ref_fee));
}
```

Additionally, the pool uses the `raw_reserve` mechanism and message mode 128 to ensure that the contract balance after each operation is a specific value. For a pool with both assets as jettons, this value is the `BUFFER_AMOUNT`. For a native pool where one of the assets is TON, this value should be `storage::token2_amount + storage::collected_token2_protocol_fee + BUFFER_AMOUNT`, but only `storage::token2_amount + BUFFER_AMOUNT` is reserved in the implementation.

Impact

The native pool cannot collect protocol fees in native tokens.

Recommendations

Reserve `storage::token2_amount + storage::collected_token2_protocol_fee + BUFFER_AMOUNT` amount of TONs for the native pool.

Remediation

This issue has been acknowledged by Bidask Protocol, and a fix was implemented in commit `72713f09` ↗.

3.4. First depositor controls the pool's initial price and bin

Target	pool, range		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

Initially, all range contracts will have their `storage::sqrt_p` state variable set to `0`. Looking at the `handle_provide_liquidity()` function in the range contract, a user is able to set up the initial bin and price arbitrarily (`current_bin` and `initial_sqrt_price` are controlled by the depositor):

```
() handle_deposit_liquidity(slice in_msg_body, int msg_value, int fwd_fee)
    impure inline {
    notify_amount = notify_nearbies();
    ;; [ ... ]

    if (storage::sqrt_p == 0) {
        int success = init_range(current_bin, initial_sqrt_price);
        ifnot (success) {
            pay_add_liquidity_fallback(storage::first_bin(), user_address,
    provide_amount_x, provide_amount_y, forward_payload);
            commit();
            throw(error::POOL_INIT_FAILED);
        }
    }

    ;; [ ... ]
}
```

Note that both `current_bin` and `initial_sqrt_price` are controllable only when the pool itself has not had any deposits, meaning that the depositor must be the first depositor.

Impact

Since the first depositor is able to control these parameters, they can set up the pool with an arbitrary initial price and bin. This allows them to set the price to an extremely high or low value, which would then disincentivise users from using the pool.

This bug could be alleviated by recreating a new pool contract, but an attacker who could continuously front-run other first depositors can continuously set these extreme initial prices and bins. This would end up either disincentivising users from using these pools or forcing them to use these pools without other alternatives.

The potential severity of the vulnerability is critical, as it could allow an attacker to set up ranges with invalid prices and active bins in relation to each other. One example is that the attacker can provide a specific `current_bin` to cause the function `init_range` to fail, making an uninitialized range send an `op::range_notify` message to its neighbor ranges. This will affect the execution of the swap flow. Because the `op::range_notify` message informs the neighbor ranges of its existence, the `op::continue_swap` message may be sent to this uninitialized range. This could result in users losing their funds for the swap.

Recommendations

We recommend that the pool's deployer sets up the initial price and bin. Subsequent ranges can be set up with prices and bins set up on the edges of their price range.

For example, if we have ranges (A, B, C) for a pool, the pool deployer might choose a price and bin within range B. Afterwards, range A would automatically have the price set to the highest price possible in itself, with the highest possible bin set as the active range, and range C would have the opposite (lowest possible bin with lowest possible price).

Remediation

This issue has been acknowledged by Bidask Protocol, and fixes were implemented in the following commits:

- `9932436a` ↗
- `d533f46d` ↗

3.5. Noncompliant transfer-notification parsing

Target	pool		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

The implementation of the pool contract parsing the `op::transfer_notification` message is not compliant with the official jetton standard.

```
if (op == op::transfer_notification) { ;; for jetton income proccesing
    (int jetton_amount, slice from_user, slice ref_ds) =
        (in_msg_body~load_coins(), in_msg_body~load_msg_addr(),
    in_msg_body~load_ref().begin_parse());
    ;; [...]
}
```

It assumes that the `forward_payload` data is always stored in a cell reference, but according to the specification ↗, the `transfer_notification` type is:

```
transfer_notification#7362d09c
  query_id:uint64
  amount:(VarUInteger 16)
  sender:MsgAddress
  forward_payload:(Either Cell ^Cell) = InternalMsgBody;
```

The `forward_payload` could be in-lined in the transfer-notification message.

Impact

This issue could cause the transaction to fail with assets that in-line the forward payload.

Recommendations

Consider handling all possible cases allowed by the specification when parsing transfer notifications.

Remediation

This issue has been acknowledged by Bidask Protocol.

Bidask Protocol provided the following response to this finding:

> We decided not to support another interfaces, because user can fully control `transfer_notification` flow with `forward_payload` delivery to pool.

3.6. Incorrect excess-amount calculation

Target	range		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

The function `pay_excess_and_mint_lp` is used to send `op::mint` messages to the LP wallet and return excess TONs to the user. However, when calculating the `amount_for_excess`, the `total_gas_consumed` is mistakenly used, which is the total required gas amount instead of the total required gas fee.

```
() pay_excess_and_mint_lp(int msg_value, int forward_fee, slice user_address,
     int x_excess, int y_excess,
                          int bins_num, cell lp_tokens_to_mint,
     cell forward_payload) impure inline {
     int total_gas_consumed  = gas_consumed() + 25000 + notify_amount *
     (GAS_FOR_NOTIFY + 8000); ;; 25 000 - gas for remaining operations
     int amount_for_excess = (msg_value - total_gas_consumed) / 2;
     ;; [...]
}
```

Impact

Since the gas amount is not multiplied by the gas price to obtain the gas fee, the computed `amount_for_excess` may be higher than expected, resulting in fewer TONs being sent with the `op::mint` message.

Recommendations

Consider updating this based on the following code:

```
int amount_for_excess = (msg_value - total_gas_consumed) / 2;
int amount_for_excess = (msg_value - total_gas_consumed * gas_price) / 2;
```

Remediation

This issue has been acknowledged by Bidask Protocol, and a fix was implemented in commit `3b79cdb6` ↗.

3.7. Duplicated gas-amount calculation

Target	range		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Low

Description

The `total_gas_consumed` calculated by the function `pay_excess_and_mint_lp` consists of three parts: 1) the gas amount already consumed by executing the operation `op::range_provide_liquidity` or `op::continue_provide_liquidity` in the range contract, 2) the gas amount required for the remaining execution, and 3) the gas amount needed to send the `op::range_notify` message.

```
() pay_excess_and_mint_lp(int msg_value, int forward_fee, slice user_address,
    int x_excess, int y_excess,
                          int bins_num, cell lp_tokens_to_mint,
    cell forward_payload) impure inline {
    int total_gas_consumed  = gas_consumed() + 25000 + notify_amount *
    (GAS_FOR_NOTIFY + 8000); ;; 25 000 - gas for remaining operations
    ;; [...]
}
```

The gas amount for sending the `op::range_notify` message includes the gas amount to send and the gas amount required to execute the function `notify_nearbies`. But the latter is already included in the first part of the `total_gas_consumed`.

```
int notify_nearbies() impure inline_ref {
    int amount = 0;
    ifnot (storage::left_notified) {
        ;; [...]
        send_message_with_stateinit(GAS_FOR_NOTIFY * gas_price,
    left_range_address, left_range_state_init, body.end_cell(), 0);
        amount += 1;
    }
    ;; [...]
}
```

Impact

The gas amount required for executing the function `notify_nearbies` is double-counted, resulting in the range contract charging higher fees than expected.

Recommendations

Consider updating this based on the following code:

```
int total_gas_consumed  = gas_consumed() + 25000 + notify_amount * (GAS_FOR_
    NOTIFY + 8000);
int total_gas_consumed  = gas_consumed() + 25000 + notify_amount * GAS_FOR_
    NOTIFY;
```

Remediation

This issue has been acknowledged by Bidask Protocol, and a fix was implemented in commit 489b2095 ↗.

3.8. Pool can get stuck in the `INITING` state

Target	pool		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The gas check in the handler for `op::add_liquidity` in the pool contract is insufficient. From testing, we have found that a user can provide just enough TON to pass the check but then cause the transaction fo fail in the middle of handling the operation.

Additionally, there is no check to ensure that the `sqrt_p` provided by the first depositor is not zero.

Impact

In both the above cases, the pool will get stuck in the `INITING` state, which will then require an admin to unlock the pool manually.

Recommendations

We recommend adding gas benchmark tests in order to know how much gas is used in each operation. This will allow refining all gas checks in all contracts so that transactions cannot fail in the middle of an operation.

We also recommend adding a check that ensures that the initial `sqrt_p` set by the first depositor is not zero.

Remediation

This issue has been acknowledged by Bidask Protocol, and a fix was implemented in commit `67b5db36` ↗.

3.9. Incorrect payload passed to the function `pay_add_liquidity_fallback`

Target	range		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The function `handle_deposit_liquidity` receives `reject_payload` and `forward_payload` from the input. When `init_range` fails, it passes `forward_payload` to the function `pay_add_liquidity_fallback` to construct and send an `op::add_liquidity_fallback` operation to the pool.

```
() handle_deposit_liquidity(slice in_msg_body, int msg_value, int fwd_fee)
    impure inline {
    ;; [...]
    (int provide_amount_x, int provide_amount_y, int first_four_bins,
    cell tokens, slice user_address,
     cell reject_payload, cell forward_payload, int current_bin,
    int initial_sqrt_price) = parse_provide_custom_liquidity(in_msg_body);

    if (storage::sqrt_p == 0) {
        int success = init_range(current_bin, initial_sqrt_price);
        ifnot (success) {
            pay_add_liquidity_fallback(storage::first_bin(), user_address,
                provide_amount_x, provide_amount_y, forward_payload);
            commit();
            throw(error::POOL_INIT_FAILED);
        }
    }

    ;; [...]
}
```

However, the operation `op::add_liquidity_fallback` expects a reject payload.

```
if (op == op::add_liquidity_fallback) { ;; unexpected handler. Not expected to
    be called.
    ;; [...]
    cell reject_payload = in_msg_body~load_maybe_ref();
    ;; [...]

    send_assets(msg_value, 9500, user_address, amount1, amount2,
    reject_payload, reject_payload);
    save_storage();
    return ();
}
```

Impact

The user may be confused if the payload forwarded with the tokens is inconsistent with the pool execution result.

Recommendations

Consider passing the `reject_payload` to the function `pay_add_liquidity_fallback`.

Remediation

This issue has been acknowledged by Bidask Protocol, and a fix was implemented in commit `f1d1d419` ↗.

3.10. The range price could be set to the boundary when there are no active bins

Target	range		
Category	Optimization	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

When the price of a pool crosses from one range to another, the old range needs to be left in the correct state.

For example, assume that a pool has three range contracts A, B, and C. If range B is the current active contract, and a swap causes the price to drop and go into range A, then range B's active bin should be set to the very first bin, and its price should be set to the lowest possible price of that bin.

If a swap causes the price to increase and go into range C, then the opposite should happen. (The active bin should be set to the highest possible bin, and the price should be set to the highest possible price.)

In the Bidask protocol, the `move_bin_left()` and `move_bin_right()` functions are used to handle moving from bin to bin. In the case where there is no available bin to the left, the following code is executed in `move_bin_left()`:

```
() move_bin_left() impure inline {
    if ((storage::current_bin % BINS_IN_ONE_ELEMENT == 0)) {
        ;; Zellic: this is the first bin in this bin group
        force_save_liquidity();
        (int index, slice bins_slice, int flag) =
    storage::bins_dict.idict_get_prev?(32,
    get_bins_group(storage::current_bin));
        if (flag) { ;; Zellic: this means a bin was found to the left
            ;; [ ... ]
        }
        else {
            storage::current_bin = storage::first_bin();
        }
        (P_a, P_b) = load_price_bounds(storage::current_bin);
        storage::sqrt_p = P_b; ;; Zellic: set the price to the upper bound of
    the new bin
    }
    else {
        storage::current_bin -= 1;
    }
}
```

In the above code, we can see that if a bin is not found to the left of the current bin, the code simply sets `storage::current_bin` to `storage::first_bin()`. The intention here is for the calling code to later send the swap over to the left range, and so the very first bin is set as the active bin.

However, in this case, the current price of the range (`storage::sqrt_p`) is still set `P_b`, which is the upper bound of the current bin (in this case, the very first bin).

Impact

The function `swap_in_bin` can set the price of the range to the lower bound of the current bin if `is_x` is true or the upper bound if `is_x` is false when no liquidity is in the bin, but it will consume more gas.

Recommendations

When no active bins are found in the current swap direction, we recommend setting the price of the range to the absolute edge of the price boundary.

Remediation

This issue has been acknowledged by Bidask Protocol, and a fix was implemented in commit `af79609c` ↗.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the TON mainnet.

During our assessment on the scoped Bidask V2 contracts, we discovered 10 findings. One critical issue was found. One was of high impact, two were of medium impact, four were of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.