Zellic

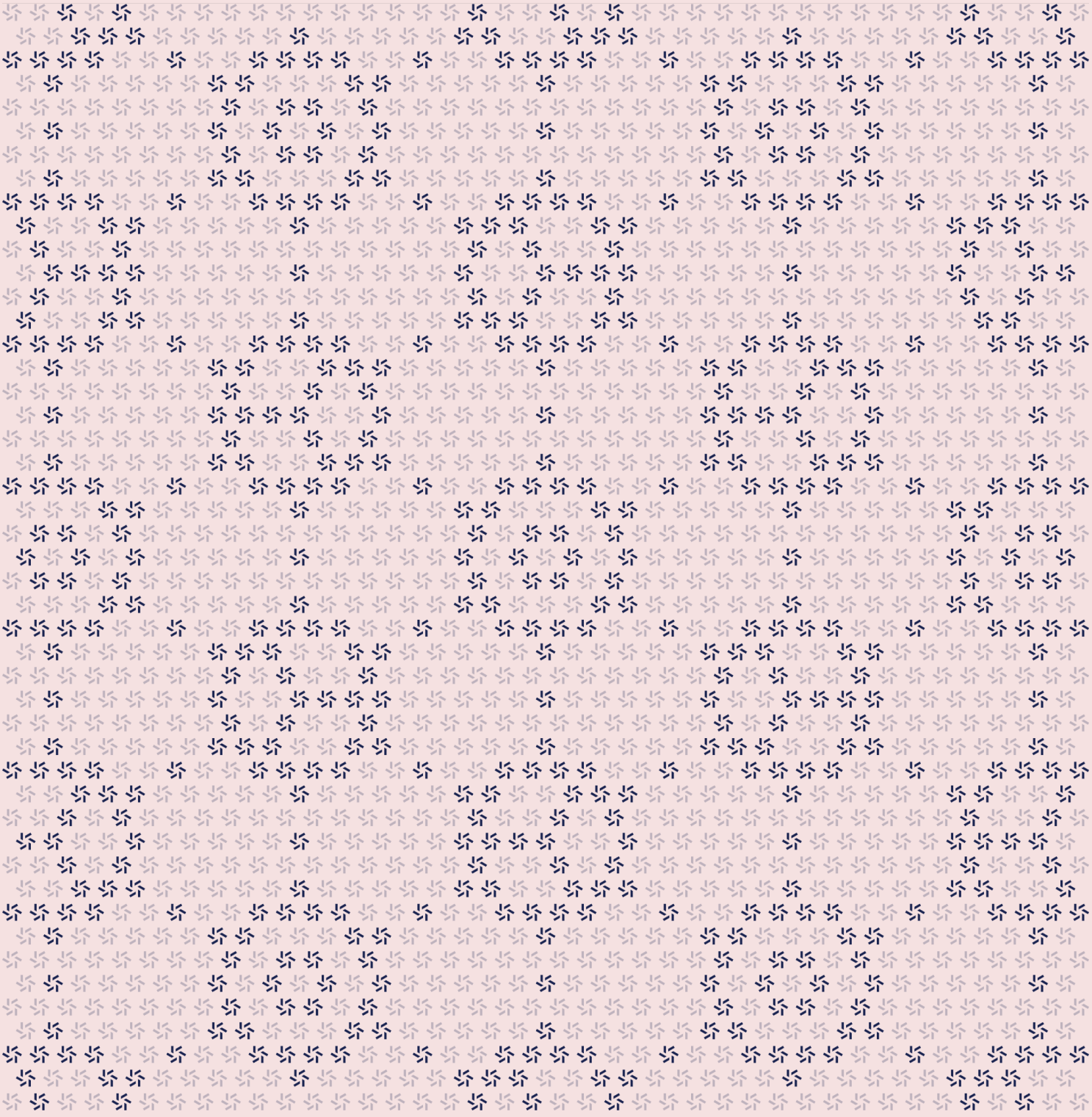
Prepared for
TonTradoor

Prepared by
Filippo Cremonese
Aaron Esau
Zellic

April 18, 2025

Tradoor

Smart Contract Security Assessment



Contents

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 TonTradoor from March 26th to April 16th, 2025. During this engagement, Zellic reviewed Tradoor'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:

- Can an on-chain attacker drain the contract of funds?
- Is it possible for a user to create an uncancelable order?
- Is there a limit to the number of orders that can be created?
- Are integers restricted to a certain range (e.g., unsigned) where applicable?
- Is the settlement logic sound?

1.3. Non-goals and Limitations

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

- Front-end components
- Offchain components
- Infrastructure relating to the project
- Key custody

Additionally, please see our notes regarding the limitations of this assessment in section 6. ↗.

1.4. Results

During our assessment on the scoped Tradoor contracts, we discovered seven findings. No critical issues were found. One finding was of high impact, one was of medium impact, one was of low impact, and the remaining findings were informational in nature. The high, medium, and low severity issues have been remediated, and the corresponding fix commit links are included in this report for reference.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of

TonTradoor in the Discussion section (4. ↗).

While our review did not identify any critical severity issues, we believe there are still areas that would benefit from further refinement before the code is moved to production. We recommend developing a more comprehensive test suite and adding clear documentation to improve maintainability, support future reviews, and build trust with end users.

Due to the time-boxed nature of security assessments, there are inherent limitations to the depth and breadth of coverage possible. In this assessment, limited documentation — both within the code and in external resources — made it challenging to fully understand certain design decisions, the intended threat model, and the rationale behind some of the mathematics.

Complex mathematical logic was not clearly explained, which required significant time to interpret. In several cases, this led us down investigative paths that ultimately proved unnecessary, consuming significant time. More thorough documentation could help avoid such detours and enable a more focused and efficient review.

Additionally, external documentation was not provided at the time of the assessment. Often, this meant there was no direction as to the roles of certain entities or the purpose of separating certain roles (e.g., owner from multisig), and resources for answering this were not available. Answering these questions does not directly aid our understanding of the code. However, without insight on the developer's intentions, observations cannot easily be made about in-code mistakes regarding the threat model. This is a significant barrier in an assessment.

In particular, it would be valuable to document:

- All non-trivial math or logic implemented by onchain contracts (pool.tact in particular)
- The roles of the protocol participants and which actions they are entrusted to perform
- The security assumptions for off-chain components

After having performed our assessment, the code we have the least coverage of is the jetton contracts. We performed a cursory review; however, the pool required significant time and attention and was prioritized.

The revision of the codebase under review lacks a comprehensive test suite. This reflects an early stage of development, and can hinder understanding during an assessment. In this protocol, it is especially a concern given the complexity of the pool and the mathematical logic involved. For information regarding our concerns and recommendations for the tests, please refer to section 4.1. ↗.

Given the complexity of this protocol — in addition to improving documentation and tests — we also strongly recommend a comprehensive assessment of the protocol including the offchain executor component. We also recommend setting up a bug-bounty program, to maximize the incentives for participating to coordinated bug disclosure.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	1
Medium	1
Low	1
Informational	4

2. Introduction

2.1. About Tradoor

TonTradoor contributed the following description of Tradoor:

> The project is designed to provide efficient and secure liquidity management and perpetual contract trading support for decentralized trading platforms while ensuring transparency and trustlessness.

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:

Tradoor Contracts

Type	Tact
Platform	TVM
Target	tradoor-contracts-v3-ton
Repository	https://github.com/TonTradoor/tradoor-contracts-v3-ton ↗
Version	`ffd7f56ec43334240fe058c90d0e34ae2a1b1781`
Programs	`contracts/**.tact`
Target	tradoor-contracts-v3-usdt
Repository	https://github.com/TonTradoor/tradoor-contracts-v3-usdt ↗
Version	`2bda3b5936c8a82be4b559ce3523fe37e0a45a0f`
Programs	`contracts/**.tact`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 3.3 person-weeks. The assessment was conducted by two consultants over the course of 3.2 calendar weeks.

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:

Filippo Cremonese
Engineer
fcremo@zellic.io

Aaron Esau
Engineer
aaron@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

March 26, 2025	Start of primary review period
March 27, 2025	Kick-off call
April 16, 2025	End of primary review period

3. Detailed Findings

3.1. Centralized and undocumented protocol design

Target	Project Wide		
Category	Protocol Risks	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The project threat model involves several actors that are fully trusted to behave correctly and not be compromised. The on-chain contracts perform little to no checks against intentional or unintentional malicious behavior.

Additionally, the project is not documented, and the design and code for off-chain components is not available for scrutiny. The executors logic — including the algorithm that selects orders to execute — and the asset-pricing mechanisms are undocumented. The fees structure is particularly confusing and opaque.

Impact

While centralization is not a vulnerability per se, it increases the attack surface of the project, adding single points of failure that might cause unintentional damages (e.g., due to bugs or human errors) or could be targeted and exploited by an attacker.

The lack of documentation and lack of access to code prevented us from having a complete understanding of the interactions between the contracts and the off-chain components. Our review assumed honest and correct behavior from all off-chain components, excluding only the unprivileged and untrusted users acting as liquidity providers or perpetual traders.

Recommendations

Mitigations to limit the potential damage made by intentional or unintentional malicious behavior should be considered. However, the risks due to compromise of trusted actors cannot be eliminated without a significant redesign.

The project should be thoroughly documented, and the code for critical off-chain components (such as the executor) should be considered for an open-source release.

Remediation

This issue has been acknowledged by TonTradoor.

3.2. Order-cancellation race condition may cause double payment

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

Description

The `ExecuteOrCancelCompensate` message handler allows the compensator entity to cancel an order and send funds to the order creator.

```
receive(msg: ExecuteOrCancelCompensate) {
    // [...]
    if(compensate.orderType != null) {
        if (compensate.orderType == ORDER_TYPE_LP) {
            self.liquidityOrders.del(compensate.orderId);
        } else {
            self.perpOrders.del(compensate.orderId);
            self.perpOrderExs.del(compensate.orderId);
        }
    }
    // [...]
}
```

However, it does not check to ensure an order exists. The Tact documentation ↗ specifies that the map's `del` function returns a boolean indicating whether the key was present in the map. So, if the key is not present, the user will be compensated even if the order does not exist.

A consequence of this is that — between the time when the compensator entity signs a transaction that calls the `ExecuteOrCancelCompensate` handler and it is actually executed on chain — there is the opportunity for an order to be canceled, executed, or otherwise deleted from the relevant map.

Note that a user can cancel their own order, meaning one aspect of the race condition may be controlled by the user. If the user has knowledge that the compensator will cancel their order, they may try to race the compensator to cancel the order themselves and thereby receive the funds twice.

Impact

In the worst-case scenario, a malicious attacker may be able to receive a double payment for canceling their order, thereby stealing funds from the protocol. This may also happen by accident if

the protocol is very active.

Recommendations

Assert that the calls to `del` return `true`:

```
receive(msg: ExecuteOrCancelCompensate) {
    // [...]
    if(compensate.orderType != null) {
        if (compensate.orderType == ORDER_TYPE_LP) {
            self.liquidityOrders.del(compensate.orderId);
            require(self.liquidityOrders.del(compensate.orderId), "order not
                found");
        } else {
            self.perpOrders.del(compensate.orderId);
            self.perpOrderExs.del(compensate.orderId);
            require(self.perpOrders.del(compensate.orderId), "order not
                found");
            require(self.perpOrderExs.del(compensate.orderId), "order not
                found");
        }
    }
    // [...]
}
```

Remediation

This finding was addressed in the TON pool contract in commit 68b5af ↗, and in the USDT pool in commit 0d6693 ↗ by implementing the suggested recommendation.

3.3. Missing sanity checks

Target	Project Wide		
Category	Code Maturity	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The codebase generally lacks input sanity checks, which, even assuming no malicious behavior, could prevent issues in case of unintentional mistakes.

Instances of missing checks include (but are not limited to) the following:

- When inserting a perpetual order, the `tpPrice` is not required to be greater (or smaller, depending on whether the order is to increase or decrease the position) than `slPrice`.
- Functions to alter configuration do not enforce sensible numbers for fee parameters, which could even be greater than 100%.
- Lastly, `FeedPrices` does not check that the prices map contains only IDs of listed assets.

Impact

The lack of sanity checks allows the pool users (both privileged and unprivileged) to lose funds or cause damage, even via accidental actions.

Recommendations

Consider adding sanity checks to message parameters where possible. Even assuming honest behavior from the sender of a message, some fields are prone to be set incorrectly, and we recommend to enforce validation for trivial cases.

Remediation

The following checks were added to the TON and USDT pool contracts, respectively in commits `68b5af` ↗ and `0d6693` ↗:

- fee checks for `UpdateBaseConfig`, `FeedPrices`, `WithdrawFee` (for the USDT pool), `ClaimProtocolFee` (for the TON pool), requiring a minimum `ctx.value`
- require that `FeedPrices` messages contain prices for all configured tokens
    - note: `FeedPrices` could still contain prices for non-configured tokens

- check `tpPrice` and `slPrice` consistency when placing or modifying a perpetual order

3.4. Lack of `bounced` handlers

Target	pool		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

As described by the Tact documentation ↗,

> When a contract sends a message with the flag `bounce` set to `true`, and if the message isn't processed properly, it will bounce back to the sender. This is useful when you want to ensure that the message has been processed properly and, if not, revert the changes.

To handle these bounced messages, the contract needs to implement `bounced` handlers — for example:

```
contract MyContract {
    bounced(msg: bounced<MyMessage>) {
        // ...
    }
}
```

There are no bounced handlers in the pool contract, despite many of the messages being sent with the `bounce` flag set to `true`.

Impact

Depending on the intentions of the contract developers and the way the contract is used, these missing handlers could be undesirable.

Recommendations

Create `bounced` handlers for the messages that are sent with the `bounce` flag set to `true`, implementing the logic that is appropriate for each message.

Remediation

This issue has been acknowledged by TonTradoor.

3.5. TON Pool `decreasePerpPosition` deletes potentially pending order

Target	pool		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

Before returning, the TON pool `decreasePerpPosition` function deletes the order associated with the `orderId` passed by the caller:

```
fun decreasePerpPosition(trxId: Int, orderId: Int, opType: Int, account:
    Address, tokenId: Int, isLong: Bool, marginDelta: Int, sizeDelta: Int,
    tradePrice: Int, fundingFeeGrowth: Int, rolloverFeeGrowth: Int): Int {
    // [...]

    accountPerpPosition.positions.set(account, directionPerpPosition);
    self.perpPositions.set(tokenId, accountPerpPosition);
    self.globalPositions.set(tokenId, globalPosition);
    self.globalLPPositions.set(tokenId, globalLPPosition);
    self.perpOrders.del(orderId);

    send(SendParameters{ to: account, value: payout, mode:
    SendIgnoreErrors | SendPayGasSeparately});

    // [...]

    return payout;
}
```

The `decreasePerpPosition` function of the USDT pool does not perform this operation.

Impact

The difference in behavior does not appear to be intentional. In the version of the codebase under review, there is no impact: the `decreasePerpPosition` function is called for two reasons:

- when executing an order – in this case, the code of both pools explicitly removes the order after calling `decreasePerpPosition`; the `del` function does not fail if the map does not contain the key being removed, so no error is thrown.

- when performing auto deleverage – in this case, the `orderId` given to `decreasePerpPosition` is fictitious, and the map does not actually contain an order with the given ID.

However, this kind of differences in behavior in otherwise identical functions is subtle and could cause bugs to be introduced in future revisions of the code.

Recommendations

We recommend to revise the code so the behavior of the `decreasePerpPosition` function is uniform for both contracts; alternatively, clearly document the difference in behavior and its rationale.

Remediation

This issue has been acknowledged by TonTradoor.

3.6. Time-out is bundled with the multisig request

Target	multisig		
Category	Protocol Risks	**Severity**	High
Likelihood	Low	**Impact**	Medium

Description

The time-out is bundled with the multisig request contracts (MultisigSigner) through the `request` cell that is passed in the constructor.

We have some concerns about the effectiveness of this design. For more details, please see section 4.4. ↗.

Impact

It could be problematic if, for example, a malicious member submits proposals with effectively infinite time-outs.

Then, the malicious member has time to compromise the other members. And when the owner changes the members in the Multisig contract, it will have no effect on the maliciously created MultisigSigner contracts.

Recommendations

We recommend not bundling the request time-out with the request.

Remediation

This finding was addressed in the TON multisig contract in commit `68b5af` ↗, and in the USDT multisig in commit `0d6693` ↗. The timeout after which a proposal expires is now a fixed value, enforced by the main Multisig contract.

3.7. Unused parameters in several messages

Target	ExecutePerpOrder		
Category	Code Maturity	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `ExecutePerpOrder` message contains a `tokenId` field, which is not used by the message handler. This parameter is not only unused but also unnecessary, since the correct token ID is already recorded in the order stored in the contract.

Additionally, the following messages contain a `premiumRate` field, which is not used by the contract:

- `ExecutePerpOrder`, which allows the executor to execute a perpetual order
- `LiquidatePerpPosition`, which liquidates a user's perpetual position
- `ADLPerpPosition`, which is used to decrease a user's perpetual position as part of the automatic-deleveraging (ADL) mechanism

Impact

These parameters are unused, and therefore this issue is reported as an Informational code-maturity finding.

Even if the `tokenId` parameter were used — and not checked to match the token ID of the order being executed — this would not meaningfully affect the threat model of an executor.

All of these message handlers are only callable by the executor, which is already capable of adversely affecting order execution via other parameters such as the asset price.

Recommendations

Remove the unneeded `tokenId` field from the `ExecutePerpOrder` message.

Remove `premiumRate` from the `ExecutePerpOrder`, `LiquidatePerpPosition`, and `ADLPerpPosition` messages.

Remediation

This issue has been acknowledged by TonTradoor.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Test suite

In our opinion, test coverage for Tradoor is not sufficient.

When building a complex contract ecosystem with multiple moving parts and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios. Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

It is important to test the invariants required for ensuring security and also verify mathematical properties from the project's specifications.

Few tests cover negative scenarios

First, while all of the major internal functions are tested with positive scenarios, there are few tests for negative scenarios.

Good test coverage has multiple effects.

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in your product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point seems contradictory, given the time investment to create and maintain tests. To expand upon that, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks something if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence. Tests have your back here. They are an indicator that the existing functionality most likely was not broken by your change to the code.

Throughout section 5. ↗, we document a great number of untested negative scenarios (and in some cases, untested positive scenarios). We recommend walking through every instruction and writing the missing tests where possible.

Contract complexity warrants more positive-scenario tests

Additionally, while there are integration tests for most handlers, these are too few given the complexity of the contract state; most of the callable code greatly depends on the current contract state (the global LP position, open orders, etc.).

We recommend writing tests that hardcode various possible/plausible contract states and testing each of the important functions (order creation, liquidation, etc.) against these states.

Finally, we recommend fuzz testing the contract to simulate users interacting with the contract in various ways. This is a great way to find edge cases and bugs that are difficult to identify through manual testing.

4.2. Limited contract storage enables denial-of-service attacks

In the pool contracts, orders are stored in a map. However, in Tact, there is a limit to the number of cells that can be stored in a map. As described by the Tact documentation ↗,

> As the upper bound of the smart contract state size is around 65000 items of type `Cell`, it constrains the storage limit of maps to about 30000 key-value pairs for the whole contract.
>
> The more entries you have in a map, the higher compute fees you will incur. Thus, working with large maps makes compute fees difficult to predict and manage.

If a user were to create a large number of orders, the map would fill up, and users would be unable to create additional orders. It would require manual intervention from an executor to cancel orders.

In practice, this attack may be prohibitively expensive — in terms of capital required to start the attack and the cost of the fees not being refunded to the attacker — because execution fees are collected at the time orders are created. Additionally, when an order is canceled, executors have the ability to specify an address to receive the fee refund from the order.

If the TonTradoor team is concerned about users being blocked from creating orders at specific times (e.g., if the attacker is expecting a user to create a large order and attempts to block it from happening), one step to further mitigate the attack is to make the fees for order creation scale with the number of existing orders. In this way, creating a large number of orders would be even more likely to be prohibitively expensive, but creating a single (legitimate) order may be worth the fees.

4.3. Unused `isPending` flag on perpetual orders

Note that the `isPending` flag, which is stored in `perpOrders`, is unused. We recommend removing it to simplify the code and save gas costs. For example, in the following handler, make these suggested changes:

```
receive(msg: ExecutePerpOrder) {
    let ctx: Context = context();

    require(self.executors.exists(ctx.sender), "invalid sender");
    let orderOpt: PerpOrder? = self.perpOrders.get(msg.orderId);
    require(orderOpt != null, "order not exist");
    let order: PerpOrder = orderOpt!!;

    require(ctx.value >= self.executePerpOrderGas + self.transferJettonGas,
    "gas not enough");
    require(!order.isPending, "order is pending");

    order.isPending = true;
    self.perpOrders.set(msg.orderId, order);

    if (!(order.opType == OP_TYPE_PERP_INCREASE_MARKET || order.opType ==
    OP_TYPE_PERP_DECREASE_MARKET)){
        require(order.triggerAbove ? (msg.price >= order.triggerPrice)
    : (msg.price <= order.triggerPrice), "not reach trigger price");
    }
    let extraGas: Int = 0;
    let increase: Bool = order.opType == OP_TYPE_PERP_INCREASE_MARKET ||
    order.opType == OP_TYPE_PERP_INCREASE_LIMIT;
    // [...]
```

4.4. Additional notes regarding `multisig.tact`

The following notes regard our observations in the multisig contract. These are not necessarily issues or security related.

Note that we have made one finding regarding the multisig contract, which is in section 3.6. ↗.

Proposals are invalidated when the configuration is updated

Proposals become "invalidated" if the configuration (i.e., members or threshold) is updated.

This is because the `receive(msg: Signed)` handler computes `opInit: StateInit = initOf MultisigSigner(myAddress(), self.members, self.requiredWeight, msg.request)` using `self.members` and `self.requiredWeight`. So, if the values change after the child contract is created, the `Signed` message will be rejected by the `require(opAddress == sender, "Invalid sender")` assertion.

Resolving this requires redesigning the contracts. In our opinion, the safest solution is to have the parent contract be the one that signers call to approve of a proposal. The parent contract would then send a message to the child contract, which is only used for storage of approvers. When the last signer determines that enough signers are in storage in the child contract, they call a function on the child contract that bundles and sends all of the signers to the parent contract. Finally, the parent contract then checks that the current signers are in that bundle and executes the proposal.

Unnecessary time-out check

In MultisigSigner, there is the check

```
require(self.request.timeout > now(), "Timeout");
```

However, Multisig's `receive(msg: Signed)` handler also checks the time-out, so there is no benefit in checking it in MultisigSigner. Consider removing the check for additional gas savings.

5. Threat Model

This provides a full threat model description for various messages. As time permitted, we analyzed the attack surface of the contracts and created a written threat model for some critical messages. A threat model documents a given message externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all the attack surface in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Contract: USDT Pool

Message: `ListToken`

This message allows to list a new token or edit the configuration of an existing asset, setting parameters including the token name, whether the asset is enabled, the maximum leverage, liquidation, and trading fees.

The message is only accepted if sent by the pool manager.

Inputs

The incoming message has the following structure:

```
message ListToken {
    tokenId: Int as uint16;
    config: TokenConfig;
}
```

- `tokenId`
  - **Validation**: None.
  - **Impact**: Unique ID identifying the token to be added or edited.
- `config`
  - **Validation**: None.
  - **Impact**: Specifies several configuration parameters of the given token.

The `config` allows to specify the following parameters:

```
struct TokenConfig {
    name: String;
    enable: Bool;
    // ==================== Trader Position Config ====================
    maxLeverage: Int as uint16;
    liquidationFee: Int as coins;
    maintenanceRate: Int as uint32;
```

```
    // ==================== Trading Fee Config ====================
    tradingFeeRate: Int as uint32;
    lpTradingFeeRate: Int as uint32;
}
```

- `name`: Name of the asset.
- `enable`: Specifies whether the asset is enabled — perpetual positions for disabled assets cannot be increased, decreased, or liquidated.
- `maxLeverage`: Maximum leverage allowed for perpetual positions in this specific asset.
- `liquidationFee`: Flat USDT liquidation fee taken from the owner of an unhealthy position when liquidated.
- `maintenanceRate`: Percentage rate required for perpetual positions to be considered healthy.
- `tradingFeeRate`: Fee charged to the user when entering or exiting a position. The rate is a percentage of the total position size.
- `lpTradingFeeRate`: Percentage of the trading fees distributed to the liquidity providers — the rest of the trading fees are taken by the pool owner.

Test coverage

The basic logic of the message handler is executed by the test-harness setup. However, the function is not individually tested. The test suite also does not contain any negative tests, including for instance a check that ensures only authorized senders are allowed.

Intended branches

- ☑ Successfully registers a token (indirectly tested).

Negative behavior

- ☐ Sender is unauthorized.

Message: `DelistToken`

This message allows to delist a token, removing it from the list of assets that are supported by the pool.

The message is only accepted if sent by the pool manager and only if there are no outstanding long or short perpetual positions.

Inputs

The incoming message has the following structure:

```
message DelistToken {
    tokenId: Int as uint16;
}
```

- `tokenId`
  - **Validation**: Must correspond to a previously listed token for which there are no outstanding perpetual positions.
  - **Impact**: Unique ID identifying the token to be delisted.

Test coverage

The basic logic of the message handler is covered by unit tests. The test suite does not contain any negative tests, including authorization checks.

Intended branches

- ☑ Successfully deregisters a token.

Negative behavior

- ☐ Sender is unauthorized.

Message: `ClaimProtocolFee`

This operation allows to transfer fees collected by the pool to an external arbitrary address.

The message is only accepted if sent by the claimer address.

Inputs

The incoming message has the following structure:

```
message ClaimProtocolFee {
    trxId: Int as uint64;
    feeReceiver: Address;
}
```

- `trxId`
  - **Validation**: None.
  - **Impact**: Identifier for the message, not used by the handler.
- `feeReceiver`

- **Validation**: None.
- **Impact**: Address that will receive the protocol fees.

Test coverage

The logic of the message handler is covered by the test suite.

The test suite does not contain negative tests, including a check that messages sent from unauthorized addresses are rejected.

Intended branches

- ☑ Successfully transfer the protocol fees to the specified fee receiver address.

Negative behavior

- ☐ Sender is not the claimer.
- ☐ There are no fees to claim.
- ☐ The message value is insufficient to cover for jetton transfer fees (not asserted by the contract).

Message: `FeedPrices`

This operation allows an executor to provide updated prices as well as new values for `globalRolloverFeeGrowth` and `globalLpFundingFeeGrowth`. The message is only accepted if sent by one of the allowlisted executor addresses.

The updated prices are provided as a map that completely replaces the previous prices stored in the contract storage. The keys and values of the new map — corresponding to the asset ID and price, respectively — are not validated in any way.

The message handler recalculates the fees due to the protocol and to the liquidity providers.

Liquidity-provider fees are increased by two logical factors:

1. `msg.lpFundingFeeGrowth - self.globalLpFundingFeeGrowth` — term due to funding fees
2. `(msg.rolloverFeeGrowth - self.globalRolloverFeeGrowth) * self.lpRolloverFeeRate / PERCENTAGE_BASIS_POINT` — term due to a percentage of the rollover fees

Protocol fees are also increased by one logical factor: `msg.rolloverFeeGrowth - self.globalRolloverFeeGrowth - lpRolloverFeeGrowthDelta` (the remaining percentage of the rollover fees not credited to the LPs).

We note that the fee recalculation only determines an internal accounting change (to the `self.globalLPFund` and `self.protocolTradingFee`) and this handler does not transfer any assets.

The values used to compute the funding and rollover fees are completely under control of the message sender, which is trusted not to act maliciously in the security model of the contract.

Inputs

The incoming message has the following structure:

```
message FeedPrices {
    trxId: Int as uint64;
    lpFundingFeeGrowth: Int as coins;
    rolloverFeeGrowth: Int as coins;
    prices: map<Int as uint16, Int as uint128>;
}
```

- `trxId`
  - **Validation**: None.
  - **Impact**: Identifier for the message — does not affect the handler logic.
- `lpFundingFeeGrowth`
  - **Validation**: None.
  - **Impact**: Used to compute the updated funding fees.
- `rolloverFeeGrowth`
  - **Validation**: None.
  - **Impact**: Used to compute the updated rollover fees.
- `prices`
  - **Validation**: None.
  - **Impact**: Dictionary with updated prices, replacing all the previously stored prices.

Test coverage

This function is not exercised by the test suite.

Messages: `CreateCompensate` and `ExecuteOrCancelCompensate`

The `CreateCompensate` message allows the executor to register a `Compensate` request that is executable after three days using `ExecutorOrCancelCompensate`. It may be canceled at any time.

Only the compensator may create, execute, or cancel these requests.

Executing the `Compensate` request performs the following actions:

- Deletes an `orderId` from `liquidityOrders` or `perpOrders` and `perpOrderEx`, if `orderType` is not null
- Transfers an arbitrary amount of jettons to an arbitrary address, if `refundReceiver` is not null and `refundAmount` is nonzero
- Transfers an arbitrary fee to the compensate receiver, if `executionFeeReceiver` is not null and `executionFee` is nonzero
- Deletes the `compensateId` (request ID) from the map that stores the requests

Inputs

The incoming messages have the following structures:

```
message CreateCompensate {
    orderType: Int? as uint8;
    orderId: Int as uint64;
    trxId: Int as uint64;
    refundReceiver: Address?;
    refundAmount: Int as coins;
    executionFeeReceiver: Address?;
    executionFee: Int as coins;
}
```

- `orderType`
  - **Validation**: None.
  - **Impact**: Determines which map the `orderId` should be removed from, if any.
- `orderId`
  - **Validation**: None.
  - **Impact**: Determines which value to remove from the map chosen by `orderType`. If the order does not exist, the request can still be created, but when executed, nothing will be removed from the map.
- `trxId`
  - **Validation**: None.
  - **Impact**: Identifier for the message — does not affect handler logic.
- `refundReceiver`
  - **Validation**: None.
  - **Impact**: If not null and `refundAmount` is greater than zero, the executor will transfer `refundAmount` to this address. If null, no transfer occurs.
- `refundAmount`
  - **Validation**: None.
  - **Impact**: If greater than zero and `refundReceiver` is not null, the executor will

transfer this amount to `refundReceiver`. If zero, no transfer occurs.

- `executionFeeReceiver`
  - **Validation**: None.
  - **Impact**: If not null and `executionFee` is greater than zero, the executor will transfer `executionFee` to this address. If null, no transfer occurs.
- `executionFee`
  - **Validation**: None.
  - **Impact**: If greater than zero and `executionFeeReceiver` is not null, the executor will transfer this amount to `executionFeeReceiver`. If zero, no transfer occurs.

```
message ExecuteOrCancelCompensate {
    isCancel: Bool;
    compensateId: Int as uint64;
    trxId: Int as uint64;
}
```

- `isCancel`
  - **Validation**: None.
  - **Impact**: If `true`, the request is canceled. If `false`, the request is executed.
- `compensateId`
  - **Validation**: The `unlockTime` associated with this `compensateId` must be less than the current time.
  - **Impact**: Specifies which request to execute or cancel. The request is always removed from the map on successful execution of the handler.
- `trxId`
  - **Validation**: None.
  - **Impact**: Identifier for the message — does not affect handler logic.

Test coverage

The tests cover the basic logic of the `CreateCompensate` and execute path of the `ExecuteOrCancelCompensate` messages.

We recommend adding tests to cover negative cases (e.g., where parameter validation fails) to prevent regressions.

Intended branches

- ☑ `CreateCompensate` successfully creates a compensation order.
- ☑ `ExecuteOrCancelCompensate` successfully executes a compensation order.

Negative behavior

- ☐ Sender is not the compensator.
- ☐ The message value is insufficient to cover for gas fees.
- ☐ The required unlock time has not elapsed (`ExecuteOrCancelCompensate` only).
- ☐ The given compensate ID does not exist (`ExecuteOrCancelCompensate` only).

Message: `JettonTransferNotification`

This message is sent by Jetton wallets to inform the pool of an incoming transfer. The message is rejected if the pool is in the stopped state.

The message must be sent by either the `tlpWallet` or the `jettonWallet` address.

If the sender is the `jettonWallet`, the message payload `forward_payload` field is read to determine which operation should be performed on the incoming assets, among the following:

- `OP_CREATE_INCREASE_LP_POSITION_ORDER`
- `OP_CREATE_INCREASE_PERP_POSITION_ORDER`
- `OP_INCREASE_AUM`

If the message is sent from the `tlpWallet` address, the operation is not read from the message `forward_payload`, but rather it is assumed that a new liquidity order should be created.

If the sender is not `jettonWallet` or `tlpWallet` or the operation is invalid, the incoming jettons are refunded and processing stops.

Inputs

The incoming message has the following structure:

```
message(0x7362d09c) JettonTransferNotification {
    query_id: Int as uint64;
    amount: Int as coins;
    sender: Address;
    forward_payload: Slice as remaining;
}
```

- `query_id`
  - **Validation**: None.
  - **Impact**: Arbitrary request ID — does not affect the handler logic.
- `amount`
  - **Validation**: Must be sufficient to cover the fees required to execute the specified order.

    - **Impact**: Amount of jettons transferred.
- `sender`
    - **Validation**: Validation is operations-dependent — more details below.
    - **Impact**: Address of the sender of the jettons.
- `forward_payload`
    - **Validation**: Validation is operations-dependent — more details below.
    - **Impact**: Slice containing the operation to be performed.

Operation: `OP_CREATE_INCREASE_LP_POSITION_ORDER`

This operation requests the creation of an order to increase the LP position of the sender.

The operation payload also specifies an `executionFee`, which must be greater than the `lpMinExecutionFee` configured in the contract storage (otherwise the jettons are returned and processing stops).

The incoming message must have sufficient TON to cover the `executionFee` plus the `createLiquidityOrderGas` amount needed to cover gas expenses for creating the order (otherwise the jettons are refunded and processing stops).

The details of the order are stored in the contract storage, including the owner of the order (the owner of the incoming jettons), the amount of jettons associated with the order, the execution fee paid, and the time the order is inserted.

The order details will be retrieved from contract storage when the order is executed by an executor.

Operation: `OP_CREATE_INCREASE_PERP_POSITION_ORDER`

This operation requests the creation of an order to increase the perpetual position of the sender.

The operation payload includes the following parameters:

- `marginDelta`: The amount to be added to the position collateral.
- `executionFee`: The fee to be paid to the executor executing the order.
- `isMarket`: The boolean specifying whether this is a market or limit order.
- `tokenId`: The identifier of the asset for which the user wants to take a position in.
- `isLong`: The boolean specifying whether the user wants to take a long or short position.
- `sizeDelta`: The size of the order.
- `triggerPrice`: The trigger price, only relevant for limit orders (`isMarket` is false).
- `requestTime`: The arbitrary number specified by the user, intended to identify the time of the price point the user wants to use for the order. For off-chain entity use only.

The payload can also include optional take-profit/stop-loss conditions:

- `tpSize`: The size of the take-profit order.
- `tpPrice`: The price at which the take-profit condition triggers.
- `slSize`: The size of the stop-loss order.
- `slPrice`: The price at which the stop-loss condition triggers.

The incoming message must have sufficient TON to cover the `executionFee` plus the `createPerpOrderGas` amount configured in the contract storage, which is needed to cover the gas spent to create the order (the incoming jettons are returned, and processing stops otherwise).

The `executionFee` must be sufficient to cover the execution fee for the order plus the execution fee for one or two additional orders if the take-profit and/or stop-loss conditional orders are requested. If the `executionFee` is insufficient, the incoming jettons are refunded and processing stops.

The requested order is then persisted in the contract storage. Each order is assigned a unique incremental ID used to index the order details in the `perpOrders` map. An event is emitted to signal the order creation to off-chain observers.

If any take-profit/stop-loss conditions are requested, they are also recorded in the `perpOrderExs` map.

Finally, the `totalExecutionFee` available to be paid out to the executors is incremented by the order `executionFee`, and any excess TON is returned to the message sender.

Operation: `OP_INCREASE_AUM`

This operation can only be requested by the manager address. The operation increases the `globalLPFund` variable, which tracks the total amount of assets provided by the LPs plus the current unrealized PNL. The variable is increased by the amount received.

Test coverage

Intended branches

- `OP_CREATE_INCREASE_LP_POSITION_ORDER`
  - ☑ Successfully create a new order to increase the LP position.
- `OP_CREATE_INCREASE_PERP_POSITION_ORDER`
  - ☑ Successfully create a new order to increase the perpetual position.
  - ☑ Successfully create a new order to increase the perpetual position with take-profit and stop-loss conditions.
- `OP_INCREASE_AUM`
  - ☑ Successfully increase the AUM of the pool.

Negative behavior

- All opcodes

- ☐ Contract is stopped.
- ☐ Caller is not the `tlpWallet` or `jettonWallet` address.

- `OP_CREATE_INCREASE_LP_POSITION_ORDER`
  - ☐ Insufficient execution fee is provided.
  - ☐ Insufficient gas is provided.
- `OP_CREATE_INCREASE_PERP_POSITION_ORDER`
  - ☐ Insufficient execution fee is provided.
  - ☐ Insufficient gas is provided.
- `OP_INCREASE_AUM`
  - ☐ Original caller is not the manager address.

Message: `CancelLiquidityOrder`

This message allows the owner of an order or an executor to cancel a liquidity order that is inserted but not pending.

The message must have at least `self.cancelLiquidityOrderGas + self.transferJettonGas` TON to cover for the gas fees required to cancel the order. The order must exist and must not be in a pending state.

An executor can cancel the order at any time, while the order owner can only cancel the order after it has been pending for at least the `orderLockTime` configured for the pool. The order is deleted from the contract storage, and the collateral associated with the order is returned to the order owner.

The execution fees paid for the order are also returned. The recipient of the execution fees can be optionally specified in the incoming message. If unspecified, the execution fees will be sent to the message sender. Finally, an event signals the order cancellation is emitted, and the excess TON is refunded to the message sender.

Inputs

The incoming message has the following structure:

```
message CancelLiquidityOrder {
    orderId: Int as uint64;
    trxId: Int as uint64;
    executionFeeReceiver: Address?;
}
```

- `orderId`
  - **Validation**: Must be an existing pending order owned by the sender (unless

the sender is an executor). Other checks are performed on the order (see earlier section).
    - **Impact**: Identifies the order to be canceled.
- `trxId`
    - **Validation**: None.
    - **Impact**: Request ID to be used for the jetton transfer refunding the order-execution fees. Not involved in the contract logic.
- `executionFeeReceiver`
    - **Validation**: None.
    - **Impact**: Optional address that will receive the execution-fees refund. If unspecified, defaults to the sender of the message.

Test coverage

Intended branches

- ☑ Successfully cancel a liquidity order and transfer fees as expected.

Negative behavior

- ☐ Insufficient gas to complete the transaction.
- ☐ Cancel an order that does not exist.
- ☐ Cancel an order that is pending.
- ☐ Cancel an order that is not owned by the sender, and the sender is not a trusted executor.
- ☐ Cancel an order before the `orderLockTime` has elapsed, and the sender is not a trusted executor.

Message: `ExecuteLiquidityOrder`

This operation allows to execute a liquidity order. The message sender must be an allowlisted executor.

The message must carry enough TON to pay for the estimated gas fees, which vary depending on whether the order is to increase the LP position (needing to send one mint message to mint LP jettons) or decrease the position (needing to send one burn message to burn LP jettons and one transfer message to transfer the underlying asset jetton back to the owner).

The message contains a map of asset prices, which is used to compute the updated global PNL. This map is not validated in any way, and the executor is trusted to provide accurate prices for all the assets in which the pool has an open position.

Next, funding and rollover fees are charged and credited to the protocol owners and LPs, increasing the `globalLPFund` and `protocolTradingFee` storage variables. The values by which the

fees are increased are controlled by the message sender and are not validated in any way.

Finally, the order is executed, with varying actions depending on whether the order was to increase or decrease the LP position.

Orders to increase the LP position require minting new LP tokens representing the position. The contract asserts that the amount of assets managed by the pool after the mint would not exceed the configured `maxLpNetCap`. If the total amount surpasses the configured threshold, execution reverts, not canceling the order and not returning the deposited jettons. The jettons are not lost, since the user can cancel the order after a lock period. Next, the LP jettons representing the position are minted to the user posting the order. The amount of LP jettons is computed proportionally, as the amount of asset jettons deposited by the user multiplied by the ratio between the total asset jettons deposited and the total amount of LP jettons in circulation.

Orders to decrease the LP position require burning the LP jettons corresponding to the position being reduced and returning the asset jettons corresponding to the position-size decrease to the order owner. The amount of asset jettons returned to the user is the amount of burned LP jettons multiplied by the ratio between the amount of asset jettons managed by the pool divided by the total LP jetton supply in circulation.

Finally, the order is deleted from the contract storage, the execution fees are sent to the receiver (which can be specified in the message and defaults to the message sender otherwise), an event is emitted to aid off-chain observers detect the execution, and any excess TON is returned to the message sender.

Inputs

The incoming message has the following structure:

```
message ExecuteLiquidityOrder {
    orderId: Int as uint64;
    trxId: Int as uint64;
    executionFeeReceiver: Address?;
    prices: map<Int as uint16, Int as uint128>;
    lpFundingFeeGrowth: Int as coins;
    rolloverFeeGrowth: Int as coins;
}
```

- `orderId`
  - **Validation**: Must refer to an existing nonpending order.
  - **Impact**: Identifies the order to be canceled.
- `trxId`
  - **Validation**: None.
  - **Impact**: Request ID to be used for the jetton transfer if the order is to decrease the position.

- `executionFeeReceiver`
  - **Validation**: None.
  - **Impact**: Optional address that will receive the execution-fees refund. If unspecified, defaults to the sender of the message.
- `prices`
  - **Validation**: None.
  - **Impact**: Map of asset prices — used to update the global PNL.
- `lpFundingFeeGrowth`
  - **Validation**: None.
  - **Impact**: Parameter used to charge funding fees.
- `rolloverFeeGrowth`
  - **Validation**: None.
  - **Impact**: Parameter used to charge rollover fees.

Test coverage

Intended branches

- ☑ Successfully execute a liquidity order. There are tests for increasing and decreasing the LP position.

Negative behavior

- ☐ Caller is not a trusted executor.
- ☐ Order does not exist.
- ☐ Order is pending. Note that it is logically impossible in the contract for this to happen, because the order is atomically deleted from the contract storage after setting `isPending` to true.
- ☐ Insufficient gas supplied to execute the order.
- ☑ Insufficient quota of liquidity to execute the LP order, when the order is to increase the LP position.

Message: `CreateDecreasePerpOrder`

This operation allows to add a market order to decrease a perpetual position. The message is rejected if the pool is in the stopped state.

The message must carry sufficient TON to cover for the minimum `perpMinExecutionFee` plus the `createPerpOrderGas` gas-cost parameters configured in the contract.

The order is assigned a unique incremental ID and inserted into the `perpOrders` map, ready to be executed.

Finally, an event signalling the order to off-chain observers is emitted, the `totalExecutionFee` amount of total fees collected to be paid to executors is increased, and any excess TON is returned to the message sender.

Inputs

The incoming message has the following structure:

```
message CreateDecreasePerpOrder {
    executionFee: Int as coins;
    tokenId: Int as uint16;
    isLong: Bool;
    marginDelta: Int as coins;
    sizeDelta: Int as coins;
    triggerPrice: Int as uint128;
    trxId: Int as uint64;
    requestTime: Int as uint32;
}
```

- `executionFee`
  - **Validation**: Must be sufficiently large to cover the minimum execution fee.
  - **Impact**: The fee to be paid to the executor for executing the order.
- `tokenId`
  - **Validation**: None.
  - **Impact**: Specifies the token for which the order is being created.
- `isLong`
  - **Validation**: None.
  - **Impact**: Indicates whether the order is for a long or short position.
- `marginDelta`
  - **Validation**: None (validation performed when the order is executed).
  - **Impact**: Specifies the amount the position margin is to be decreased by.
- `sizeDelta`
  - **Validation**: None (validation performed when the order is executed).
  - **Impact**: Specifies the amount the position size is to be decreased by.
- `triggerPrice`
  - **Validation**: None.
  - **Impact**: Allows to specify a trigger condition for the order, which becomes executable only if the price crosses this threshold (above for long positions, below for short).

- `trxId`
    - **Validation**: None.
    - **Impact**: Not used by the on-chain contract.
- `requestTime`
    - **Validation**: None.
    - **Impact**: Not used by the on-chain contract.

Test coverage

Intended branches

- ☑ Successfully create a decrease-short-position order.
- ☑ Successfully create a decrease-long-position order.

Negative behavior

- ☐ Insufficient `ctx.value`.
- ☐ Insufficient `executionFee`.
- ☐ Protocol is stopped.

Message: `CreateTpSlPerpOrder`

This operation allows to add conditional take-profit and/or stop-loss perpetual position orders. The message is rejected if the pool is in the stopped state.

The message must carry sufficient TON to cover the order `executionFee` and the `createPerpOrderGas` fees for all the conditions specified. The fees must thus be paid once if only a take-profit or stop-loss order is requested, and they must be paid twice if both conditions are specified. A condition is considered not requested if the size or trigger price is zero. Technically, it is possible to not specify either condition. This is not an issue for the pool, as it only causes the user to waste gas fees.

Next, the requested take-profit and/or stop-loss orders are inserted. Each order is assigned a unique incremental ID, used as a key to store the order details into the `perpOrders` map. An event is emitted for each inserted order to aid off-chain observers.

Finally, the `totalExecutionFee` amount of total fees collected to be paid to executors is increased, and any excess TON is returned to the message sender.

Inputs

The incoming message has the following structure:

```
message CreateTpSlPerpOrder {
    executionFee: Int as coins;
    tokenId: Int as uint16;
    isLong: Bool;
    tpSize: Int as coins;
    tpPrice: Int as uint128;
    slSize: Int as coins;
    slPrice: Int as uint128;
    trxId: Int as uint64;
    requestTime: Int as uint32;
}
```

- `executionFee`
  - **Validation**: Must be sufficient to cover the `self.perpMinExecutionFee` for however many orders will be created.
  - **Impact**: The fee to be paid to the executor for executing the order.
- `tokenId`
  - **Validation**: None.
  - **Impact**: Identifier of the asset for which the user wants to take a position in.
- `isLong`
  - **Validation**: None.
  - **Impact**: Whether the user wants to take a long or short position. If `true`, for take-profit orders, `triggerAbove` will be `true`, and for stop-loss orders, `triggerAbove` will be `false`. If `false`, the opposite will be true.
- `tpSize`
  - **Validation**: None.
  - **Impact**: The size of the take-profit order. This is the amount of the asset that will be sold if the take profit is triggered.
- `tpPrice`
  - **Validation**: None.
  - **Impact**: The price at which the take-profit order will be triggered.
- `slSize`
  - **Validation**: None.
  - **Impact**: The size of the stop-loss order. This is the amount of the asset that will be sold if the stop loss is triggered.
- `slPrice`
  - **Validation**: None.
  - **Impact**: The price at which the stop-loss order will be triggered.
- `trxId`

- **Validation**: None.
- **Impact**: Identifier for the message — does not affect handler logic.

- `requestTime`
  - **Validation**: None.
  - **Impact**: N/A.

Test coverage

Intended branches

- ☑ Successfully create take-profit/stop-loss order.

Negative behavior

- ☐ Insufficient `ctx.value`.
- ☐ Insufficient `executionFee`.
- ☐ Protocol is stopped.

Message: `CancelPerpOrder`

This operation allows the owner of an order or an executor to cancel an inserted order.

An executor can cancel an order at any time. The order owner can only cancel market orders after they have been pending for the `orderLockTime` configured for the pool.

The message must carry enough TON to cover for the `cancelPerpOrderGas` gas fees plus the `transferJettonGas` costs (configured in the contract) if the order being canceled was to increase a position, since canceling orders increasing a position requires returning the jettons deposited when the order was created.

Orders to increase a perpetual position can also have associated take-profit/stop-loss conditions; in case they do, the additional execution fees paid in advance are also refunded to the owner of the order.

Next, any take-profit/stop-loss execution fees deposited in advance as well as the collateral deposited for position-increase orders are refunded to the order owner.

Execution fees paid for the order itself (not for the take-profit/stop-loss conditions) are sent to the execution-fee receiver address optionally specified in the message. If unspecified, this address defaults to the message sender.

Finally, the `totalExecutionFee` amount of total collected fees collected to be paid to executors is decreased to account for the reimbursements, an event signalling off-chain observers of the cancellation is emitted, and any excess TON is returned to the message sender.

Inputs

The incoming message has the following structure:

```
message CancelPerpOrder {
    executionFeeReceiver: Address?;
    orderId: Int as uint64;
    trxId: Int as uint64;
}
```

- `executionFeeReceiver`
  - **Validation**: None.
  - **Impact**: Optional address that will receive the execution-fees reimbursement.
- `orderId`
  - **Validation**: Must be an existing, nonpending order (owner by the message sender if not an executor).
  - **Impact**: Identifies the order to be canceled.
- `trxId`
  - **Validation**: None required.
  - **Impact**: Used as a message identifier in the transfer message returning jettons to the order owner — does not affect the contract logic.

Test coverage

Intended branches

- ☑ Successfully cancels a perpetual order and transfers fees as expected.

Negative behavior

- ☐ Insufficient gas to complete the transaction.
- ☐ Cancel an order that does not exist.
- ☐ Cancel an order that is pending.
- ☐ Cancel an order that is not owned by the sender, and the sender is not a trusted executor.
- ☐ Cancel an order before the `orderLockTime` has elapsed, and the sender is not a trusted executor.

Message: `ExecutePerpOrder`

This operation allows one of the trusted executors to execute a previously requested perpetual order. Orders can only be executed if they involve a registered and enabled asset.

The message must carry enough TON to cover for the `executePerpOrderGas` and `transferJettonGas` amounts configured in the contract.

The price for the asset is provided in the message. This price is not verified and is fully controlled by the executor. Limit orders are checked to ensure the price provided in the message satisfies the trigger condition.

Additionally, orders increasing the position size could have specified optional take-profit and/or stop-loss conditions; these conditions are turned into orders, which are inserted in the book at this moment.

The user long or short position size and margin are adjusted to increase or decrease them according to the order type and parameters. If the order is for decreasing a position, the size and margin changes are limited so that the position size and margin do not become negative (accounting for fees in the case of the margin).

The global net position is also adjusted accordingly; the global net position represents the imbalance between long and short positions taken by the traders, and it is in effect the net exposure taken collectively by the liquidity providers, which is not compensated between the perpetual traders. The global position of the same type of the order (long or short) is first reduced. If the order is bigger than the current global net position, the global net position takes the remaining size, with inversed side.

The following fees are applied:

- Trading fees are computed as a percentage of the position size increase or decrease, configured per token (`tokenConfig.tradingFeeRate`). A per-token configurable percentage of the trading fees (`tokenConfig.lpTradingFeeRate`) is passed to the LPs, while the rest is credited to the protocol. The protocol also receives a configurable percentage (`normalPositionShareRate`) of the PNL change realized by the LPs, only if the PNL is positive.
- Funding fees are always added to the position margin, crediting the user. The funding fee is computed as $(FundingFeeGrowth - CurrentPositionFundingFeeGrowth) * PositionSize$.
- Rollover fees are always taken from the position margin, charging the user. The fee is computed as $(RolloverFeeGrowth - CurrentPositionRolloverFeeGrowth) * PositionSize$.

We note that the executor fully controls the $FundingFeeGrowth$ and $RolloverFeeGrowth$ parameters and can therefore decide the fee values, even turning them negative. The fee mechanisms are opaque and controlled by the executors; we did not receive details on how these values are computed off chain nor design notes detailing the rationale behind the fees.

Orders to increase a position size and orders to decrease a position, leaving it nonempty, are subject to margin and leverage checks, ensuring the user position does not cross the risk

thresholds configured for the asset. If the health checks fail, the order cannot be executed and the state changes are reverted.

Orders to decrease a position also trigger a payout of the requested margin decrease.

Finally, an event signals the order execution is emitted, the order is deleted from storage, execution fees are sent to the receiver (optionally specified in the message, defaulting to the message sender), and the excess TON attached with the message is refunded to the sender.

Inputs

The incoming message has the following structure:

```
message ExecutePerpOrder {
    executionFeeReceiver: Address?;
    orderId: Int as uint64;
    trxId: Int as uint64;
    tokenId: Int as uint16;
    price: Int as uint128;
    premiumRate: Int as int32;
    fundingFeeGrowth: Int as int128;
    rolloverFeeGrowth: Int as int128;
}
```

- `executionFeeReceiver`
  - **Validation**: None.
  - **Impact**: Optional address that will receive the execution fees.
- `orderId`
  - **Validation**: Must be an existing, nonpending order.
  - **Impact**: Identifies the order to be executed.
- `trxId`
  - **Validation**: None required.
  - **Impact**: Does not affect the core contract logic.
- `tokenId`
  - **Validation**: None.
  - **Impact**: Not used and not necessary — the token ID is taken from the order stored in the contract.
- `price`
  - **Validation**: None.
  - **Impact**: Price of the asset of the order, used to compute position values, PNLs, and fees.

- `premiumRate`
    - **Validation**: None.
    - **Impact**: Not used.
- `fundingFeeGrowth`
    - **Validation**: None.
    - **Impact**: Used to compute the funding fees.
- `rolloverFeeGrowth`
    - **Validation**: None.
    - **Impact**: Used to compute rollover fees.

Test coverage

Intended branches

- ☑ Execute increase perp position normally.
- ☑ Execute decrease perp position normally.
- ☑ Execute close perp position normally.

Negative behavior

- ☐ User is not a trusted executor.
- ☐ Order does not exist.
- ☐ Insufficient gas is provided to complete the transaction.
- ☐ Order is pending. Note that it is logically impossible in this contract for the order to be pending.
- ☐ Trigger price has not been reached, if applicable.
- ☐ Token does not exist in the `tokenConfigs`, when increasing the perpetual position. Note that it is possible for users to create orders using any arbitrary `tokenId` — including those that are not registered in the configuration. These orders can still be canceled but not executed, and therefore, we do not consider this to be an issue.
- ☐ Token does not exist in the `tokenConfigs` when decreasing the perpetual position. Please see our note above regarding the `tokenId` parameter.
- ☐ Token is disabled when increasing the perpetual position.
- ☐ Token is disabled when decreasing the perpetual position.
- ☐ Attempting to decrease a position whose size is not greater than zero (`require(perpPosition.size > 0, "position not exist")`).
- ☐ Margin rate is too high when increasing the perpetual position.
- ☐ Insufficient margin when decreasing the perpetual position, after accounting for fees in the realized PNL.
- ☐ Margin rate is too high when decreasing the perpetual position.
- ☐ Leverage is too high after increasing the perpetual position.

- ☐ Leverage is too high after decreasing the perpetual position.

Message: `LiquidatePerpPosition`

As described by the comment above the message handler,

```
/**
 * This function is called when a LiquidatePerpPosition message is received.
 * It performs various checks, adds a new order to the order book, and sends an
    update position message to the pool.
 *
 * @param msg The LiquidatePerpPosition message containing information about
    the liquidation.
 */
```

Inputs

The incoming message has the following structure:

```
message LiquidatePerpPosition {
    liquidationFeeReceiver: Address?;
    tokenId: Int as uint16;
    account: Address;
    isLong: Bool;
    trxId: Int as uint64;
    price: Int as uint128;
    premiumRate: Int as int32;
    fundingFeeGrowth: Int as int128;
    rolloverFeeGrowth: Int as int128;
}
```

- `liqudationFeeReceiver`
  - **Validation**: None.
  - **Impact**: Not used by the contract, liquidation fees are not collected. The name implies this should determine which address should receive the liquidation fee. If not provided in the message, the caller's address is used.
- `tokenId`
  - **Validation**: Must have an existing configuration in `tokenConfigs`.
  - **Impact**: Specifies which `tokenId` is being liquidated.
- `account`

- **Validation**: Must have an open position for the `tokenId`.
- **Impact**: Specifies which position is being liquidated.

- `isLong`
  - **Validation**: None.
  - **Impact**: Specifies the direction the position is being liquidated.
- `trxId`
  - **Validation**: None.
  - **Impact**: Not used by the on-chain contract.
- `price`
  - **Validation**: None.
  - **Impact**: The trade price to use when calculating the PNL.
- `premiumRate`
  - **Validation**: None.
  - **Impact**: Not used by the on-chain contract.
- `fundingFeeGrowth`
  - **Validation**: None.
  - **Impact**: Used to compute the funding fees.
- `rolloverFeeGrowth`
  - **Validation**: None.
  - **Impact**: Used to compute rollover fees.

Test coverage

Intended branches

- ☑ Successfully liquidate long positions.
- ☑ Successfully liquidate short positions.

Negative behavior

- ☐ Insufficient gas to process the message.
- ☐ Caller is not a trusted executor.
- ☐ The `tokenId` does not exist in the `tokenConfigs`.
- ☐ The token is disabled.
- ☐ The account does not have an open position for the `tokenId`.
- ☐ The margin is too high to liquidate.

Message: `ADLPerpPosition`

This message creates an order to decrease a perpetual position via ADL mechanics, typically initiated when the off-chain executor's risk thresholds are breached and margin health must be preserved.

This message must be dispatched by a trusted executor. Note that all parameters are fully controlled, and the executor is — as always — entirely trusted and responsible for calling this function at the proper time and correctly calculating and providing parameters.

Inputs

The incoming message has the following structure:

```
message ADLPerpPosition {
    tokenId: Int as uint16;
    account: Address;
    isLong: Bool;
    marginDelta: Int as coins;
    sizeDelta: Int as coins;
    trxId: Int as uint64;
    price: Int as uint128;
    premiumRate: Int as int32;
    fundingFeeGrowth: Int as int128;
    rolloverFeeGrowth: Int as int128;
}
```

- `tokenId`
  - **Validation**: Must be a valid `tokenId` in the contract's `tokenConfigs` map.
  - **Impact**: Specifies the token for which the position is being adjusted.
- `account`
  - **Validation**: Must have an existing position.
  - **Impact**: Specifies the account whose position is being adjusted.
- `isLong`
  - **Validation**: None.
  - **Impact**: Specifies whether the position is long or short.
- `marginDelta`
  - **Validation**: None — inherently constrained to an unsigned int and limited by the position's margin.
  - **Impact**: The amount of margin to decrease the position by.
- `sizeDelta`

- **Validation**: None.
- **Impact**: The amount of position size to decrease.

- `trxId`
  - **Validation**: None.
  - **Impact**: Not used by the on-chain contract.
- `price`
  - **Validation**: None.
  - **Impact**: Used to calculate the PNL to realize.
- `premiumRate`
  - **Validation**: None.
  - **Impact**: Unused by the on-chain contract.
- `fundingFeeGrowth`
  - **Validation**: None.
  - **Impact**: Used to compute the funding fees.
- `rolloverFeeGrowth`
  - **Validation**: None.
  - **Impact**: Used to compute rollover fees.

Test coverage

Intended branches

- ☑ Can successfully auto-deleverage a position.

Negative behavior

- ☐ Insufficient gas to process the message.
- ☐ Caller is not a trusted executor.
- ☐ The `tokenId` is not configured in the contract.
- ☐ The token is disabled in the contract.
- ☐ The `account` does not have an existing position.
- ☐ The position does not have sufficient margin after realizing the PNL and accounting for fees.
- ☐ The margin rate is too high.
- ☐ Leverage is too high after the position is auto-deleveraged.

Message: `UpdateBaseConfig`

This operation allows the owner to update the configuration of the contract, including

- the `multisig` address (therefore, the owner inherits all capabilities of the multisig, manager, compensator, and claimer, which can be set using the `SetManager` message),
- the `tlpJetton` address,
- the `tlpWallet` address,
- the `jettonWallet` address,
- which addresses are considered trusted executors, and
- execution fees, `minStorageReserve`, and all gas parameters.

Inputs

The incoming message has the following structure:

```
struct GasConfig {
    mintJettonGas: Int as coins;
    burnJettonGas: Int as coins;
    transferJettonGas: Int as coins;
    createPerpOrderGas: Int as coins;
    cancelPerpOrderGas: Int as coins;
    executePerpOrderGas: Int as coins;
    createLiquidityOrderGas: Int as coins;
    cancelLiquidityOrderGas: Int as coins;
    executeLiquidityOrderGas: Int as coins;

    minStorageReserve: Int as coins;
    lpMinExecutionFee: Int as coins;
    perpMinExecutionFee: Int as coins;
}

struct ExecutorConfig {
    executors: map<Address, Bool>;
}

struct ContractConfig {
    multisig: Address;
    tlpJetton: Address;
    tlpWallet: Address;
    jettonWallet: Address;
}

message UpdateBaseConfig {
    gasConfig: GasConfig?;
    executorConfig: ExecutorConfig?;
    contractConfig: ContractConfig?;
}
```

- `gasConfig`
  - **Validation**: None.
  - **Impact**: Specifies the new gas parameters to use.
- `executorConfig`
  - **Validation**: None.
  - **Impact**: Specifies the new trusted executors to enable or disable.
- `contractConfig`
  - **Validation**: None.
  - **Impact**: Specifies the new `multisig`, `tlpJetton`, `tlpWallet`, and `jettonWallet` addresses to use.

Test coverage

Intended branches

- ☐ Successfully updates the configuration. Note that there is a test to confirm that the `UpdateBaseConfig` is called successfully and sends a reply to the caller, but it does not assert that all of the contract configuration parameters are properly updated as expected.

Negative behavior

- ☐ Caller is not the owner.

Message: `SetManager`

This operation allows the multisig to change the `manager`, `compensator`, and `claimer` addresses. There is no timelock on this operation.

Note that the owner can change the multisig address using the `UpdateBaseConfig` message, and therefore, the owner inherently has all of the capabilities that the multisig, manager, compensator, and claimer also have.

Inputs

The incoming message has the following structure:

```
message SetManager {
    manager: Address;
    compensator: Address;
    claimer: Address;
}
```

- `manager`
    - **Validation**: None.
    - **Impact**: The new manager address.
- `compensator`
    - **Validation**: None.
    - **Impact**: The new compensator address.
- `claimer`
    - **Validation**: None.
    - **Impact**: The new claimer address.

Test coverage

Intended branches

- ☐ Successfully update the `manager`, `compensator`, and `claimer` addresses. Note that there is a multisig test that uses `SetManager` as the proposal, but it does not confirm that the addresses were properly updated.

Negative behavior

- ☐ Caller is not the multisig.

Message: `JettonUpdateContent`

This operation simply forwards an arbitrary cell to the `tlpJetton` address, as long as the caller is the contract owner.

In the jetton/jetton_master.tact file, the `JettonUpdateContent` message handler simply checks that the caller is the pool then updates the storage variable `jetton_content` with the new value.

This data is used for TEP-64 ↗.

Inputs

The incoming message has the following structure:

```
message JettonUpdateContent {
    jetton_content: Cell;
}
```

- `jetton_content`:

- **Validation**: None.
- **Impact**: This is the cell containing the new metadata for the `tlpJetton` to use.

Test coverage

Intended branches

- ☐ Successfully update the token metadata.

Negative behavior

- ☐ Caller is not the pool owner.

5.2. Contract: TON Pool

The TON pool contract is largely identical to the USDT pool contract. The following noteworthy but trivial differences between the two can be observed:

- the TON pool has no `ClaimProtocolFee` message; it is replaced by the `WithdrawFee` message
- the TON pool `CancelLiquidityOrder`, `ExecuteLiquidityOrder`, `CancelPerpOrder`, `ExecutePerpOrder`, `LiquidatePerpPosition` message do not have a field to specify the fee receiver
- the following fee parameters have no default value declared in the TON pool contract:
  - `lpMinExecutionFee`
  - `perpMinExecutionFee`
  - `minStorageReserve`

Additionally, since the TON pool uses native TON currency instead of USDT jettons as collateral for orders, handling of incoming assets and the flow for initiating orders to increase LP and perpetual positions have significant differences. Unlike the USDT pool, the TON pool `JettonTransferNotification` handler accepts messages from the `tlpWallet`, the wallet for the LP jetton controlled by the pool. Messages from this address are automatically assumed to request a decrease of the liquidity position. All other messages are rejected, returning the incoming jettons to the sender.

Jetton operations to request an increase of a position were replaced by messages as follows:

- `OP_CREATE_INCREASE_LP_POSITION_ORDER` for the USDT pool is replaced by the `CreateAddLiquidityOrder` message in the TON pool
- `OP_CREATE_INCREASE_PERP_POSITION_ORDER` for the USDT pool is replaced by the `CreateIncreasePerpOrder` message in the TON pool
- `OP_INCREASE_AUM` for the USDT pool is replaced by `IncreaseAum` in the TON pool

The TON pool also calculates execution fees differently, since executing orders which use TON as

collateral instead of USDC jettons requires different messages (and different costs) to transfer the collateral.

Lastly, we note that the TON pool `LiquidatePerpPosition` message contains a `liquidationFeeReceiver` field which is not actually used to send any liquidation fees. We could not determine whether this is a remnant of a previous version or the skeleton of a future change, but do not consider this an exploitable issue on its own.

6. Assessment Results

During our assessment on the scoped Tradoor contracts, we discovered seven findings. No critical issues were found. One finding was of high impact, one was of medium impact, one was of low impact, and the remaining findings were informational in nature. The high, medium, and low severity issues have been remediated, and the corresponding fix commit links are included in this report for reference.

While our review did not identify any critical severity issues, we believe there are still areas that would benefit from further refinement before the code is moved to production. We recommend developing a more comprehensive test suite and adding clear documentation to improve maintainability, support future reviews, and build trust with end users.

Due to the time-boxed nature of security assessments, there are inherent limitations to the depth and breadth of coverage possible. In this assessment, limited documentation — both within the code and in external resources — made it challenging to fully understand certain design decisions, the intended threat model, and the rationale behind some of the mathematics.

Complex mathematical logic was not clearly explained, which required significant time to interpret. In several cases, this led us down investigative paths that ultimately proved unnecessary, consuming significant time. More thorough documentation could help avoid such detours and enable a more focused and efficient review.

Additionally, external documentation was not provided at the time of the assessment. Often, this meant there was no direction as to the roles of certain entities or the purpose of separating certain roles (e.g., owner from multisig), and resources for answering this were not available. Answering these questions does not directly aid our understanding of the code. However, without insight on the developer's intentions, observations cannot easily be made about in-code mistakes regarding the threat model. This is a significant barrier in an assessment.

In particular, it would be valuable to document:

- All non-trivial math or logic implemented by onchain contracts (pool.tact in particular)
- The roles of the protocol participants and which actions they are entrusted to perform
- The security assumptions for off-chain components

After having performed our assessment, the code we have the least coverage of is the jetton contracts. We performed a cursory review; however, the pool required significant time and attention and was prioritized.

The test suite is also not comprehensive. This reflects an early stage of development, and can hinder understanding during an assessment. In this protocol, it is especially a concern given the complexity of the pool and the mathematical logic involved. For information regarding our concerns and recommendations for the tests, please refer to section 4.1. ↗.

Given the complexity of this protocol — in addition to improving documentation and tests — we also strongly recommend a comprehensive assessment of the protocol including the offchain executor component. We also recommend setting up a bug-bounty program, to maximize the incentives for participating to coordinated bug disclosure.

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.