Itos Security Review

Pashov Audit Group

Conducted by: ast3ros, piken, Ch_301

May 24th 2025 - May 25th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **itos-finance/Commons** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Itos

Itos Finance is a platform that uses an on-chain derivative engine to create customizable financial payoffs, combined with a portfolio management system that enables cross-margin trading, siloed portfolios, performance tracking, while allowing multiple protocols to share liquidity. The scope was focused on RFT contract, which provides utilities for handling token requests and payments between contracts, including both non-reentrant and reentrant settlement functions, with support for ERC165-compliant RFT payers.

5. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash - cba52b0e2fbd57746869d15ee48bbd83667dc38a

fixes review commit hash - bb73627474b3fdb8e1e341132a146171b603778e

Scope

The following smart contracts were in scope of the audit:

- `RFT`

7. Executive Summary

Over the course of the security review, ast3ros, piken, Ch_301 engaged with Itos to review Itos. In this period of time a total of **6** issues were uncovered.

Protocol Summary

Protocol Name	Itos
Repository	https://github.com/itos-finance/Commons
Date	May 24th 2025 - May 25th 2025
Protocol Type	Derivative engine

Findings Count

Severity	**Amount**
High	1
Medium	1
Low	4
Total Findings	**6**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Payers exploit reentrantSettle to bypass payments with self-transfers	High	Resolved
[M-01]	Potential gas griefing attack from malicious payer	Medium	Acknowledged
[L-01]	RFTLib incompatible with fee-on-transfer tokens	Low	Acknowledged
[L-02]	Reentrancy protection bypass allows unauthorized reentrant calls	Low	Resolved
[L-03]	Manipulation in settle and reentrantSettle lets payers bypass validation	Low	Acknowledged
[L-04]	Unbounded loops in reentrantSettle() risk out-of-gas errors in calls	Low	Acknowledged

8. Findings

8.1. High Findings

[H-01] Payers exploit `reentrantSettle` to bypass payments with self-transfers

Severity

Impact: High

Likelihood: Medium

Description

The `reentrantSettle` function in RFTLib contains a vulnerability that allows malicious contracts to implement the `IRFTPayer` interface to completely avoid payment obligations. The vulnerability is from how the function tracks cumulative balance changes (`transact.delta`) across nested calls.

When a contract requests tokens using `reentrantSettle`, the function:

- Records the expected balance change in `transact.delta[token]`.
- Calls the payer's `tokenRequestCB` if they implement `IRFTPayer`.
- Validates final balances against the cumulative delta.

However, a malicious payer can exploit this by calling back into the requesting contract during `tokenRequestCB`, causing it to call `reentrantSettle` again with `payer = requester address` and a negative amount. This results in a self-transfer that:

- Doesn't change the contract's actual token balance.
- Zeros out the tracked delta (`+x` from original request, `-x` from the callback).
- Passes the final balance validation despite no tokens being received.

```
function reentrantSettle(
>>>     address payer, // @audit payer is set to requester
        address[] memory tokens,
        int256[] memory balanceChanges,
        bytes memory data
    ) internal returns (bytes memory cbData) {
         ...
            // Handle and track all balance changes.
            int256 change = balanceChanges[i];
            if (change < 0) {
>>>             TransferHelper.safeTransfer(token, payer, uint256
//(-change)); // @audit if payer == address(this), this is a self-transfer
            }
            // If we want tokens we transfer from when it is not an RFTPayer.
            // Otherwise we wait to request at the end.
            if (change > 0 && !isRFTPayer) {
                TransferHelper.safeTransferFrom(token, payer, address
                  (this), uint256(change));
            }

            // Handle bookkeeping.
>>>         transact.delta[token] += change; // @audit delta is reduced because
// change is negative
        }
         ...
        }
    }
```

Consider a scenario:

- VictimContract calls `reentrantSettle(MaliciousPayer, +1 ETH)` -> transact.delta[token] = +1 ETH
- `MaliciousPayer.tokenRequestCB` executes: -> Calls VictimContract to trigger `reentrantSettle(VictimContract, -1 ETH)`
- VictimContract self-transfers 1 ETH (balance unchanged) -> transact.delta[token] = +1 ETH - 1 ETH = 0
- Final validation: Expected delta (0) == Actual change (0) -> Attack succeeds, MaliciousPayer pays nothing

It leads to loss of fund for the VictimContract.

Recommendations

Prevent self-transfers in RFTLib**: Add a check to ensure `payer != address(this)`:

```
function reentrantSettle(...) internal returns (bytes memory cbData) {
+   require(payer != address(this), "Self-transfer not allowed");
    ...
}
```

8.2. Medium Findings

[M-01] Potential gas griefing attack from malicious payer

Severity

Impact: Medium

Likelihood: Medium

Description

`RFTLib` provides a series of functions which allow caller to request token payment from a RFT payer by calling `IRFTPayer(payer).tokenRequestCB()`.

When executing `RFTLib#settle()` or `RFTLib#reentrantSettle()`, it will verify that the actual balance change matches the expected amount, preventing payers from evading payment.

However, the caller doesn't know whether `payer` is trustworthy. Since there is no gas cap on the call of `IRFTPayer(payer).tokenRequestCB()`, an attacker can craft a malicious payer contract and caller could be subject to gas griefing.

Recommendations

Allow caller to set gas cap on call of `payer#tokenRequestCB()`. Besides the call of `payer#supportsInterface()` should have a fixed gas limit.

8.3. Low Findings

[L-01] RFTLib incompatible with fee-on-transfer tokens

When the RFT library requests tokens from a non-RFTPayer address (EOA or non-RFTPayer contract), it uses `safeTransferFrom` to pull the exact requested amount. However, with `fee-on-transfer` tokens, the actual received amount will be less than the requested amount due to transfer fees being deducted.

The issue occurs in both `settle` and `reentrantSettle` functions:

```
function settle(
        address payer,
        address[] memory tokens,
        int256[] memory balanceChanges,
        bytes memory data
    ) internal returns (int256[] memory actualDeltas, bytes memory cbData) {
         ...
            // If we want tokens we transfer from when it is not an RFTPayer.
            // Otherwise we wait to request at the end.
            if (change > 0 && !isRFTPayer) {
                TransferHelper.safeTransferFrom(token, payer, address
                  (this), uint256(change));
            }

        ...

            // Validate our balances.
            uint256 finalBalance = IERC20(token).balanceOf(address(this));
            actualDeltas[i] = U256Ops.sub(finalBalance, startBalances[i]);
            if (actualDeltas[i] < balanceChanges[i]) {
                revert InsufficientReceive
                  (token, balanceChanges[i], actualDeltas[i]);
            }


        transact.status = ReentrancyStatus.Idle;
    }
```

```
function reentrantSettle(
        address payer,
        address[] memory tokens,
        int256[] memory balanceChanges,
        bytes memory data
    ) internal returns (bytes memory cbData) {
       ...
            // If we want tokens we transfer from when it is not an RFTPayer.
            // Otherwise we wait to request at the end.
            if (change > 0 && !isRFTPayer) {
                TransferHelper.safeTransferFrom(token, payer, address
                  (this), uint256(change));
            }

            // Handle bookkeeping.
            transact.delta[token] += change;
        }
        ...
    }
```

It's recommended to calculate the actual received amount: Instead of assuming the full amount is received, calculate the actual difference in balance before and after transferring.

[L-02] Reentrancy protection bypass allows unauthorized reentrant calls

The RFTLib implements reentrancy protection between `settle()` and `reentrantSettle()` functions, but fails to protect against reentrancy through `request()` and `requestOrTransfer()` functions.

While `settle()` and `reentrantSettle()` properly check and update the `ReentrancyStatus`, the `request()` and `requestOrTransfer()` functions make external calls to `IRFTPayer(payer).tokenRequestCB()` and `isSupported()` without any reentrancy protection. This allows a malicious payer contract to reenter and call either `settle()` or `reentrantSettle()`, potentially bypassing intended transaction flow and balance validation logic.

The vulnerability occurs because:

- `request()` and `requestOrTransfer()` don't check or modify `transact.status`.
- External calls in these functions can trigger callbacks that reenter protected functions.
- This breaks the assumption that balance tracking in `settle()`/`reentrantSettle()` is atomic.

To resolve this, add reentrancy protection to `request()` and `requestOrTransfer()` functions by updating `ReentrancyStatus` to `Locked`.

[L-03] Manipulation in `settle` and `reentrantSettle` lets payers bypass validation

Both the `settle` and `reentrantSettle` functions in RFTLib use a balance accounting mechanism that assumes exclusive control over token balance changes during execution. Both functions track the starting balance and expected delta for each token, then validate the final balance against these values. However, during the `tokenRequestCB` callback, the payer has full execution control and can arbitrarily modify the requester's token balance by calling a separate, unrelated public function on the requester contract that also affects token balances, or by triggering a callback from another protocol the requester interacts with.

For example transfer less than requested, then use other mechanisms to increase the requester's balance (e.g., trigger third-party contract transfers, claimming rewards, token balance rebasing).

```
transact.startBalance[token] = IERC20(token).balanceOf(address(this));

   ...

   // Payer callback - can execute arbitrary code
   cbData = IRFTPayer(payer).tokenRequestCB
   //(tokens, balanceChanges, data); // @audit Payer callback - can execute arbitrary

   ...

   uint256 expectedBalance = U256Ops.add
     (transact.startBalance[token], expectedDelta);
   uint256 finalBalance = IERC20(token).balanceOf(address
   //(this)); // @audit Final validation assumes only tracked changes occurred
   if (finalBalance < expectedBalance) {
      revert InsufficientReceive(token, expectedDelta, actualDelta);
   }
```

It can lead to the loss of funds for the requester.

Recommendations

- Limit calling `settle` and `reentrantSettle` to trusted payer.
- For untrusted payers, use `transferFrom` flow.

[L-04] Unbounded loops in `reentrantSettle()` risk out-of-gas errors in calls

The `reentrantSettle()` function contains multiple unbounded loops that can lead to out-of-gas errors, particularly problematic since this function is designed to be called multiple times in nested/reentrant scenarios within the same transaction. The gas consumption compounds across nested calls, making the function susceptible to gas limit failures.

The `reentrantSettle()` function contains several gas-intensive operations that scale with the number of tokens and the depth of reentrancy:

- **Primary processing loop** - Iterates through all tokens for transfers and bookkeeping:

```
for (uint256 i = 0; i < tokens.length; ++i) {
    address token = tokens[i];
    // ... token processing including potential transfers and storage updates
}
```

- **Cleanup loop in outer context** - Iterates through all accumulated tokens:

```
if (outerContext) {
    uint256 lastIdx = transact.tokens.length - 1;
    for (uint256 i = 0; i <= lastIdx; ++i) {
        // ... balance validation and cleanup operations
    }
}
```

- **External callback** - Makes calls to IRFTPayer.tokenRequestCB() which must process token arrays:

```
if (isRFTPayer) {
    cbData = IRFTPayer(payer).tokenRequestCB(tokens, balanceChanges, data);
}
```

This could lead to:

- Transaction failures due to gas limit exceeded.
- Potential for griefing attacks by forcing expensive operations.
- Denial of service for protocols relying on `reentrantSettle()`.

Recommendations

Add maximum bounds for token arrays to prevent excessive gas consumption.