PixelSwap DEX

Security Assessment

December 19, 2024

Prepared for:
Runchen Sun
PixelSwap Labs Ltd.

Prepared by: **Tarun Bansal and Guillermo Larregay**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Guillermo Larregay, Consultant
guillermo.larregay@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
June 20, 2024	Pre-project kickoff call
June 28, 2024	Status update meeting #1
July 8, 2024	Status update meeting #2
July 17, 2024	Delivery of report draft
July 17, 2024	Report readout meeting
December 19, 2024	Delivery of comprehensive report

Executive Summary

Engagement Overview

PixelSwap Labs Ltd. engaged Trail of Bits to review the security of PixelSwap DEX smart contracts. The PixelSwap DEX is a constant-product-formula-based decentralized exchange developed in the Tact language to be deployed on the TON blockchain network.

A team of two consultants conducted the review from June 24 to July 16, 2024, for a total of six engineer-weeks of effort. Our testing efforts focused on finding issues impacting the availability and integrity of the target system. With full access to source code and documentation, we performed static and dynamic testing of the target smart contracts using automated and manual processes. The off-chain software components used by the system were not in scope for this review.

Observations and Impact

The PixelSwap DEX smart contracts are divided into well-defined components, both to keep track of user balances and pair reserves and to keep these different balances in sync in the limited smart contract execution environment of the TON blockchain network. However, the complex message-passing flow and error-handling mechanisms make it difficult to follow the code paths and to understand the security properties of all system components.

During the review, we found multiple high-severity issues arising from a lack of user data validation that enable theft of user deposits (TOB-PXL-3, TOB-PXL-6, and TOB-PXL-13). We also found a high-severity issue, TOB-PXL-11, related to the wrong compilation of a return value from a recursive call function.

Some issues, such as TOB-PXL-1, TOB-PXL-2, TOB-PXL-7, TOB-PXL-8, and TOB-PXL-9, are logic bugs that highlight the test suite's low coverage. Other issues are related to incorrect settlement vault balance tracking, incorrect gas checks, incorrect gas value computation, inconsistent message formatting, and the use of an incorrect formula.

We also noted that the settlement vault balance functionality is incomplete and is not used to validate balance transfers between funding and execution contracts.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that the PixelSwap Labs Ltd. team take the following steps prior to the production deployment:

- **Remediate the findings disclosed in this report.** Various issues allow attackers to drain all the assets from the protocol smart contracts. These findings should be

addressed as part of direct remediation or as part of any refactoring that may occur when addressing other recommendations.

- **Reduce the codebase complexity.** Break the messages and functions to handle different types of assets and operations to reduce the complexity of the codebase.
- **Document and follow a strategy for gas checks and computations.** Develop a strategy for calculating gas check parameters and the remaining gas value, document this strategy, and follow it for all the functions of the smart contracts.
- **Document the message and asset flow through the system.** Create diagrams for all of the message and asset flow paths through the smart contracts to identify points of failure and the reachability of an inconsistent state. Document assumptions of all the user interactions with the smart contracts and validate their correctness to find possible malicious inputs.
- **Expand and improve the test suite.** Tests should cover edge cases and unexpected inputs. Check all of the state updates in the test cases to avoid a false sense of security from incorrect or incomplete test cases. Ensure that the TON balance of the system smart contracts does not go down after a user interaction to maintain a healthy TON balance for storage rent. See appendix E for more information.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	9
Medium	2
Low	2
Informational	9
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	2
Authentication	1
Data Validation	21

Project Goals

The engagement was scoped to provide a security assessment of the PixelSwap DEX. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the access controls correctly implemented? Can a non-privileged account gain access to privileged functionality?
- Are messages correctly handled? How are complex structures decoded? Is it possible for malformed data to be decoded incorrectly, permanently affecting the system?
- Is it possible for a user to access funds that do not belong to them? Is it possible to drain a contract or otherwise get funds stuck in a contract?
- Can a malicious actor modify internal accounting? Can they affect third parties' interactions with the contracts?
- Do all contracts handle AFR and bounced messages as expected? Is it possible for a malicious actor to forge AFR messages?
- Are the contract state variables correctly initialized, and their values correctly modified during the contract's lifetime?
- Are the DEX swap and liquidity provider token minting equations correctly implemented? Could malicious actors gain value in a swap, or receive more LP tokens than the value added?
- Are all fund flows considered for adding or removing liquidity? Can funds become lost in any DEX operations?
- Do tests reflect real-life situations? Do the tests fail to consider any user flows?
- Is the system implementing standards (such as TEP-74) correctly? Can any standard-compliant contract interact with the PixelSwap DEX and expect standard behavior?

Project Targets

The engagement involved a review and testing of the following target.

PixelSwap DEX

Repository	https://github.com/nx-fi/pixelswap
Version	`5c24e13fcae51b250bd70e0a812ce69a9b3dee4a`
Type	TACT / FunC
Platform	TON

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- The scope for the current audit included the following files:
  - `contracts/pixelswap_funding.tact`
  - `contracts/pixelswap_messages.tact`
  - `contracts/pixelswap_settlement.tact`
  - `contracts/pixelswap_streampool.tact`
  - `contracts/pixelswap_streampool_data.tact`
  - `contracts/pixelswap_streampool_messages.tact`
  - `contracts/univ2_math.tact`
  - `contracts/utils/access_control_single_role_non_transferrable.tact`
  - `contracts/utils/access_control_single_role.tact`
  - `contracts/utils/address.tact`
  - `contracts/utils/afr.tact`
  - `contracts/utils/msgtools.tact`
  - `contracts/utils/option.tact`
  - `contracts/utils/ownable_2step.tact`
  - `contracts/utils/pausable_single_role.tact`
  - `contracts/utils/time.tact`
  - `contracts/jetton/jetton_factory.tact`
- We understood and reviewed the full flow of messages and funds from the user TON or Jetton wallet to the funding wallet, funding master contract, settlement contract, and Stream Pool contract. This included the data validation in each step, the AFR message flows, and the potential state reversions.
- We covered the integrations with the TEP-74 Jetton standard. This included reviewing the external message handlers for Jetton messages, data validation steps to keep track of balances, and AFR handlers for state reversion.
- We reviewed the implementation of the TEP-74 standard for the LP tokens in the Jetton Factory trait contract.
- We reviewed the message handlers for all contracts. These handlers are public entrypoints that any user can interact with. We reviewed access controls, gas and value calculations, and data validation for incoming messages.
- We reviewed the internal functionality of the funding wallet, funding master, settlement, and Stream Pool contracts. This includes the internal calculation functions for adding and removing liquidity and performing swaps in the Stream

Pool contract, depositing and withdrawing assets in the settlement contract, and the interactions between users and the funding contracts.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- Given the complexity of the system and the number of issues found during the audit, we believe that additional time for the engagement would have improved our coverage of the items listed below. It also could have increased our understanding and testing of the codebase, potentially leading to discovery of more issues.

- We did not cover the `oracle.tact` and `oracle_data.tact` contracts since they were not used anywhere in the PixelSwap DEX. Instead, we focused on currently implemented functionality to maximize our understanding of the system.

- We did not cover the following libraries and utilities:
  - `fixed_point.tact`
  - `math.tact`
  - `weighted_math.tact`
  - FunC files, such as `mathlib.fc`, `extra.fc`

- All interactions, tests, and flows related to the Stream Pool contract were made with a single actor, and no simultaneous transactions from multiple senders or actors were tested. Additional testing should be performed to ensure that concurrent interactions do not put the system in an inconsistent state.

- We did review the gas calculations, but given the complexity of the interactions and the number of code paths that could use gas to receive, process, or send messages in an asynchronous manner, we cannot assure that the calculation results are correct for every potential case. Even though we found issues related to gas calculations (such as TOB-PXL-7, TOB-PXL-20, and TOB-PXL-23), we recommend improving the test suite to cover all possible execution paths in the contracts and measure the gas usage for each particular case.

- We did not test the system for malformed input messages. We considered that attacker messages could have incorrect or deliberately malicious data, but not the cases where the data is corrupt or cannot be parsed by the receiving contract. These cases should be tested in the unit tests focusing on the data parsing functions.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase uses arithmetic operations to calculate the amount out for swaps, the amount of LP tokens to mint, the amount of tokens to transfer on removing liquidity, and new reserves of the pool on every user action. These operations follow the correct rounding direction to benefit the protocol. However, the system parameters are not bounded, and their impact has not been documented. We also found an issue related to using the wrong formula (TOB-PXL-10). The test suite does not cover several arithmetic edge cases.	**Weak**
Auditing	Most of the state-changing functions in the contracts emit events for off-chain monitoring. However, there is no specific documentation describing the events, their types, and the requirements for emission. We were not informed about any incident monitoring system or incident response plan. Regarding events, there is a name duplication issue between the `TokenBalanceChangeEvent` in the funding and the settlement contract. Some events have specific message types, and other events just reuse existing incoming messages and forward them. Given the issues described, no naming convention is in place. Refer to appendix C for more information about code quality issues and appendix D for recommendations on designing and implementing an incident response plan.	**Moderate**
Authentication /	The system implements only a few access control roles	**Moderate**

Access Controls	for privileged functions following the principle of least privilege. It is mentioned that certain privileged roles are meant to be bound to a hardware wallet or multisignature wallet, but it is not possible for us to verify these claims. Crucial privileged operations implement a two-step process; however, the existing test suite does not cover the access controls of all the privileged functions or the functionality for granting and revoking roles. The system can benefit from additional documentation specifying which actors are granted roles, their privileges, and the process of granting and revoking roles in the system.	
Complexity Management	The codebase includes unnecessary complexity that hinders automated and manual review. The functions overuse nested operations such as conditionals, exceptions, or recursive calls with complex return values. A single `PlaceOrder` message is used to implement all the core functionality of the system, which makes it difficult to follow the code and asset flow and introduces several security issues related to invalid user inputs. Using different parameters for gas checks and remaining gas computation also introduces unnecessary complexity. The codebase can benefit from the use of different messages to transfer TON and Jetton, swap and add/remove liquidity, and break messages and functions to be more granular in general. There are several instances of code duplication or code blocks that could be extracted into functions. This makes the codebase more difficult to read and can introduce bugs in certain situations, for example, when only one instance of the duplicated code is modified. Refer to appendix C for more information.	**Weak**
Decentralization	The project implements a decentralized exchange (DEX) in the TON network. However, it is not fully decentralized. All privileged operations are managed by the PixelSwap Labs team, either by hardware wallet accounts or	**Weak**

	multisignature wallets, as stated in the documentation. Additionally, these changes do not implement a timelock feature to allow users to opt out of these changes. There is no mention of a decentralized governance system to be implemented. System configuration parameters (such as adding funding or execution contracts or changing gas configuration) can only be changed by the owners.	
Documentation	The documentation is a multi-level specification of the important components. The codebase includes inline and NatSpec comments, and there is consistency across all documentation. Some inline comments are wrong and outdated. Additional documentation on the selection of gas parameters and gas value arithmetic is required to facilitate the computation of the correct gas parameters to ensure the execution of full message flow and maintain a stable system state.	**Moderate**
Low-Level Manipulation	From a Tact language point of view, the usage of raw FunC or assembly code can be considered low-level. In the PixelSwap codebase, multiple low-level libraries are used for basic TON functionality, such as `stdlib.fc`, math calculations, or a standard-compliant Jetton wallet library in FunC. As stated in this report's Project Coverage section, the low-level files were considered out of scope, and we recommend further analysis and testing for these libraries.	**Further Investigation Required**
Testing and Verification	The existing test suite includes unit tests for most of the functionality, but the test cases focus on the "happy path" and do not cover edge cases, adversarial user input, or failure cases. The internal transfer test cases fail. Additionally, several test cases do not check the resulting state sufficiently to ensure the correctness of the smart contract storage updates. There are no tests verifying the correct emission of events (external messages).	**Weak**

	The target version of the codebase is still under development, as evident from all the commented-out code in the contracts and functions, the debug "dump" statements, and the undocumented code. We recommend cleaning up the codebase, expanding the test suite's coverage and edge cases, and implementing additional test strategies such as fuzzing or end-to-end integration testing before the codebase is deployed and published. The test suite should be run in the CI/CD pipeline to detect bugs as early as possible.	
Transaction Ordering	We did not perform any tests on the codebase regarding transaction ordering risks. Since the TON network is asynchronous by design, the order in which messages are processed is not known from outside the chain, and it can be challenging to configure a specific order of processing for network messages.	**Further Investigation Required**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	The BulkInternalTransfer receiver computes the wrong total amounts	Data Validation	High
2	Granting the role to the master funding contract on a funding wallet contract will result in draining the funding wallet balance	Authentication	High
3	Users can inflate their funding wallet balance by sending an expired Swap message to the settlement contract	Data Validation	High
4	SettlementVault balances are not updated in the PlaceOrder_Partial_2 receiver of the settlement contract	Data Validation	Informational
5	Lack of validation of PixelswapStreamPool configuration parameters	Data Validation	Informational
6	Users can drain the TON balance of the PixelswapSettlement contract	Data Validation	High
7	Users can avoid paying the gas fee for the token creation transaction	Data Validation	Low
8	The tokens_count initial value in the PixelswapStreamPool contract is zero	Data Validation	Informational
9	The returned TON amount from an order execution result is ignored	Data Validation	Informational
10	Wrong formula used for LP amount calculation	Data Validation	High
11	Users can use nested PlaceOrders to drain the settlement contract	Data Validation	High

12	The LP tokens are never burned by the Stream Pool contract	Data Validation	**High**
13	Lack of the pair_id and token_id validation in the PixelswapStreamPool contract	Data Validation	**High**
14	The value attached to messages is not checked to be positive	Data Validation	**Informational**
15	The current balance is not checked before sending a message with a non-zero value	Data Validation	**High**
16	The exec_id value is not validated for the internal orders in nested PlaceOrder messages	Data Validation	**Medium**
17	The token_balance get function reverts if a user balance is 0	Data Validation	**Undetermined**
18	Different parsing formats for Jetton notification messages	Data Validation	**Medium**
19	The JettonFactory contract allows minting zero tokens	Data Validation	**Informational**
20	Incorrect gas calculations in several contracts	Data Validation	**Undetermined**
21	A privileged account can drain the PixelswapStreamPool contract	Access Controls	**Low**
22	The fee recipient accounts cannot be changed in the Stream Pool contract	Access Controls	**Informational**
23	The gas checks in the PixelswapStreamPool contract are wrongly placed	Data Validation	**Informational**
24	Users cannot deposit only Jetton to their funding wallet	Data Validation	**Informational**

Detailed Findings

1. The BulkInternalTransfer receiver computes the wrong total amounts	
Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-1
Target: `contracts/pixelswap_funding.tact`	

Description

The `BulkInternalTransfer` receiver function of the `PixelswapFundingWallet` contract computes the wrong total amounts to spend by adding the amount of the first entry multiple times.

In the `PixelswapFundingWallet` contract, when a `BulkInternalTransfer` message is processed, a loop calculates the total amount of tokens to be spent from the wallet. However, the loop does not increase the index variable `i`, so the result values will be the amounts for the first entry times the total number of entries.

Consequently, when the `spend_wallet` internal function is called, an incorrect balance will be deducted, or a panic will be raised if the balance is insufficient.

```
let i: Int = 0;
repeat (msg.num_entries) {
    let tx: SingleTransfer = msg.entries.get(i)!!;
    total_token_amt += tx.token_amt;
    total_ton_amt += tx.ton_amt;
}
```

Figure 1.1: The `i` variable is never incremented inside the loop (`pixelswap/contracts/pixelswap_funding.tact#L474–L479`)

Additionally, the loop in the receiver for `BulkInternalTransfer` in the `PixelswapFunding` contract is correctly implemented, which can result in the sum of user balances not matching the balance of the `PixelswapSettlement` contract, allowing users to withdraw more than they own and incorrectly decreasing the balance of the sender.

Exploit Scenario

Bob sends a `BulkInternalTransfer` message with three entries, the first of which has 0 `token_amt` and 0 `ton_amt`. Bob's funding wallet spends 0 of his tokens, but the master

funding contract correctly adds funds to the receivers' wallets. Bob is able to withdraw funds that do not belong to him from the funding contract.

Recommendations

Short term, increment the index variable `i` in the loop.

In the long term, expand the test suite to implement test cases for bulk transfers and ensure that these tests also consider malicious inputs.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

2. Granting the role to the master funding contract on a funding wallet contract will result in draining the funding wallet balance

Severity: **High**	Difficulty: **High**
Type: Authentication	Finding ID: TOB-PXL-2
Target: `contracts/pixelswap_funding.tact`	

Description

Granting the privileged role to the `PixelswapFunding` contract on a user-owned `PixelswapFundingWallet` contract will result in the user balance being spent even if the user is the recipient of the `InternalTransfer` message. This can drain the user's balance by repeatedly forwarding the `InternalTransfer` message between the funding wallet and the master contract.

Users have funding wallets to interact with the system. In these wallets, they are both owners and privileged users (they have a privileged role). Any user can grant the privileged role to an address on their funding wallets via the `GrantRole` message of the access control trait, as shown in figure 2.1:

```
/// Grant a role to a user.
receive(msg: GrantRole) {
    require(sender() == self.owner, "AccessControl: sender must be the owner");
    require(!self.has_role(msg.user), "AccessControl: user already has the role");
    self.grant_role(msg.user);
}
```

Figure 2.1: The owner can grant privileged access to any address (pixelswap/contracts/utils/access_control_single_role_non_transferrable.tact#L51–L56)

If a user grants the privileged role to the master funding contract, then the `InternalTransfer` message receiver function will always execute the first `if` block for the messages sent from the master funding contract, as shown in figure 2.2:

```
receive(msg: InternalTransfer) {
    require(self.has_role(sender()) || sender() == self.master, "Unauthenticated
sender"); // afr::allow-trap-at-input-boundary
    if (self.has_role(sender())) {
        require(context().value >= self.gas_config.gas_check_internal_transfer *
MILLITON, "Insufficient gas"); // afr::allow-trap-at-input-boundary
        require(msg.from_fund_id == self.fund_id, "Invalid fund ID"); //
afr::allow-trap-at-input-boundary
```

```
        require(msg.user == self.owner, "Invalid user"); //
afr::allow-trap-at-input-boundary
        require(msg.from_subaccount == self.subaccount, "Invalid subaccount"); //
afr::allow-trap-at-input-boundary
        // spend Jetton/TON balance
        self.spend_wallet(msg.token_id, msg.token_amt);
        // forward InternalTransfer message
        msg.toCell().send_to(self.master);
    } else if (sender() == self.master) {
        require(msg.to_fund_id == self.fund_id, "Invalid fund ID"); //
afr::allow-trap-impossible-path
        // fund Jetton/TON balance
        self.fund_wallet(msg.token_id, msg.token_amt, 0);
        refund_gas_to_user_value(msg.user, self.msg_value());
    }
}
```

Figure 2.2: The `else-if` block is never executed if the Funding contract is privileged (`pixelswap/contracts/pixelswap_funding.tact#L446–L463`)

This will spend the user's balance instead of adding funds to the user's wallet and send the same message to the master funding contract, which will then send it back to the user's funding wallet contract, and it will keep spending the user's balance repeatedly until the gas sent with the message is consumed.

Exploit Scenario

Alice accidentally grants the privileged role to the master funding contract on her funding wallet. From then on, any internal transfer she receives will not be accrued to her wallet and will drain her wallet balance.

Recommendations

Short term, review the order in which the checks are performed to ensure that even if the funding contract is privileged, it does not affect the expected outcome of the process. Also, consider limiting accounts that can get privileged roles in each system contract.

Long term, identify and document all possible actions that privileged accounts can take, along with their associated risks. This will facilitate codebase reviews and prevent future mistakes.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

3. Users can inflate their funding wallet balance by sending an expired Swap message to the settlement contract

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-3
Target: `contracts/pixelswap_streampool.tact`	

Description

Users can inflate their funding wallet balance of any token by sending a Swap message that will trigger a failure reply message from the `PixelswapStreamPool` contract. Users can use this to drain the settlement contract.

There are two ways to initiate a swap in the system: sending a `PlaceOrder` message to the user's funding wallet and sending a Swap message to the `PixelswapSettlement` contract.

The Swap message to the `PixelswapSettlement` contract has an `extra_tokens` field. The `token_gas` and `token_forward_milliton` fields of the `extra_tokens` structure are used to calculate the required gas amount and to decide if the output tokens should be added to the user's funding wallet or sent to the user's external wallet. All other fields of the `extra_tokens` values are ignored in the Swap message receiver function of the `PixelswapSettlement` contract. The Swap receiver of the settlement contract then creates a `PlaceOrder` message and sends it to the `PixelswapStreamPool` contract for execution.

If the `place_order` internal function of the `PixelswapStreamPool` contract detects a failure case, it sends the `PlaceOrder` message back to the `PixelswapSettlement` contract as a failure reply message. The settlement contract then forwards it to the `PixelswapFunding` contract, which forwards it to the user's `PixelswapFundingWallet` contract.

The `PixelswapFundingWallet` then processes the `PlaceOrder` message as unspent order, as shown in figure 3.1:

```
/// Unspend inputs of a [`PlaceOrder`](./pixelswap_messages.md#message-placeorder)
message.
fun unspend_order_inputs(msg: PlaceOrder) {
    require(msg.user == self.owner, "Invalid user"); //
afr::allow-trap-at-input-boundary
    require(msg.subaccount == self.subaccount, "Invalid subaccount"); //
afr::allow-trap-at-input-boundary
```

```
    require(msg.fund_id == self.fund_id, "Invalid fund ID"); //
afr::allow-trap-at-input-boundary
    if (msg.token_is_input) {
        let new_token_amt: Int = self.token_balances.get(msg.token_id)!! +
msg.token_amt; // afr::allow-trap-impossible-path
        self.token_balances.set(msg.token_id, new_token_amt);
        emit(TokenBalanceChangeEvent { token_id: msg.token_id, diff: msg.token_amt
}.toCell());
    }
    if (msg.extra_tokens != null) {
        let extra_tokens: ExtraTokens = load_extra_tokens(msg.extra_tokens!!); //
afr::allow-trap-impossible-path
        let ton_spent: Int = 0;
        let rem: Slice = extra_tokens.rem; // afr::allow-trap-impossible-path
        repeat (extra_tokens.num_tokens) {
            let token_info: TokenInfo = rem.load_token_info(); //
afr::allow-trap-impossible-path
            if (token_info.token_is_input) {
                if (token_info.token_id == ton_address()) {
                    ton_spent += token_info.token_amt;
                } else {
                    let new_token_amt: Int =
self.token_balances.get(token_info.token_id)!! + token_info.token_amt; //
afr::allow-trap-impossible-path
                    self.token_balances.set(token_info.token_id, new_token_amt);
                    emit(TokenBalanceChangeEvent { token_id: token_info.token_id,
diff: token_info.token_amt }.toCell());
                }
            }
        }
        if (ton_spent > 0) {
            let new_ton_bal: Int = self.token_balances.get(ton_address())!! +
ton_spent; // afr::allow-trap-impossible-path
            self.token_balances.set(ton_address(), new_ton_bal);
            emit(TokenBalanceChangeEvent { token_id: ton_address(), diff: ton_spent
}.toCell());
        }
    }
    if (msg.ref_po != null) {
        // recursively unspend inputs
        let ref_po: Cell = msg.ref_po!!; // afr::allow-trap-impossible-path
        self.unspend_order_inputs(ref_po.asSlice().load_place_order());
    }
}
```

Figure 3.1: The `token_is_input` check in `unspend_order_inputs` (pixelswap/contracts/pixelswap_funding.tact#L689–L726)

However, the `unspent_order_inputs` function of the `PixelswapFundingWallet` contract assumes that the tokens included in the `extra_tokens` field were spent from the user's wallet if the `token_is_input` is set to `true` and adds these tokens to the user's

balance, while the input tokens in the `extra_tokens` are not spent from the user's funding wallet if they initiate the swap by sending a `Swap` message to the settlement contract.

Exploit Scenario

Eve sends an expired `Swap` message with an `extra_tokens` field containing a `TokenInfo` for 1,000 USDT tokens with the `token_is_input` variable set to `true`. This message triggers a failure reply message from the `PixelswapStreamPool` contract, which is forwarded to Eve's funding wallet contract. The funding wallet contract adds 1,000 USDT to her balance, assuming they were spent from her wallet to initiate the swap. Eve withdraws these USDT tokens from her funding wallet and repeats this process to drain all the tokens from the settlement contract.

Recommendations

Short term, in the `Swap` message receiver of the settlement contract, check that the `extra_tokens` field of the message does not include any input tokens.

Long term, document the message flow in both success and failure cases and write test cases to ensure correct behavior in each case.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

4. SettlementVault balances are not updated in the PlaceOrder_Partial_2 receiver of the settlement contract

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-4
Target: `contracts/pixelswap_settlement.tact`	

Description

The `PlaceOrder_Partial_2` message receiver function of the `PixelswapSettlement` contract is used to relay messages between the funding contract and execution contracts, but it does not update the `SettlementVault` balances to reflect the transfer of funds between funding and execution contracts.

When a user initiates an order from their funding wallet, the `PlaceOrder` message is sent from their funding wallet to the master funding contract, which sends the same message to the `PixelswapSettlement` contract. The settlement contract then fetches the execution contract address from its storage and sends the `PlaceOrder` message to the `PixelswapSteampool` contract. However, the `PlaceOrder_Partial_2` message receiver of the `PixelswapSettlement` contract does not remove the input tokens from the `SettlementVault` of the funding contract and does not add the same tokens to the `SettlementVault` of the execution contract.

Similarly, when a failure message is sent from the execution contract to the settlement contract, the `PlaceOrder_Partial_2` message receiver of the `PixelswapSettlement` contract does not remove the input tokens from `SettlementVault` of the execution contract and does not add the same tokens to the `SettlementVault` of the funding contract before sending the message to the funding contract.

This results in an imbalance between the balances stored in the funding contracts, settlement vaults, and reserves stored in the execution contracts.

```
receive(msg: PlaceOrder_Partial_2) {
    require(context().value >= self.gas_config.gas_check_place_order_partial *
MILLITON, "Insufficient gas"); // afr::allow-trap-at-input-boundary
    let exec_addr_opt: Address? = self.exec_contracts.get(msg.exec_id);
    let fund_addr_opt: Address? = self.fund_contracts.get(msg.fund_id);
    let msg_cell: Cell = msg.toCell();
    if (msg_cell.afr_require(exec_addr_opt != null && fund_addr_opt != null,
self.gas_config.storage_gas_reserve)) { return; } // "Invalid exec or fund contract
ID"
    let exec_addr: Address = exec_addr_opt!!;
```

```
    let fund_addr: Address = fund_addr_opt!!;
    if (sender() == exec_addr) { // The message is AFR
        msg_cell.send_to(fund_addr); // Send AFR back to Funding
    } else { // Normal message from Funding
        require(sender() == fund_addr, "Unauthorized sender"); //
afr::allow-trap-irrelevant-path
        if (msg_cell.afr_require(!self.is_paused(),
self.gas_config.storage_gas_reserve)) { return; } // "Contract paused"
        msg_cell.send_to(exec_addr);
    }
}
```

Figure 4.1: Code handling `PlaceOrder` message not updating balances correctly for AFR messages (`pixelswap/contracts/pixelswap_settlement.tact#L571–L586`)

Recommendations

Short term, add calls to the `token_balance_add` and `token_balance_reduce` internal functions to correctly update the settlement vault balances of the funding and execution contracts while processing the `PlaceOrder_Partial_2` message.

Long term, create message flow and fund flow diagrams, and document all of the places where the balances of different components and users are tracked in the system.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

5. Lack of validation of PixelswapStreamPool configuration parameters

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PXL-5
Target: `contracts/pixelswap_streampool.tact`	

Description

The owner of the `PixelSwapStreamPool` contracts can set the configuration parameters for the pool and the gas to any value without limits. This can lead to broken functionality or high TON spending for users in the pools until the values are fixed.

According to the documentation, the owner role belongs to a multisignature wallet, so it is expected that prior to executing a configuration change, more than one actor will review the transaction. Thus, using a multisignature wallet makes it difficult to set wrong configuration values.

```
receive(msg: SetPoolConfig) {
    self.requireOwner(); // afr::allow-trap-at-input-boundary
    require(self.config.toCell().hash() != msg.config.toCell().hash(), "No
changes"); // afr::allow-trap-at-input-boundary
    require(msg.config.default_protocol_fee_bps >= 0 &&
msg.config.default_protocol_fee_bps <= 10000, "Invalid protocol fee"); //
afr::allow-trap-at-input-boundary
    self.config = msg.config;
    emit(PoolConfigSetEvent { config: msg.config }.toCell());
    refund_gas();
}

receive(msg: SetPoolGasConfig) {
    self.requireOwner(); // afr::allow-trap-at-input-boundary
    require(self.gas.toCell().hash() != msg.gas.toCell().hash(), "No changes"); //
afr::allow-trap-at-input-boundary
    self.gas = msg.gas;
    refund_gas();
}
```

Figure 5.1: The configuration is changed without validation (pixelswap/contracts/pixelswap_streampool.tact#L393–L407)

Exploit Scenario

Alice and Bob are signers for the 2-out-of-3 multisignature wallet that controls the deployed `PixelswapStreamPool` contract. Alice creates and signs a configuration change transaction, accidentally setting the `remove_liquidity` gas configuration to an extremely

high value. Bob trusts Alice, so he signs the transaction without reviewing it. When it is executed, users' interactions are more expensive than expected.

Recommendations

Short term, set some reasonable limits to the configuration parameters that can be set in the `PixelswapStreamPool` contract.

Long term, document the expected values for each parameter and their safe ranges. Add test cases for the configuration change functions.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

6. Users can drain the TON balance of the PixelswapSettlement contract

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-6
Target: `contracts/pixelswap_settlement.tact`	

Description

The gas parameter fields `forward_milliton` and `gas_transfer` of the `WithdrawFunds` message can be used to drain the TON balance of the `PixelswapSettlement` contract, as the values specified in these fields are not spent from the user's funding wallet.

The `WithdrawFunds` message in the Settlement contract allows users to withdraw the balance of their funding wallet to their external wallet. The `WithdrawFunds` receiver function of the settlement contract calls the `send_tokens` internal function to transfer the specified token and amount to the user:

```
receive(msg: WithdrawFunds) {
    require(sender() == self.fund_contracts.get(msg.fund_id)!!, "Unauthorized fund
contract"); // afr::allow-trap-irrelevant-path
    // TODO: validate in vault
    self.send_tokens(msg.token_id, msg.user, msg.token_amt, msg.forward_milliton,
msg.gas_transfer);
    if (msg.token_id == ton_address()) {
        let ton_amt: Int = msg.token_amt + (msg.forward_milliton + msg.gas_transfer)
* MILLITON;
        self.token_balance_reduce(msg.token_id, -ton_amt, true, msg.fund_id,
msg.subaccount);
    } else {
        self.token_balance_reduce(msg.token_id, -msg.token_amt, true, msg.fund_id,
msg.subaccount);
        let ton_amt: Int = (msg.forward_milliton + msg.gas_transfer) * MILLITON;
        self.token_balance_reduce(ton_address(), -ton_amt, true, msg.fund_id,
msg.subaccount);
    }
}
```

Figure 6.1: The `WithdrawFunds` message receiver in the settlement contract (`pixelswap/contracts/pixelswap_settlement.tact#L635–L647`)

The `send_tokens` function transfers the sum of the `token_amt`, `forward_milliton`, and `gas_transfer` values when the provided `token_id` is the TON native currency:

```
fun send_tokens(token_id: Address, destination: Address, token_amt: Int,
```

```
forward_milliton: Int, gas_transfer: Int) {
    if (token_id == ton_address()) {
        token_amt = token_amt + (forward_milliton + gas_transfer) * MILLITON;
        if (token_amt > 0) {
            destination.send_ton(token_amt, "Pixelswap".asComment());
        }
    } else {
        if (token_amt > 0 || forward_milliton > 0) {
            token_id.transfer_with_text(
                destination,
                token_amt,
                forward_milliton * MILLITON,
                gas_transfer * MILLITON,
                0,
                "Pixelswap"
            );
        }
    }
}
```

Figure 6.2: The send_tokens function (pixelswap/contracts/pixelswap_settlement.tact#L728–L746)

However, the `WithdrawFunds` message receiver of the `PixelSwapFundingWallet` contract does not spend the user's TON balance by the sum of `forward_milliton` and `gas_transfer` values for TON transfer:

```
receive(msg: WithdrawFunds) {
    require(context().value >= self.gas_config.gas_check_withdraw_funds * MILLITON,
"Insufficient gas"); // afr::allow-trap-at-input-boundary
    self.require_role(sender()); // afr::allow-trap-irrelevant-path
    require(msg.user == self.owner, "Invalid user"); //
afr::allow-trap-at-input-boundary
    require(msg.subaccount == self.subaccount, "Invalid subaccount"); //
afr::allow-trap-at-input-boundary
    require(msg.fund_id == self.fund_id, "Invalid fund ID"); //
afr::allow-trap-at-input-boundary
    // spend Jetton/TON balance
    self.spend_wallet(msg.token_id, msg.token_amt);
    // spend forward TON balance in case of Jetton transfer
    let ton_spent: Int = (msg.gas_transfer + msg.forward_milliton) * MILLITON;
    if (ton_spent > 0 && !msg.token_id.is_ton()) {
        self.spend_wallet(ton_address(), ton_spent);
    }
    // forward WithdrawFunds message
    msg.toCell().send_to_value(self.master, self.msg_value());
}
```

Figure 6.3: The `WithdrawFunds` receiver in the funding wallet contract (pixelswap/contracts/pixelswap_funding.tact#L427–L442)

A user can specify non-zero values for the `forward_milliton` and `gas_transfer` fields in the `WithdrawFunds` message to drain the TON balance of the settlement contract.

Exploit Scenario
Bob sends 1 nanoton to his funding wallet and creates a message to withdraw all TON balance from the settlement contract.

Recommendations
Short term, consider one of the following:

- Ensure that the sum of the `token_amt`, `forward_milliton`, and `gas_transfer` values is deducted from the user's funding wallet while processing the `WithdrawFunds` message.
- Or ensure that the `forward_milliton` and `gas_transfer` values are zero when the `msg.token_id` is the TON address.
- Or update the `send_tokens` to send only `token_amt` TON when transferring the native TON.

Long term, implement tests that verify the total amount of TON balance of the system smart contracts is kept constant after each user action.

Fix Review Status
After conducting a fix review, the team determined that this issue has been resolved.

7. Users can avoid paying the gas fee for the token creation transaction

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-7
Target: `contracts/pixelswap_streampool.tact`	

Description

To create a new token, a user must send a `CreateToken` message to the Stream Pool contract with enough value to pay for the token creation fee. This fee is also used as a mitigation feature to disincentivize the creation of excessively many tokens, given that the tokens are stored in a mapping structure inside the Stream Pool contract's storage.

In the `CreateToken` message handler, the message value is checked to ensure that it is enough to pay the creation fee, but it doesn't take into account the gas costs needed to execute the transaction.

This allows any user to create new tokens without paying for the computation gas required to create them, using the Stream Pool contract balance to pay for it.

```
receive(msg: CreateToken) {
    require(self.has_role(sender()) || context().value >=
self.config.token_creation_fee * MILLITON, "Incorrect token creation fee"); //
afr::allow-trap-at-input-boundary
    require(self.has_role(sender()) || self.config.enable_token_creation, "Token
creation not possible"); // afr::allow-trap-at-input-boundary
    require(self.tokens_config.get(msg.token_id) == null, "Token already exists");
// afr::allow-trap-at-input-boundary
    self.tokens_count += 1;
    self.tokens_config.set(msg.token_id, TokenConfig {
        // is_validated: false,
        is_active: true, // by default, tokens are active and can be used to create
pairs without validation
        // token_wallet_address: newAddress(0, 0x04), // Dummy address. Address 0
implies TON so we avoid that.
        // gas_forward_milliton: self.gas.default_token_forward_milliton,
        // gas_transfer: self.gas.default_token_transfer,
        // query_id: 0,
    });
    emit(TokenCreatedEvent { token_id: msg.token_id }.toCell());
    self.creation_fee_recipient.send_ton(self.config.token_creation_fee * MILLITON,
null);
}
```

Figure 7.1: The message value validation missing the check for gas costs (`pixelswap/contracts/pixelswap_streampool.tact#L331–L346`)

Exploit Scenario

Alice decides to drain the balance of a `PixelswapStreamPool` contract. To do this, she starts sending `CreateToken` messages to the Stream Pool with a value just above (i.e., by 1 nanoton) the token creation fee.

Repeating this will drain the `PixelswapStreamPool` contract balance, which will be used to pay for all the `CreateToken` message processing. Admins will need to send TON to it to continue paying the storage fee.

Recommendations

Short term, modify the value check in the `CreateToken` message receiver to ensure the user provides enough TON for the required gas.

Long term, expand checks in the test cases to ensure that the TON balance of the system smart contracts does not go down after user actions that do not transfer TON.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

8. The tokens_count initial value in the PixelswapStreamPool contract is zero

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-PXL-8
Target: `contracts/pixelswap_streampool.tact`	

Description

The `token_count` variable in the `PixelswapStreamPool` contract is meant to track how many tokens are added. All stream pool contracts add the TON as a token in the `init` function, but the `tokens_count` variable is not updated in the `init` function, and it remains 0.

This has no impact on the current state of the codebase, as the tokens are stored in a non-iterable mapping.

Recommendations

Short term, make sure the `tokens_count` variable is updated in the `init` function to track the correct number of tokens added to the `PixelswapStreamPool` contract.

Long term, expand the test suite to implement test cases to check the variable values in all of the contracts. This will ensure correct value updates.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

9. The returned TON amount from an order execution result is ignored

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PXL-9
Target: `contracts/pixelswap_settlement.tact`	

Description

When an order is processed in the `PixelswapStreamPool` contract, an `OrderExecutionResult` message is returned to the settlement contract. This message contains information about the amounts of tokens and TON that should be deducted from the settlement vault balance of the `exec_id` and added to the settlement vault balance of the `fund_id`.

The `process_order_execution_result` internal function of the settlement contract reduces the `token_amt` of the `token_id` token from the settlement vault balance of the `exec_id` and adds the same balance to the settlement vault of the `fund_id` if the tokens are not transferred to the user's external wallet. However, this function ignores the value of the `ton_amt` field of the `OrderExecutionResult` message and does not update the settlement vault balance for the native TON token:

```
fun process_order_execution_result(msg: OrderExecutionResult, res: Cell?, gas_spent:
Int): ProcessOrderExecutionResult {
    let include_current: Bool = false;
    // Distribute results
    if ((msg.token_id == null || msg.token_amt == 0) && (msg.ton_amt > 0)) { // if
no jetton to send, but some ton to send
        self.token_balance_reduce(ton_address(), -msg.ton_amt, false, msg.exec_id,
msg.pair_id); // TODO: #398
        if (0.or_unwrap(msg.token_gas) > 0) { // send to external
            gas_spent += (0.or_unwrap(msg.token_forward_milliton) + msg.token_gas!!)
* MILLITON;
            self.send_tokens(ton_address(), msg.user, msg.ton_amt,
0.or_unwrap(msg.token_forward_milliton), msg.token_gas!!);
        } else { // send to funding: forward to Funding -> Funding Wallet
            include_current = true;
            self.token_balance_add(ton_address(), msg.ton_amt, true, msg.fund_id,
msg.subaccount);
        }
    } else if (msg.token_id != null) { // has jetton to send
        self.token_balance_reduce(msg.token_id!!, -msg.token_amt, false,
msg.exec_id, msg.pair_id);
        if (0.or_unwrap(msg.token_gas) > 0) { // send to external
            gas_spent += (0.or_unwrap(msg.token_forward_milliton) + msg.token_gas!!)
```

```
* MILLITON;
            self.send_tokens(msg.token_id!!, msg.user, msg.token_amt,
0.or_unwrap(msg.token_forward_milliton) + msg.ton_amt, msg.token_gas!!);
        } else { // send to funding: forward to Funding -> Funding Wallet
            include_current = true;
            self.token_balance_add(msg.token_id!!, msg.token_amt, true, msg.fund_id,
msg.subaccount);
        }
    } // otherwise do nothing because no tokens to send
    let oer: Cell? = OrderExecutionResult {
            exec_id: msg.exec_id,
            fund_id: msg.fund_id,
            user: msg.user,
            subaccount: msg.subaccount,
            pair_id: msg.pair_id,
            token_id: msg.token_id,
            token_amt: msg.token_amt,
            token_gas: msg.token_gas,
            token_forward_milliton: msg.token_forward_milliton,
            ton_amt: msg.ton_amt,
            ref: res,
        }.toCell();
    return ProcessOrderExecutionResult {
        gas_spent,
        res : include_current ? oer : res, // FIX: depends on
https://github.com/tact-lang/tact/issues/392
        inner: msg.ref,
    };
}
```

Figure 9.1: The branch that lacks validation checks for tokens (pixelswap/contracts/pixelswap_settlement.tact#L769–L809)

The existing execution contract `PixelswapStreamPool` does not send a non-zero value of the `ton_amt` in the `OrderExecutionResult` message; therefore, this issue does not lead to any loss of funds in the current system. New execution contracts may send a Jetton and TON together in a single `OrderExecutionResult` message, which will result in an out-of-sync balance of settlement vaults and other contracts.

Recommendations

Short term, if the `ton_amt` value is non-zero, update the settlement vault balances of the `exec_id` and `fund_id` for the TON token when processing the `OrderExecutionResult` message.

Long term, create message flow and fund flow diagrams and document all of the places where the balances of different components and users are tracked in the system.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

10. Wrong formula used for LP amount calculation

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-10
Target: `contracts/univ2_math.tact`	

Description

The `PixelswapStreamPool` contract uses the wrong formula to compute the LP token amount for adding liquidity. This results in liquidity providers losing on the fee earned from the swaps.

The `PixelswapStreamPool` contract calls the `calc_add_lp_balanced` function of the `univ2_math.tact` library contract to compute the LP token amount to mint for a liquidity provider. The `calc_add_lp_balanced` function computes the LP token amount by computing the square root of the new reserves and deducting the last total supply of the LP tokens from the result of the square root:

```
inline fun calc_add_lp_balanced(amount0: Int, amount1: Int, reserve0: Int, reserve1:
Int, lp_supply: Int, fee_bps: Int, weight0: Int): LiquidityResult {
    if (lp_supply == 0) {
        return LiquidityResult { amount0, amount1, liquidity: sqrt(amount0 *
amount1), fee0: 0, fee1: 0 };
    }
    if (amount0 <= 0 || amount1 <= 0 || reserve0 <= 0 || reserve1 <= 0 || lp_supply
< 0 || fee_bps < 0 || fee_bps >= 10000 || weight0 <= 0 || weight0 >= 10000) {
        return LiquidityResult { amount0: 0, amount1: 0, liquidity: 0, fee0: 0,
fee1: 0 };
    }
    let actual_amount0: Int = amount0;
    let actual_amount1: Int = _native_muldivc(amount0, reserve1, reserve0);
    if (actual_amount1 > amount1) {
        actual_amount0 = _native_muldivc(amount1, reserve0, reserve1);
        actual_amount1 = amount1;
        if (actual_amount0 > amount0) { // if still not possible (due to rounding),
abort
            return LiquidityResult { amount0: 0, amount1: 0, liquidity: 0, fee0: 0,
fee1: 0 };
        }
    }
    let new_lp: Int = sqrt((reserve0 + actual_amount0) * (reserve1 +
actual_amount1)) - lp_supply;
    if (new_lp <= 0) {
        return LiquidityResult { amount0: 0, amount1: 0, liquidity: 0, fee0: 0,
fee1: 0 };
```

```
    }
    return LiquidityResult { amount0: actual_amount0, amount1: actual_amount1,
liquidity: new_lp, fee0: 0, fee1: 0 };
}
```

Figure 10.1: The `calc_add_lp_balanced` function (`pixelswap/contracts/univ2_math.tact#L73–L94`)

However, the remove liquidity function computes the amount of tokens to transfer in proportion to the LP tokens to burn with respect to the total supply of the LP tokens. This results in liquidity providers losing their share of the swap fee because the swap fee accrued by the pool up to a certain time is given to the new liquidity providers instead of being distributed among the existing liquidity providers.

The `calc_add_lp_unbalanced` function of the `univ2_math.tact` library contract also uses the wrong formula for calculating the LP token amount.

Exploit Scenario

Let us consider a TON / USDC pool with 0 liquidity.

- Alice deposits 10,000,000,000 nanotons TON and 10,000,000,000 nanotons USDC in the pool and mints 10,000,000,000 nanotons LP tokens.
- Bob swaps 1,000,000,000 nanotons TON for 908,264,387 nanotons USDC. The reserves after the swap are 10,999,970,000 nanotons TON and 9,091,735,613 nanotons USDC.
- Eve deposits 10,999,970,000 nanotons TON and 9,091,735,613 nanotons USDC, and the protocol mints 10,000,881,879 nanotons LP tokens for Eve.
- Eve redeems her LP tokens for 11,000,455,010 nanotons TON and 9,092,136,485 nanotons USDC, which is more than the 10,999,484,989 nanotons TON and 9,091,334,740 nanotons USDC that Alice can redeem by removing all her liquidity.

Recommendations

Short term, use the following formula for the LP token amount calculation in the `calc_add_lp_balanced` and `calc_add_lp_unbalanced` functions when a user adds liquidity to a non-empty pool:

```
let new_lp: Int = min(_native_muldivc(actual_amount0, lp_supply, reserve0),
_native_muldivc(actual_amount1, lp_supply, reserve1));
```

Figure 10.2: The correct formula for LP token amount calculation

Long term, define high-level system invariants and implement end-to-end test cases to check these invariants with multiple transactions by different users. Consider using a fuzzer to test these invariants.

Fix Review Status
After conducting a fix review, the team determined that this issue has been resolved.

11. Users can use nested PlaceOrders to drain the settlement contract

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-11
Target: `contracts/pixelswap_funding.tact`	

Description

The `spend_order_inputs` function of the `PixelswapFundingWallet` contract does not spend the input tokens of the nested `PlaceOrder` messages, which allows users to use nested `PlaceOrder` messages to drain all the assets from the settlement contract.

The `PlaceOrder` receiver function of the `PixelswapFundingWallet` contract calls the `spend_order_inputs` internal function to update the user's balance of tokens for the order. The `PlaceOrder` message has an optional `ref_po` field, which can have a `ref` to another `PlaceOrder` message. The `spend_order_inputs` function processes these nested `PlaceOrder` messages by recursively calling the `spend_order_inputs` function:

```
if (msg.ref_po != null) {
    // recursively spend inputs
    let ref_po: Cell = msg.ref_po!!; // afr::allow-trap-impossible-path
    return self.spend_order_inputs(ref_po.asSlice().load_place_order(), gas_spent);
}
```

Figure 11.1: Snippet of the `spend_order_input` function processing nested orders (`pixelswap/contracts/pixelswap_funding.tact#L681–L685`)

However, the return statement of the `spend_order_inputs` function compiles to the following FunC code:

```
return ((($self'gas_config'storage_gas_reserve, $self'gas_config'gas_consumption,
$self'gas_config'gas_check_place_order, $self'gas_config'gas_check_withdraw_funds,
$self'gas_config'gas_check_internal_transfer,
$self'gas_config'gas_check_bulk_internal_transfer,
$self'gas_config'gas_for_place_order,
$self'gas_config'gas_for_place_order_cell_depth), $self'master, $self'fund_id,
$self'owner, $self'subaccount, $self'user_role, $self'token_balances),
(($self'gas_config'storage_gas_reserve, $self'gas_config'gas_consumption,
$self'gas_config'gas_check_place_order, $self'gas_config'gas_check_withdraw_funds,
$self'gas_config'gas_check_internal_transfer,
$self'gas_config'gas_check_bulk_internal_transfer,
$self'gas_config'gas_for_place_order,
$self'gas_config'gas_for_place_order_cell_depth), $self'master, $self'fund_id,
$self'owner, $self'subaccount, $self'user_role,
```

```
$self'token_balances)~$PixelswapFundingWallet$_fun_spend_order_inputs($Slice$_fun_lo
ad_place_order($Cell$_fun_asSlice($ref_po)), $gas_spent));
```

Figure 11.2: The FunC code compilation of the return statement of the `spend_order_inputs` function

The above FunC code returns the first FunC tensor, which is computed by processing the input tokens specified in the outermost `PlaceOrder` instead of returning the result of the recursive call of the `$PixelswapFundingWallet$_fun_spend_order_inputs` FunC function. The value returned from this function is then stored in the contract storage.

Exploit Scenario

Eve deposits 1000,000,000 nanotons TON and 1000,000,000 nanotons USDC in her funding wallet. Eve then sends a `PlaceOrder` message to add liquidity of 1000,000,000 nanotons TON and 1000,000,000 nanotons USDC to the TON/USDC pool, with the `ref_po` having another `PlaceOrder` order message to add the same liquidity again. The `spend_order_inputs` function of the funding wallet contract deducts Eve's balance for only the outermost `PlaceOrder` message, stores the new values of 0 TON and 0 USDC balance in the contract storage, and sends the whole `PlaceOrder` message to the funding master contract. The whole `PlaceOrder` is then successfully processed by the `PixelswapStreamPool` contract to mint LP token amounts for both `PlaceOrder` messages for Eve. Eve then redeems all her LP tokens from the pool and withdraws 2000,000,000 nanotons TON and 2000,000,000 nanotons USDC.

Recommendations

Short term, replace the return statement of the recursive call with the following:

```
if (msg.ref_po != null) {
    // recursively spend inputs
    let ref_po: Cell = msg.ref_po!!; // afr::allow-trap-impossible-path
    gas_spent = self.spend_order_inputs(ref_po.asSlice().load_place_order(), gas_spent);
    return gas_spent;
}
```

Figure 11.3: Snippet of the `spend_order_input` function processing nested orders (`pixelswap/contracts/pixelswap_funding.tact#L681–L685`)

Long term, expand the test suite to check execution of nested orders. Check the whole system state after a transaction in a test case to ensure correctness of the test cases.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

12. The LP tokens are never burned by the Stream Pool contract

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-12
Target: `contracts/jetton/jetton_factory.tact`	

Description

The `PixelswapStreamPool` contract sends the `burn` message to the Jetton master contract of the pair's LP token instead of the Jetton wallet owned by itself. This results in the LP tokens never being burned.

Users remove their liquidity by transferring their LP tokens to the `PixelswapStreamPool` contract with a payload to send the `RemoveLiquidityJettonNotification` message to the `PixelswapStreamPool` contract. The `RemoveLiquidityJettonNotification` handler function of the Stream Pool contract updates the reserves and LP token supply stored in the `pair_config` map and calls the `self.burn` function. The `burn` function is inherited from the `JettonFactory` trait contract:

```
fun burn(pair_id: Int, amount: Int) {
    send(SendParameters {
        to: self.jetton_address_of(pair_id),
        value: self.gas_send_burn,
        mode: SendIgnoreErrors,
        body: JETTONFACTORY__JettonBurn {
            query_id: 0,
            amount: amount,
            response_destination: myAddress(),
            custom_payload: null
        }.toCell(),
        bounce: false
    });
}
```

Figure 12.1: The `burn` function in the `JettonFactory` trait (`pixelswap/contracts/jetton/jetton_factory.tact#L88–L101`)

The `burn` function of the `JettonFactory` trait sends the `burn` message to the Jetton master contract of the LP token with the `bounce` flag set to `false`. The Jetton master contract reverts the transaction, and the `burn` message is never processed by the Jetton contracts of the LP token.

Exploit Scenario

Alice adds 1000,000,000 nanotons TON and 1000,000,000 nanotons USDC to the pool and mints 1000,000,000 nanotons LP tokens. After some time, Alice removes her liquidity by

transferring her LP tokens to the `PixelswapStreamPool` contract, but the LP tokens are not burned by the Stream Pool contract, and the LP token balance of the Stream Pool contract is increased by 1,000,000,000 nanotons LP tokens.

Recommendations

Short term, update the `burn` function of the `JettonFactory` trait contract to send the `burn` message to the Jetton wallet owned by the Stream Pool contract.

Long term, check the whole system state after a transaction in a test case to ensure correctness of the test cases.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

13. Lack of the pair_id and token_id validation in the PixelswapStreamPool contract

Severity: **High**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-13
Target: `contracts/pixelswap_streampool.tact`	

Description

The `PlaceOrder` message receiver function of the `PixelswapStreamPool` contract does not validate that the `pair_id` specified in the message corresponds to the `token_id` values specified in the message. This allows users to provide arbitrary `token_id` values to execute a swap or add liquidity to a pair without transferring the required tokens.

The protocol allows users to transfer their Jettons and execute a protocol action in a single transaction by specifying order parameters in the `forward_payload` field of the `transfer` message of the Jetton contract. The Jetton contract sends the `OrderJettonNotification` message to the settlement contract, which adds a deposit to the user's funding wallet, executes a swap, or adds liquidity to the specified pair based on the `forward_payload` value specified in the `OrderJettonNotification` message.

However, the `OrderJettonNotification` message receiver cannot verify that the message is sent by the correct Jetton wallet contract and is not sent by a malicious actor; it considers the sender as the `token_id` for the order to be executed. This allows users to add a funding wallet balance for an arbitrary `token_id`, execute a swap without transferring Jettons, and add liquidity to a pool without transferring Jettons by sending the `OrderJettonNotification` message from an arbitrary address.

```
receive(msg: OrderJettonNotification) {
    require(context().value >= self.gas_config.gas_check_jetton_notification *
MILLITON, "Insufficient gas"); // afr::allow-trap-at-input-boundary
    let ton_amt: Int = context().value -
self.gas_config.gas_for_incoming_jetton_transfer * MILLITON; // token0 is TON
    let token_id: Address = sender();
    let payload: Slice = msg.forward_payload;
    ...
```

Figure 13.1: A snippet of the `OrderJettonNotification` receiver function (pixelswap/contracts/pixelswap_settlement.tact#L327–L331)

As a result, the `PlaceOrder` message receiver function of the `PixelswapStreamPool` contract does not check if the user-provided value of the `pair_id` field corresponds to the

`token_id` values provided in the `PlaceOrder` message. This allows attackers to send the `OrderJettonNotification` message from an arbitrary address to steal tokens from the settlement contract.

Exploit Scenario 1

Eve sends an `OrderJettonNotification` message from her smart wallet to swap USDC for TON from the TON/USDC pool. The order is executed successfully, and Eve gets free TON from the settlement contract.

Exploit Scenario 2

Eve sends an `OrderJettonNotification` message from her smart wallet to deposit a fake Jetton to her funding wallet. She then sends a `PlaceOrder` message to swap USDC for TON by specifying the fake Jetton as input token `token_id` and TON/USDC pair `pair_id`. The order is executed successfully, and Eve gets free TON from the settlement contract.

Recommendations

Short term, validate that the `pair_id` specified in the `PlaceOrder` message corresponds to the provided `token_id` values to ensure correct token transfers before executing an order.

Long term, validate all user inputs at all of the system's entrypoints to ensure the correctness and security of the system.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

14. The value attached to messages is not checked to be positive

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PXL-14
Target: `contracts/utils/msgtools.tact`	

Description

The message tools library implements helper functions to deal with message passing between contracts. Some of these functions deal with messages that carry TON value, and the caller can specify the amount of TON to send.

In two of these functions, there is no check to ensure that the value to be set is positive. Given that the parameter is of type `Int`, it is possible to pass a negative value and break functionality.

The vulnerable functions are `send_to_value` and `send_and_deploy_value`, shown in figures 14.1 and 14.2:

```
extends inline fun send_to_value(self: Cell, to: Address, value: Int) {
    send(SendParameters {
        bounce: false,
        to: to,
        value: value,
        mode: SendIgnoreErrors,
        body: self
    });
}
```

Figure 14.1: The `send_to_value` function (pixelswap/contracts/utils/msgtools.tact#L95–L103)

```
extends inline fun send_and_deploy_value(self: Cell, toinit: StateInit, value: Int) {
    send(SendParameters {
        bounce: false,
        to: contractAddress(toinit),
        value: value,
        mode: SendIgnoreErrors,
        body: self,
        code: toinit.code,
        data: toinit.data
    });
}
```

Figure 14.2: The `send_and_deploy_value` function (pixelswap/contracts/utils/msgtools.tact#L131–L141)

The following list shows examples of potentially dangerous usages of this function in the current codebase:

- `OrderJettonNotification` handler in `pixelswap_settlement.tact`
- `RemoveLiquidityJettonNotification` handler in `pixelswap_streampool.tact`
- `Swap` handler in `pixelswap_settlement.tact`
- `OrderExecutionResult` handler in `pixelswap_settlement.tact`
- `InternalTransfer` handler in `pixelswap_settlement.tact`
- `Deposit` function in `pixelswap_settlement.tact`

Most of these usages depend on the current values for the gas configuration, lack of check or enforcement of `context().value` that could be bypassed by future implementations, or values that depend on the execution state (reliance on `gas_consumed()`).

Recommendations

Short term, add a validation for the value parameter to be positive in the `send_to_value` and `send_and_deploy_value` functions.

Long term, check all usages of these functions in the codebase and ensure that all calculations for the value parameters are well-defined, bounded, and never negative. Enforce these checks by creating new test cases.

Fix Review Status

After conducting a fix review, the team determined that this issue has not been resolved.

15. The current balance is not checked before sending a message with a non-zero value

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-PXL-15
Target: `contracts/utils/msgtools.tact`	

Description
All message sending functions in the message tools library use the `SendIgnoreErrors` mode. If the sender contract's current TON balance is insufficient to send the specified value, the message will not be sent and will be dropped silently in the action phase. Consequently, funds can be lost, or the system state can become out of sync between contracts.

This issue can be exploited in different ways, one of which is described in the Exploit Scenario section below.

Exploit Scenario
In the settlement contract, when the `token_id` is the TON native asset, the `send_tokens` function sends a token amount equal to the sum of the specified `token_amt`, the forwarding gas, and the gas cost to pay for the transfer. However, it never checks that the contract's current balance exceeds the value to be sent.

This results in a loss of funds for the user, because the funding wallet's TON balance has already been reduced at this point.

Recommendations
Short term, when dealing with TON amounts to be transferred in messages, first ensure that the current contract's balance is enough. In case of failure, notify the sender via AFR.

Long term, document all gas usages in the different message flows; ensure that the contract balances cannot drop below a certain threshold; and implement tests that check that the functionality of the system is not broken in low-balance situations.

Fix Review Status
After conducting a fix review, the team determined that this issue has not been resolved.

16. The exec_id value is not validated for the internal orders in nested PlaceOrder messages

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-16
Target: `contracts/pixelswap_streampool.tact`, `contracts/pixelswap_settlement.tact`	

Description

When a `PlaceOrder` message arrives at the `PixelswapStreamPool` contract, the `exec_id` is checked to ensure it matches the current contract's `exec_id`. This is also checked in the settlement contract to ensure that the message is routed to the correct `PixelswapStreamPool`:

```
receive(msg: PlaceOrder) { // FEAT: handle nested PlaceOrder for atomic swaps and
split paths
    require(sender() == self.settlement, "Unauthorized sender"); //
afr::allow-trap-irrelevant-path
    // If exec_id doesn't match, the order is routed to the wrong contract. This
should never happen.
    require(msg.exec_id == self.exec_id, "Invalid exec_id"); //
afr::allow-trap-impossible-path
    let p: ProcessOrderResult = self.process_order(msg, null, 0);
    while (p.ref_po != null) {
        p = self.process_order(p.ref_po!!.asSlice().load_place_order(), p.exec_res,
p.gas_spent);
    }
    if (p.exec_res != null) {
        if (msg.toCell().afr_require(context().value > p.gas_spent + storage_fees()
+ gas_consumed(), self.gas.storage_reserve)) { return; }
        let remaining_gas: Int = context().value - storage_fees() - gas_consumed();
        p.exec_res!!.send_to_value(self.settlement, remaining_gas);
    }
}
```

Figure 16.1: `PlaceOrder` message handler in the `PixelswapStreamPool` contract (`pixelswap/contracts/pixelswap_streampool.tact#L175–L188`)

```
receive(msg: PlaceOrder_Partial_2) {
    require(context().value >= self.gas_config.gas_check_place_order_partial *
MILLITON, "Insufficient gas"); // afr::allow-trap-at-input-boundary
    let exec_addr_opt: Address? = self.exec_contracts.get(msg.exec_id);
    let fund_addr_opt: Address? = self.fund_contracts.get(msg.fund_id);
    let msg_cell: Cell = msg.toCell();
```

```
    if (msg_cell.afr_require(exec_addr_opt != null && fund_addr_opt != null,
self.gas_config.storage_gas_reserve)) { return; } // "Invalid exec or fund contract
ID"
    let exec_addr: Address = exec_addr_opt!!;
    let fund_addr: Address = fund_addr_opt!!;
    if (sender() == exec_addr) { // The message is AFR
        msg_cell.send_to(fund_addr); // Send AFR back to Funding
    } else { // Normal message from Funding
        require(sender() == fund_addr, "Unauthorized sender"); //
afr::allow-trap-irrelevant-path
        if (msg_cell.afr_require(!self.is_paused(),
self.gas_config.storage_gas_reserve)) { return; } // "Contract paused"
        msg_cell.send_to(exec_addr);
    }
}
```

Figure 16.2: `PlaceOrder` message handler in the settlement contract (`pixelswap/contracts/pixelswap_settlement.tact#L571–L586`)

However, `PlaceOrder` messages can have nested `PlaceOrder` messages in the `ref_po` field, and those nested `exec_id`s are never checked by the settlement or Stream Pool contracts.

Additionally, when the order is processed by `process_order()`, a `ProcessOrderResult` message is generated with the `exec_id` of the current Stream Pool, no matter what the original message's `exec_id` was. This result is sent back to the settlement contract to keep track of the accounting, and the `process_order_execution` function updates the settlement vault balances using the `exec_id` set by the Stream Pool.

```
   let exec_res: Cell? = self.make_execution_results(self.exec_id, msg.fund_id,
msg.user, msg.subaccount, pair_id,
        token0_id, amountx_to_send, token0_id_transfer_gas,
token0_id_token_forward_milliton,
        token1_id, amounty_to_send, token1_id_transfer_gas,
token1_id_token_forward_milliton,
        inner_res);

    // if (msg.ref_po != null) {
    //     let ref_po: Cell = msg.ref_po!!; // afr::allow-trap-impossible-path
    //     self.process_order(ref_po.asSlice().load_place_order(), gas_balance);
    // }
    gas_spent += (
            token0_id_transfer_gas
            + token0_id_token_forward_milliton
            + token1_id_transfer_gas
            + token1_id_token_forward_milliton
            + self.gas.default_place_order
        ) * MILLITON;
```

```
    if (msg.toCell().afr_require(context().value > gas_spent,
self.gas.storage_reserve)) { return ProcessOrderResult { ref_po: null, exec_res:
inner_res , gas_spent: gas_spent}; } // "Insufficient gas"

    return ProcessOrderResult { ref_po: msg.ref_po, exec_res: exec_res, gas_spent:
gas_spent };
}
```

Figure 16.3: The execution results will have the current contract's `exec_id` (`pixelswap/contracts/pixelswap_streampool.tact#L658–L678`)

Exploit Scenario

Bob, a PixelSwap user, creates an order composed of a TON/USDC swap and a deposit to a USDC/USDT pool located in a different Stream Pool contract. In order to save some time, he nests both orders in a single `PlaceOrder` message, believing that the system will correctly identify the execution contracts that must be called. The order executes successfully and the resulting reserves of the TON/USDC pair will be incorrect since both orders were processed by the TON/USDC pool execution contract.

Recommendations

Short term, check that the `exec_id` value matches the current execution contract's ID for all orders in a nested `PlaceOrder` message. Notify the user in case an invalid order is not processed so they can retry later or at least be aware of the execution failure.

Long term, consider re-engineering the architecture and the way nested orders are created, transmitted, and handled. Alternatively, consider rewriting the `PlaceOrder` nesting feature in order to make it less error-prone (e.g.,, by nesting only the non-redundant information).

Fix Review Status

After conducting a fix review, the team determined that this issue has not been resolved.

17. The token_balance get function reverts if a user balance is 0

Severity: **Undetermined**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-17
Target: `contracts/pixelswap_funding.tact`	

Description

The `token_balance` function in the `PixelswapFundingWallet` contract should return the current wallet's balance for the specified token. However, the balances mapping entry is deleted when the user balance becomes zero, so the `token_balance` function reverts when it is called for users with zero balance.

This can affect external off-chain integrations, potentially breaking external projects that are attempting to get information from Pixelswap.

```
get fun token_balance(token_id: Address): Int {
    return self.token_balances.get(token_id)!!;
}
```

Figure 17.1: The `token_balance` getter (`pixelswap/contracts/pixelswap_funding.tact#L738–L740`)

Recommendations

Short term, return the expected value of zero if the mapping entry does not exist.

Long term, consider rewriting the parts of the code where the mapping entry is deleted for zero balance results. Since test cases can also break when this call reverts, it is recommended that the test suite be improved to detect this kind of unexpected behavior.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

18. Different parsing formats for Jetton notification messages

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-PXL-18
Target: `contracts/pixelswap_settlement.tact`, `contracts/pixelswap_streampool_messages.tact`	

Description

The Jetton notification message for `transfer_notification`, with ID `0x7362d09c`, is defined differently for the settlement and the Stream Pool contracts. In particular, the `fund_id` and `subaccount` structure fields are read in reverse order, as shown in figures 18.1 and 18.2.

```
message(0x7362d09c) OrderJettonNotification {
    fund_id: Int as uint32;
    subaccount: Int as uint32;
    amount: Int as coins;
    sender: Address;
    forward_payload: Slice as remaining;
}
```

Figure 18.1: The definition of the message in the settlement contract (pixelswap/contracts/pixelswap_settlement.tact#L188–L194)

```
message(0x7362d09c) RemoveLiquidityJettonNotification {
    subaccount: Int as uint32;
    fund_id: Int as uint32;
    amount: Int as coins;
    user: Address;
    forward_payload: Slice as remaining;
}
```

Figure 18.2: The definition of the message in the Stream Pool contract (pixelswap/contracts/pixelswap_streampool_messages.tact#L33–L39)

Exploit Scenario

Off-chain components and UI projects mistakenly send the wrong value for the `fund_id` and `subaccount` fields, causing funds to be lost.

Recommendations

Short term, standardize the structure of the messages that share the same ID or same fields.

Long term, avoid duplication of message types. Even if the functions' data needs are different, having different names and structures for the same message is confusing and error-prone. Define shared messages in a single place.

Fix Review Status
After conducting a fix review, the team determined that this issue has been resolved.

19. The JettonFactory contract allows minting zero tokens

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-PXL-19
Target: `contracts/jetton/jetton_factory.tact`	

Description

The `mint` function of the `JettonFactory` trait does not validate that the amount of coins to mint is greater than zero, therefore allowing a mint of zero tokens.

```
fun mint(token_id: Int, to: Address, amount: Int) {
    let master_message: Cell = beginCell()
        .storeUint(395134233, 32) // opCode(0x178d4519) - InternalTransfer
        .storeUint(0, 64) // queryId
        .storeCoins(amount) // Mint Amount
        .storeAddress(myAddress()) // from_address
        .storeAddress(sender()) // response -> excesses
        .storeCoins(self.gas_send_master_message) // forward_ton_amount
        .storeBool(false) // isHaving the Custom_payload
        .endCell();
    send(SendParameters {
        to: self.jetton_address_of(token_id),
        value: self.mint_forward_amount + self.gas_send_mint,
        bounce: false,
        mode: SendIgnoreErrors,
        body: JETTONFACTORY__Mint {
            query_id: 0,
            to_address: to,
            forward_ton_amount: self.mint_forward_amount,
            message_body: master_message
        }.toCell()
    });
}
```

Figure 19.1: The `mint` function from the `JettonFactory` contract (pixelswap/contracts/jetton/jetton_factory.tact#L64–L86)

In the current state of the codebase, it is not possible to trigger this zero-amount mint for the LP tokens since the `add_liquidity` function checks for a minimum liquidity of 1000 to be added. However, if this same trait is used in other contracts, it can become an issue.

Recommendations

Short term, add a validation check to ensure the amount is greater than zero.

Long term, consider the possible edge cases in all functions, define the behavior, and implement test cases to ensure compliance.

Fix Review Status
After conducting a fix review, the team determined that this issue has not been resolved.

20. Incorrect gas calculations in several contracts

Severity: **Undetermined**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-PXL-20
Target: `contracts/pixelswap_streampool.tact`, `contracts/pixelswap_settlement.tact`	

Description

TON contracts, being a part of an asynchronous blockchain, can process messages in a non-deterministic order. In some cases, the incoming messages require the contract to perform changes in the storage, send new messages, and other gas-consuming tasks.

It is important to take into account the gas costs associated with executing functions or sending messages to ensure that the contract's final balance after execution does not drop below a threshold. Otherwise, if the balance is zero, the contract cannot pay for storage costs and can eventually be destroyed.

We noticed several places where the gas calculations were not correct or did not consider all of the costs that must be paid. We did not assess the exact severity of this issue because these calculations may have a long-reaching impact on the affected functions and further calls, and can depend on external factors unknown at the time.

- Swap message handler in `pixelswap_settlement.tact` should consider the storage fees, the gas consumed, and the gas for paused contract refund.
- Later, line 463 should check that the resulting amount is greater than `gas_check_swap_message`.
- Lines 454 and 468 should deduct `gas_consumed()`.
- `PlaceOrder` message handler in `pixelswap_streampool.tact` should consider the message transfer fee.
- `RemoveLiquidityJettonNotification` message handler should consider the `transfer_notification_handler` gas fee or use the remaining gas instead of `context().value`.
- Later, line 297 subtracts `remove_liquidity` gas twice because the call was already made. It is not checked that the remaining gas is more than the required `token0` and `token1` gas amounts.

Recommendations

Short term, ensure that the gas calculations correctly account for the message transfer fee, storage fee, computation fee, and forward TON amount in the mentioned places and all other places where they are calculated.

Long term, measure all execution paths and their gas requirements to ensure that the amounts are correctly calculated through tests.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

21. A privileged account can drain the PixelswapStreamPool contract

Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-PXL-21
Target: `fcontracts/pixelswap_streampool.tact`	

Description

In the `PixelswapStreamPool` contract, there are two ways to create new tokens and new trading pairs by sending `CreateToken` and `CreateTradingPair` messages: the first one requires being a privileged account, and the second one requires the sender to be enabled for adding tokens and pairs, and also paying a creation fee intended to avoid spamming.

Since the privileged account does not require paying the fee or sending TON at all, it is possible to drain the balance of the `PixelswapStreamPool` contract because the message handler sends the fee to the recipient in all cases.

```
receive(msg: CreateToken) {
    require(self.has_role(sender()) || context().value >=
self.config.token_creation_fee * MILLITON, "Incorrect token creation fee"); //
afr::allow-trap-at-input-boundary
    require(self.has_role(sender()) || self.config.enable_token_creation, "Token
creation not possible"); // afr::allow-trap-at-input-boundary
    require(self.tokens_config.get(msg.token_id) == null, "Token already exists");
// afr::allow-trap-at-input-boundary
    self.tokens_count += 1;
    self.tokens_config.set(msg.token_id, TokenConfig {
        // is_validated: false,
        is_active: true, // by default, tokens are active and can be used to create
pairs without validation
        // token_wallet_address: newAddress(0, 0x04), // Dummy address. Address 0
implies TON so we avoid that.
        // gas_forward_milliton: self.gas.default_token_forward_milliton,
        // gas_transfer: self.gas.default_token_transfer,
        // query_id: 0,
    });
    emit(TokenCreatedEvent { token_id: msg.token_id }.toCell());
    self.creation_fee_recipient.send_ton(self.config.token_creation_fee * MILLITON,
null);
}
```

Figure 21.1: Fee payment in `CreateToken` message handler (pixelswap/contracts/pixelswap_streampool.tact#L331–L346)

```
receive(msg: CreateTradingPair) {
```

```
    require(self.has_role(sender()) || context().value >=
(self.config.pair_creation_fee + self.gas.pair_creation) * MILLITON, "Incorrect pair
creation fee"); // afr::allow-trap-at-input-boundary
    require(self.has_role(sender()) || self.config.enable_pair_creation, "Pair
creation not possible"); // afr::allow-trap-at-input-boundary
    require(msg.token0_id.to_int() < msg.token1_id.to_int(), "Token0 must have a
lower address"); // afr::allow-trap-at-input-boundary
    require(msg.fee_bps >= 10 && msg.fee_bps <= 250, "Fee must be between 0.10% and
2.50%"); // afr::allow-trap-at-input-boundary
    require(msg.weight0 > 0 && msg.weight0 < 100, "weight0 must be between 1% and
99%"); // afr::allow-trap-at-input-boundary

...[snipped]...

    self.create_jetton_root(pair_id);
    emit(PairCreatedEvent { pair_id, token0_id: msg.token0_id, token1_id:
msg.token1_id, fee_bps: msg.fee_bps, weight0: msg.weight0 }.toCell());
    self.creation_fee_recipient.send_ton(self.config.pair_creation_fee * MILLITON,
null);
}
```

Figure 21.2: Fee payment in `CreateTradingPair` message handler (pixelswap/contracts/pixelswap_streampool.tact#L331–L346)

Exploit Scenario

Alice, a privileged account in the Stream Pool contract, decides to add new tokens and trading pairs to the system. Since she is not required to send additional value in her messages, she sends messages with the minimum amount possible to pay for the computation.

While the `PixelswapStreamPool` contract has a balance, she is able to add the tokens and pairs. However, once the balance becomes zero, the transactions will start failing, putting the contract at risk of not being able to pay storage fees.

Recommendations

Short term, check for the contract balance before sending the fee payment.

Long term, improve the test suite to consider all combinations of privileged and unprivileged users performing actions on the system. Check for balances before and after the transactions are executed, and ensure that the final balance is enough for storage fees.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

22. The fee recipient accounts cannot be changed in the Stream Pool contract

Severity: **Informational**	Difficulty: **Undetermined**
Type: Access Controls	Finding ID: TOB-PXL-22
Target: `contracts/pixelswap_streampool.tact`	

Description

The `creation_fee_recipient` and `protocol_fee_recipient` addresses are set once in the `PixelswapStreamPool` contract constructor and cannot be changed.

If, for some reason, access to those accounts is lost or compromised, the protocol will continue to send funds to them, increasing the financial damage over time and making it impossible for the team to recover funds.

Moreover, the `protocol_fee_recipient` account is not used in the `PixelswapStreamPool` contract.

Recommendations

Short term, consider having a privileged method to change these fee recipients.

Long term, determine if these accounts will be used, what security measures will be in place for setting and changing their values, and under what circumstances the change should be made.

Fix Review Status

After conducting a fix review, the team determined that this issue has not been resolved.

23. The gas checks in the PixelswapStreamPool contract are wrongly placed

Severity: **Informational**	Difficulty: High
Type: Data Validation	Finding ID: TOB-PXL-23
Target: `contracts/pixelswap_streampool.tact`	

Description

The gas checks in the `PlaceOrder` message receiver of the `PixelswapStreamPool` contract are placed after the `pair_status` update. This can lead to loss of funds for user and pair reserves and balances going out of sync.

The `PlaceOrder` message receiver of the `PixelswapStreamPool` contract adds two gas checks. It adds one at the end of the `process_order` internal function:

```
gas_spent += (
        token0_id_transfer_gas
        + token0_id_token_forward_milliton
        + token1_id_transfer_gas
        + token1_id_token_forward_milliton
        + self.gas.default_place_order
    ) * MILLITON;

if (msg.toCell().afr_require(context().value > gas_spent, self.gas.storage_reserve)) { return
ProcessOrderResult { ref_po: null, exec_res: inner_res , gas_spent: gas_spent}; } //
"Insufficient gas"

return ProcessOrderResult { ref_po: msg.ref_po, exec_res: exec_res, gas_spent: gas_spent };
```

Figure 23.1: Snippet of the `process_order` function of the `PixelswapStreamPool` (`pixelswap/contracts/pixelswap_streampool.tact#L667–L677`)

It adds another at the end of the receiver function:

```
if (p.exec_res != null) {
    if (msg.toCell().afr_require(context().value > p.gas_spent + storage_fees() +
gas_consumed(), self.gas.storage_reserve)) { return; }
    let remaining_gas: Int = context().value - storage_fees() - gas_consumed();
    p.exec_res!!.send_to_value(self.settlement, remaining_gas);
}
```

Figure 23.2: Snippet of the `PlaceOrder` receiver function of the `PixelswapStreamPool` (`pixelswap/contracts/pixelswap_streampool.tact#L183–L187`)

However, the `process_order` function updates the pair reserves in the `pair_status` map for the specified `pair_id` before these gas checks. If, for some reason, these gas checks fail, then it can have the following effect:

- If the gas check in the receiver function fails, then the `PlaceOrder` message will be sent back to the settlement contract and will add the input tokens back to the user's funding wallet balance.
- If the gas check in the `process_order` function fails while processing a nested `PlaceOrder` message, then only the current `PlaceOrder` and its nested `PlaceOrder` messages are returned to the settlement contract. This adds input tokens to the user's funding wallet balance only for these orders instead of all of the orders. The user also loses the output token of swaps for successfully executed `PlaceOrder` messages.
- In every case, the reserves of the specified pair and token balances of the settlement contract go out of sync, resulting in an unstable system state.

Currently, the gas checks in the `PixelswapFundingWallet` and `PixelswapSettlement` contracts ensure that the gas checks in the `PixelswapStreamPool` contract do not fail. However, future updates to the codebase could introduce a path to fail these gas checks.

Recommendations

Short term, place the above-mentioned gas checks before calls to the state-modifying functions to ensure the correct state in case of gas check failure.

Long term, document the system state specification with user actions and their effect on the system state to identify potential issues arising from complex user interactions.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

24. Users cannot deposit only Jetton to their funding wallet

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-PXL-24
Target: `contracts/pixelswap_settlement.tact`	

Description

The gas check in the `OrderJettonNotification` handler of the `PixelswapSettlement` contract ensures that the TON sent by the user is more than the value of the `gas_check_jetton_notification` parameter, which is 500 millitons, and that the `ton_amt` is calculated by subtracting the `gas_for_incoming_jetton_transfer` value, which is 400 millitons, from the message value.

Because of this, the user needs to deposit at least 100 millitons of TON along with any Jetton deposit, and there is no way for a user to deposit only Jetton to their funding wallet.

```
require(context().value >= self.gas_config.gas_check_jetton_notification * MILLITON,
"Insufficient gas"); // afr::allow-trap-at-input-boundary
let ton_amt: Int = context().value - self.gas_config.gas_for_incoming_jetton_transfer *
MILLITON; // token0 is TON
```

Figure 24.1: A snippet of the `OrderJettonNotification` handler in the `PixelswapSettlement` (`pixelswap/contracts/pixelswap_settlement.tact#L328–L329`)

Recommendations

Short term, accept the TON amount to deposit as a message parameter instead of calculating it in the contract, and return the excess TON sent as the message value.

Long term, implement test cases for all real-life use cases to ensure expected behavior from the smart contracts.

Fix Review Status

After conducting a fix review, the team determined that this issue has been resolved.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.

Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Findings

This appendix contains findings that do not have immediate or obvious security implications. However, they may facilitate exploit chains targeting other vulnerabilities, become easily exploitable in future releases, or decrease code readability. We recommend fixing the issues reported here.

- **Remove unnecessary conditional in `AccessControlSingleRole`.** The `del()` function returns true or false depending on the existence of the value to be deleted.
- **Use a more appropriate name for the `IncreaseTokenBalance` message.** The message is sent to the `PixelswapSettlementVault` contract for both increasing and decreasing the token balance.
- **Fix the comment in `AccessControlSIngleRoleNonTransferrable`.** The trait implements the owner role, which is not transferable.
- **Remove the unused `is_token_info` variable.**
- **Avoid using magic constants.** Literal values are set directly in code (e.g., #1, #2, #3, #4). This makes the functions harder to read, and more difficult to maintain in case the requirements are updated.
- **Remove useless code.** The `add_liquidity` and `remove_liquidity` functions calculate the amounts and fees using the `calc_add_lp_*` and `calc_remove_lp` functions from `univ2_math.tact`. The math functions explicitly state that the fees returned are always 0; however, the Stream Pool caller calculates protocol fees and updates the `fees_payable` fields in the pair status. Besides having no effect, this code increases the computation cost and makes the functions more complex to understand.
- **Remove duplicated code.** There are some instances of duplicated code, or sections of code that could be abstracted into separate functions to improve readability. Some examples include #1, #2, and #3. Having duplicated code increases the likelihood of adding bugs when only one of the instances is modified, and also increases the complexity of the function, degrading readability.
- **Remove debug calls and commented code statements.** In several contracts, most notably on `PixelswapStreamPool`, there are a significant number of `dump` calls, debug print calls, and code statements commented out. The debug statements should be converted into tests, validating the state before and after the intended calls.

D. Incident Response Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which PixelSwap Labs will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific system components.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

E. Testing Improvement Recommendations

This appendix aims to provide general recommendations on improving processes and enhancing the quality of the PixelSwap DEX test suite.

Identified Testing Deficiencies

During the audit, we encountered a number of issues that could have been prevented or minimized through better test practices and improved test coverage (e.g., TOB-PXL-1, TOB-PXL-3, and TOB-PXL-10, among others). We identified several deficiencies in the test suite that could make further testing and development more difficult and thereby reduce the likelihood that the test suite will find security issues:

- The test suite does not fully cover all of the contracts and functions in the codebase, and it does not test for all edge cases.
- Multiple negative unit tests do not test for a specific failure case; instead, they will pass on any failure inside the test body. This reduces confidence that the test suite will catch test failures, makes tests and issues harder to triage and debug, and could lead to false negatives.
- Tests do not check for event emission (external messages) or transaction results and only rely on checking balances or balance deltas for specific accounts, ignoring other relevant checks.
- The test directory lacks a coherent structure; all DEX unit tests are placed inside the same file. This can make it difficult to identify which components and behaviors are being tested, making it harder to onboard developers.

To address these deficiencies and improve the PixelSwap DEX's test coverage and processes, we recommend that the PixelSwap Labs team define a clear testing strategy and create guidelines on how testing is performed in the codebase. Our general guidelines for improving test suite quality are as follows:

1. **Define a clear test directory structure.** A clear directory structure helps organize the work of multiple developers, makes it easier to identify which components and behaviors are being tested, and gives insight into the overall test coverage.
2. **Write a design specification of the system, its components, and its functions in plain language.** Defining a specification can allow the team to more easily detect bugs and inconsistencies in the system, reduce the likelihood that future code changes will introduce bugs, improve the maintainability of the system, and allow the team to create a robust and holistic testing strategy.

3. **Use the function specifications to guide the creation of unit tests.** Creating a specification of all preconditions, postconditions, failure cases, entry points, and execution paths for a function will make it easier to maintain high test coverage and identify edge cases.

4. **Use the interaction specifications to guide the creation of integration tests.** An interaction specification will make it easier to identify the interactions that need to be tested and the external failure cases that need to be validated or guarded against, and it will help identify issues related to access controls and external calls.

5. **Use fork testing for integration testing with third-party smart contracts and to ensure that the deployed system works as expected.** Fork testing can be used to test interactions between the protocol contracts and third-party smart contracts by providing an environment that is as close to production as possible. Additionally, fork testing can be used to identify whether the deployed system is behaving as expected.

6. **Keep the team updated on new tooling and tool releases.** Since Tact and FunC are relatively new languages and not many tools are developed for them yet, it is recommended for the team to be updated on the development of third-party tools such as fuzzers, mutation testers, and static and dynamic analyzers, among others. Additionally, keep track of new compiler releases, bug fixes, and their potential effects on the project codebase.

Directory Structure

Creating a specific directory structure for the system's tests will make it easier to develop and maintain the test suite and find coverage gaps. This section contains brief guidelines on defining a directory structure.

- Create individual directories for each test type (e.g., `unit/`, `integration/`, `fork/`, `fuzz/`) and for the utility contracts. The individual directories can be further divided into directories based on components or behaviors being tested.

- Create a single base contract that inherits from the shared utility contracts and is inherited by individual test contracts. This will help reduce code duplication across the test suite.

- Create a clear naming convention for test files and test functions to make it easier to filter tests and understand the properties or contracts that are being tested.

Unit Testing

The provided Blueprint test suite covers a limited number of scenarios and situations. Some tests are failing, and there are sparse negative test cases. We provide the following general recommendations based on our findings:

- **Define a specification for each function** and use it to guide the development of the unit tests. Examples of function specifications can be found in the Function Specification Example section above.
- **Improve the unit tests' coverage** so that they test all functions and contracts in the codebase. Use coverage reports and mutation testing to guide the creation of additional unit tests.
- **Use positive unit tests** to test that functions and components behave as expected. Ideally, each unit test should test a single property, with additional unit tests for edge cases. The unit test should test that all expected side effects are correct.
- **Improve the use of negative unit tests** by not defining test cases that pass on any failure within a test body; instead, each negative unit test should test for a specific failure case.
- **Ensure that no tests fail under normal conditions.** Any failing tests should be an alert that the code is not working as expected, and the team should fix the issue immediately.

Integration and Fork Testing

Integration tests build on unit tests by testing how individual components integrate with each other or with third-party contracts. It can often be useful to run integration testing on a fork of the network to make the testing environment as close to production as possible and to minimize the use of mock contracts whose implementation can differ from third-party contracts. We provide the following general recommendations for performing integration and fork testing:

- **Use the interaction specification to develop integration tests.** Ensure that the integration tests aid in verifying the interactions specification.
- **Identify valuable input data for the integration tests** that can maximize code coverage and test potential edge cases.
- **Use negative integration tests**, similar to negative unit tests, to test common failure cases.
- **Use fork testing to build on top of the integration testing suite.** Fork testing will aid in testing third-party contract integrations and the proper configuration of the system once it is deployed.

F. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

From November 25 to November 27, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the PixelSwap Labs Ltd. team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 24 issues described in this report, PixelSwap Labs Ltd. has resolved 19 issues and has not resolved the remaining five issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	The BulkInternalTransfer receiver computes the wrong total amounts	Resolved
2	Granting the role to the master funding contract on a funding wallet contract will result in draining the funding wallet balance	Resolved
3	Users can inflate their funding wallet balance by sending an expired Swap message to the settlement contract	Resolved
4	SettlementVault balances are not updated in the PlaceOrder_Partial_2 receiver of the settlement contract	Resolved
5	Lack of validation of PixelswapStreamPool configuration parameters	Resolved
6	Users can drain the TON balance of the PixelswapSettlement contract	Resolved
7	Users can avoid paying the gas fee for the token creation transaction	Resolved
8	The tokens_count initial value in the PixelswapStreamPool contract is zero	Resolved
9	The returned TON amount from an order execution result is ignored	Resolved
10	Wrong formula used for LP amount calculation	Resolved

11	Users can use nested PlaceOrders to drain the settlement contract	Resolved
12	The LP tokens are never burned by the Stream Pool contract	Resolved
13	Lack of the pair_id and token_id validation in the PixelswapStreamPool contract	Resolved
14	The value attached to messages is not checked to be positive	Unresolved
15	The current balance is not checked before sending a message with a non-zero value	Unresolved
16	The exec_id value is not validated for the internal orders in nested PlaceOrder messages	Unresolved
17	The token_balance get function reverts if a user balance is 0	Resolved
18	Different parsing formats for Jetton notification messages	Resolved
19	The JettonFactory contract allows minting zero tokens	Unresolved
20	Incorrect gas calculations in several contracts	Resolved
21	A privileged account can drain the PixelswapStreamPool contract	Resolved
22	The fee recipient accounts cannot be changed in the Stream Pool contract	Unresolved
23	The gas checks in the PixelswapStreamPool contract are wrongly placed	Resolved
24	Users cannot deposit only Jetton to their funding wallet	Resolved

Detailed Fix Review Results

TOB-PXL-1: The BulkInternalTransfer receiver computes the wrong total amounts
Resolved in PR #505. The `BulkInternalTransfer` receiver function of the `PixelswapFundingWallet` contract now correctly updates the loop index variable.

TOB-PXL-2: Granting the role to the master funding contract on a funding wallet contract will result in draining the funding wallet balance
Resolved in PR #559. The role condition check order has been reversed to fix the issue.

TOB-PXL-3: Users can inflate their funding wallet balance by sending an expired Swap message to the settlement contract
Resolved in PR #589. The `PixelswapSettlement` contract has been updated to throw an error if the `extra_tokens` has a `TokenInfo` with the `token_is_input` attribute set to true.

TOB-PXL-4: SettlementVault balances are not updated in the PlaceOrder_Partial_2 receiver of the settlement contract
Resolved in PR #558. The `PlaceOrder_Partial_2` message receiver function of the `PixelswapSettlement` contract now correctly updates the `SettlementVault` balances to reflect the transfer of funds between funding and execution contracts

TOB-PXL-5: Lack of validation of PixelswapStreamPool configuration parameters
Resolved in PR #557 and PR #677. The gas fee configuration values are not validated to be within a limit.

TOB-PXL-6: Users can drain the TON balance of the PixelswapSettlement contract
Resolved in PR #556. Now the sum of the `token_amt`, `forward_milliton`, and `gas_transfer` values is deducted from the user's funding wallet while processing the `WithdrawFunds` message.

TOB-PXL-7: Users can avoid paying the gas fee for the token creation transaction
Resolved in PR #555. The `CreateToken` message receiver has been updated to ensure that the user and admin both provide enough TON for the required gas fee.

TOB-PXL-8: The tokens_count initial value in the PixelswapStreamPool contract is zero
Resolved in PR #553. The `init` function now updates the `tokens_count` variable to track the correct number of tokens added to the `PixelswapStreamPool` contract.

TOB-PXL-9: The returned TON amount from an order execution result is ignored
Resolved in PR #552. Now the `SettlementVault` balances are updated for the TON amount value in the order execution result even when the `msg.token_id` is not null.

TOB-PXL-10: Wrong formula used for LP amount calculation
Resolved in PR #564. The updated code uses the correct formula for LP amount calculation.

TOB-PXL-11: Users can use nested PlaceOrders to drain the settlement contract
Resolved in PR #575. Now the `spend_order_inputs` function of the `PixelswapFundingWallet` contract calls itself recursively to correctly update the user's funding wallet balance.

TOB-PXL-12: The LP tokens are never burned by the Stream Pool contract
Resolved in PR #560. The PixelSwap Labs Ltd. team updated the `burn` function of the `JettonFactory` trait contract to send the `burn` message to the Jetton wallet owned by the Stream Pool contract.

TOB-PXL-13: Lack of the pair_id and token_id validation in the PixelswapStreamPool contract
Resolved in PR #605. The `pair_id` and `token_id` are now validated correctly against the user-provided `token_id`.

TOB-PXL-14: The value attached to messages is not checked to be positive
Unresolved. The issue has not been resolved.

The client provided the following context for this finding's fix status:

> *Acknowledged. Passing a negative value here will just cause a system error. This will also never happen in the code logic.*

TOB-PXL-15: The current balance is not checked before sending a message with a non-zero value
Unresolved. The issue has not been resolved.

The client provided the following context for this finding's fix status:

> *When exploring a fix we concluded that aborting with AFRs is not sufficient in many cases. In many scenarios the transaction would already be completed when (and if) balance would fall below zero, and in most cases it's either impossible or too complicated to recover from this faulty state.*
>
> *Instead we conducted an extensive review of the code logic to make sure a) no code path sends out more TON than it should have, accounting for all credits and debits, b) prior to sending TON the code always checks for the remaining balance of GAS_RESERVE (which is configured to a different value per smart contract), and c) gas out + processing fee < gas in.*
>
> *In some specific scenarios, for example making a call to a contract after a long idle time, c cannot be guaranteed due to the storage fee. We do not have a solution for this, and*

this problem is also existing for native TON smart contracts such as Jetton contracts, so we deem it good enough. To mitigate the risk we initially fund the contracts with enough TON to cover storage fees for years.

TOB-PXL-16: The exec_id value is not validated for the internal orders in nested PlaceOrder messages
Unresolved. The issue has not been resolved.

The client provided the following context for this finding's fix status:

> *Acknowledged. Architecture of nested orders will be changed.*

TOB-PXL-17: The token_balance get function reverts if a user balance is 0
Resolved in PR #562. The `token_balance` get function now returns 0 instead of reverting.

TOB-PXL-18: Different parsing formats for Jetton notification messages
Resolved in PR #565. The `RemoveLiquidityJettonNotification` message has been updated to follow a common parsing format for the jetton notification message.

TOB-PXL-19: The JettonFactory contract allows minting zero tokens
Unresolved. The issue has not been resolved.

The client provided the following context for this finding's fix status:

> *Acknowledged. Testing showed that negative transfer amounts will be rejected by the system. This will also never happen in the code logic.*

TOB-PXL-20: Incorrect gas calculations in several contracts
Resolved in PR #548. The codebase has been updated to correct gas calculations.

TOB-PXL-21: A privileged account can drain the PixelswapStreamPool contract
Resolved in PR #555. The `CreateToken` and `CreateTradingPair` message receiver functions now send the creation fee only when they are called by a non privileged user.

TOB-PXL-22: The fee recipient accounts cannot be changed in the Stream Pool contract
Unresolved. The issue has not been resolved.

The client provided the following context for this finding's fix status:

> *Acknowledged. The protocol fee collector will be a peripheral smart contract that is outside the scope of this audit.*

TOB-PXL-23: The gas checks in the PixelswapStreamPool contract are wrongly placed
Resolved in PR #543. The gas checks in the `PixelswapStreamPool` contract have been removed because they are already included in the entrypoint contracts.

TOB-PXL-24: Users can not deposit only Jetton to their funding wallet
Resolved in PR #548. The `gas_for_incoming_jetton_transfer` has been updated to be equal to the the `gas_check_jetton_notification` parameter, which is 500 millitons.

G. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.