MeMeStore Audit Report

Mon Jul 08 2024

contact@bitslab.xyz https://twitter.com/tonbit_

TonBit

MeMeStore Audit Report

1 Executive Summary

1.1 Project Information

Description	A protocol to pre-purchase meme coins and add liquidity automatically
Type	DeFi
Auditors	TonBit
Timeline	Fri Jun 14 2024 - Mon Jul 08 2024
Languages	Tact
Platform	Ton
Methods	Architecture Review, Unit Testing, Manual Review
Source Code	https://github.com/MeMeStoreJack/memestore-contracts-ton
Commits	0ae5de03c7d6a404ed95a626735a3ef82e960b4d a9dcb39423aa71266386a6f1223690fae46d267e 82d9a21d9ff42faf28c72df8fd9291cd505a17ef 453f2f7b3b407364fc3942a22f821ddbca7907dd 235db8d2ca04c4b0eb0fa169dc9c315253069f14 1c0b2cdcb0728102272ecf2835067d6b649bc3ce

1.2 Files in Scope

The following are the SHA1 hashes of the original reviewed files.

ID	File	SHA-1 Hash
STO	sources/stonfi.tact	6725a3101cbab2f23465fdf195e0c7ec053fa01e
RPA	sources/ReferrerParent.tact	7f83729501c700e359cd90efc07feb736bc1a189
PAY	sources/Payouts.tact	e7a6948615c5da3ad95552162f1a69b55aba6135
MES	sources/messages.tact	7d27526a3578625a51a30e6c2cb097da41190f64
JET	sources/jetton.tact	04f243d48e655107839d42c7ca5d738bf6b301bd
FMJ	sources/FairMintJetton.tact	a37e6e90ea388825df5c8145561a8b019c354cdd
BCJ	sources/BondCurveJetton.tact	804e4c3c589a62a7dd148f9d403d1e304fdf566a

1.3 Issue Statistic

Item	Count	Fixed	Acknowledged
Total	11	11	0
Informational	0	0	0
Minor	3	3	0
Medium	2	2	0
Major	5	5	0
Critical	1	1	0

1.4 TonBit Audit Breakdown

TonBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Integer overflow/underflow by bit operations
- Number of rounding errors
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting
- Unchecked CALL Return Values

1.5 Methodology

The security team adopted the **"Testing and Automated Analysis"**, **"Code Review"** strategy to perform a complete security test on the code in a way that is closest to the real attack. The main entrance and scope of security testing are stated in the conventions in the "Audit Objective", which can expand to contexts beyond the scope according to the actual testing needs. The main types of this security audit include:

(1) Testing and Automated Analysis

Items to check: state consistency / failure rollback / unit testing / value overflows / parameter verification / unhandled errors / boundary checking / coding specifications.

(2) Code Review

The code scope is illustrated in section 1.2.

(3) Audit Process

- Carry out relevant security tests on the testnet or the mainnet;
- If there are any questions during the audit process, communicate with the code owner in time. The code owners should actively cooperate (this might include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- The necessary information during the audit process will be well documented for both the audit team and the code owner in a timely manner.

2 Summary

This report has been commissioned by MeMeStore to identify any potential issues and vulnerabilities in the source code of the MeMeStore smart contract, as well as any contract dependencies that were not part of an officially recognized library. In this audit, we have utilized various techniques, including manual code review and static analysis, to identify potential vulnerabilities and security issues.

During the audit, we identified 11 issues of varying severity, listed below.

ID	Title	Severity	Status
BCJ-1	Users Can Transfer Tokens from Their Wallets after the `SellInternal` Message is Executed	Critical	Fixed
BCJ-2	`last_token_price` is Calculated Incorrectly	Major	Fixed
BCJ-3	A Malicious User can Drain the Contract's Funds Using Dust Amounts	Major	Fixed
BCJ-4	User Fund Loss Due to Race Conditions	Major	Fixed
BCJ-5	Sell and SellInternalReply Bypass	Medium	Fixed
BCJ-6	Missing Parameter Validation in `SetTradeConfig` Function	Minor	Fixed
BCJ-7	Unused Function	Minor	Fixed
JET-1	Refund of Excess Gas Fees	Minor	Fixed
PAY-1	Depletion of Storage Fees	Major	Fixed

PAY-2	Signature Replay	Major	Fixed
PAY-3	Replace + with Bitwise OR (|)	Medium	Fixed

3 Participant Process

Here are the relevant actors with their respective abilities within the MeMeStore Smart Contract :

Owner

- Owner can initialize the contract through the `Initialize` message.
- Owner can set trade configurations through the `SetTradeConfig` message.
- Owner can stop the contract through the `Stop` message.
- Owner can resume the contract through the `Resume` message.
- Owner can withdraw funds from the contract by sending a `Withdraw` message, provided they are the owner of the contract.
- Owner can destroy the contract and transfer any remaining balance to themselves by sending a `Destroy` message, provided they are the owner of the contract.
- Owner can update the token content through `TokenUpdateContent` .
- Owner can close minting through `Owner: MintClose` .

User

- Users can buy tokens through the `Buy` message.
- Users can sell tokens through the `Sell` message.
- Users can add TON liquidity through the `add_ton_v4` message.
- Users can add meme token liquidity through the `add_meme_v4` message.
- Users can participate in minting tokens by sending a `FairMint` message.
- Users can create a pool with expired tokens by sending a `createPoolWithExpired` message after the minting period has ended.
- Users can initiate a payout by sending a `WithdrawTicket` message with specific details including value, nonce, expiry, and payout ID.
- Users can deposit funds into the contract by sending a `Deposit` message with a minimum value of 1 TON.
- Users can register a new referrer by sending a `NewReferrer` message, providing another address as the referrer, ensuring they are not referring themselves and

include a minimum value of 0.1 TON.

- Users can provide a wallet address through `ProvideWalletAddress`.
- Users can transfer tokens using `JettonDefaultWallet` through `TokenTransfer`.
- Users can burn tokens using `JettonDefaultWallet` through `TokenBurn`.

4 Findings

BCJ-1 Users Can Transfer Tokens from Their Wallets after the `SellInternal` Message is Executed

Severity: Critical

Status: Fixed

Code Location:

sources/BondCurveJetton.tact#216-291

Descriptions:

Currently the `Sell` operation is triggered in the contract through the following process:

1. The user sends a `Sell` message to the `BondCurveJetton` contract
2. The contract sends a `SellInternal` message to the user's jetton wallet to check if the user wallet has enough jettons in it
3. The user wallet sends a `SellInternal` message to `BondCurveJetton` contract to send `SellInternalReply` message for sell operation.

However, there is a loophole in the above operation, the user can transfer the jetton in his wallet to another account after the `SellInternal` is executed, so that the user can get the corresponding ton without paying any jetton.

Suggestion:

It is recommended that the contract performs the `sell` operation by receiving the `JettonTransferNotification` message.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

BCJ-2 `last_token_price` is Calculated Incorrectly

Severity: Major

Status: Fixed

Code Location:

sources/BondCurveJetton.tact#177

Descriptions:

In the `BondCurveJetton.buy()` function, the protocol calls `get_amount_out()` to calculate `token_amount` based on `trade_info.remain_amount`.

```
let token_amount: Int = self.get_amount_out(trade_info.remain_amount, true,
self.this_balance, self.this_supply);
    require(token_amount <= self.this_supply, "Insufficient token balance");
```

However, when calculating `last_token_price`, the code uses `(msg.ton_value * ton("1")) / token_amount`, which is incorrect.

```
let last_token_price: Int = (msg.ton_value * ton("1")) / token_amount;
```

The correct calculation for `last_token_price` should be `(trade_info.remain_amount * ton("1")) / token_amount`. Using the incorrect formula results in an overestimated price, which adversely affects the calculations within `trade_step = 2`.

Suggestion:

It is recommended to use `trade_info.remain_amount` instead of `msg.ton_value` to calculate `last_token_price`.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

BCJ-3 A Malicious User can Drain the Contract's Funds Using Dust Amounts

Severity: Major

Status: Fixed

Code Location:

sources/BondCurveJetton.tact#168-196

Descriptions:

In the `BondCurveJetton.buy()` function, each time the `buy()` function is called, the protocol sends `fee + ton("0.01")` to `self.trade_fee_receiver` and `ton("0.2")` to the sender.

```
  send(SendParameters{
        to: self.trade_fee_receiver,
        bounce: true,
        value: trade_info.fee_value + trade_info.referrer_amount +
trade_info.up_referrer_amount + ton("0.01"),
        mode:  SendIgnoreErrors,
        body: "trade+referrer fee".asComment()
      });

      self.this_balance += trade_info.remain_amount;
      let last_token_price: Int = (msg.ton_value * ton("1")) / token_amount;
      // Transfer from the master's jetton wallet to the buyer
      let init: StateInit = initOf JettonDefaultWallet(myAddress(), myAddress());
      send(SendParameters{
        to: contractAddress(init),
        bounce: true,
        value: ton("0.2"),
        mode:  SendIgnoreErrors,
        body: TokenTransfer{
          query_id: 0,
          amount: token_amount,
          destination: sender,
          response_destination: sender,
          custom_payload: null,
          forward_ton_amount: 0,
          forward_payload: beginCell().endCell().asSlice()
        }.toCell(),
```

```
        code: init.code,
        data: init.data
    });
```

A malicious user can repeatedly make purchases with dust amounts, draining the contract's funds, ultimately leading to insufficient liquidity in the protocol and preventing users from executing operations.

Suggestion:

It is recommended to estimate the required gas fee and ensure that `ctx.value` is greater than all fees plus the estimated gas. Currently, the minimum deposit of 0.2 is not sufficient to prevent malicious users from draining the contract funds.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

BCJ-4 User Fund Loss Due to Race Conditions

Severity: Major

Status: Fixed

Code Location:

sources/BondCurveJetton.tact#303-333

Descriptions:

When `BondCurveJetton` handles `JettonTransferNotification`, if the `trade_step` is in a stage beyond 1, the jettons cannot be sold. However, the user's jettons are still transferred to the contract and need to be returned to the user. This is due to potential race conditions, where the previous user's transaction may have already reached the threshold, preventing subsequent users from selling.

```
    receive(msg: JettonTransferNotification) {
        if (self.trade_step == 1) {
            let ctx: Context = context();
            let init_state: StateInit = initOf JettonDefaultWallet(myAddress(), myAddress());
            require(ctx.sender == contractAddress(init_state), "contract sender is not from
child");

            let ton_amount: Int = self.get_amount_out(msg.amount, false, self.this_balance,
self.this_supply);
            self.this_balance = self.this_balance - ton_amount;
            let last_token_price: Int = (ton_amount * ton("1")) / msg.amount;
            let trade_info: TradeInfo = self.split_fee(ton_amount);
            self.this_supply += msg.amount;

            send(SendParameters{
                to: self.trade_fee_receiver,
                bounce: true,
                value: trade_info.fee_value + trade_info.referrer_amount +
trade_info.up_referrer_amount + ton("0.01"),
                mode:  SendIgnoreErrors,
                body: "trade+referrer fee".asComment()
            });

            send(SendParameters{
```

```
            to: msg.sender,
            bounce: false,
            value: trade_info.remain_amount + ton("0.01"),
            mode:  SendIgnoreErrors,
            body: "remain amount".asComment()
        });
        emit(SellEvent{sender: msg.sender, amount: trade_info.remain_amount,
tokenAmount: msg.amount, lastTokenPrice: last_token_price, tradeInfo:
trade_info}.toCell());
        self.forward(msg.sender, "Sell finish".asComment(), false, null);
    }
  }
```

Suggestion:

It is recommended to refund the jettons to the user when the `trade_step` is greater than 1 during the processing of `JettonTransferNotification`.

Resolution:

This issue has been fixed. The client has refunded the jettons to the user.

BCJ-5 Sell and SellInternalReply Bypass

Severity: Medium

Status: Fixed

Code Location:

sources/BondCurveJetton.tact#223-271

Descriptions:

In `BondCurveJetton`, users can bypass message verification by writing their own contracts or initiating transfers from their wallets directly to the wallet held by `BondCurveJetton`. The purpose of these two messages (`Sell` and `SellInternalReply`) is merely for parameter checking, which can be done directly on the front-end. Then, users can initiate transfers to the wallet held by `BondCurveJetton`, making the process simpler and more straightforward.

```
    receive(msg: Sell){
        let ctx: Context = context();
        self.requireNotStopped();
        require(self.trade_step == 1, "not trade");
        require(msg.amount > 0, "value is zero");
        let ton_amount: Int = self.get_amount_out(msg.amount, false, self.this_balance,
self.this_supply);
        require(ton_amount > 0, "Sell amount too low");
        require(ton_amount <= self.this_balance, "Insufficient TON in reserves");

        let init: StateInit = initOf JettonDefaultWallet(sender(), myAddress());
        let seller: Address = contractAddress(init);

        send(SendParameters{
            to: seller,
            bounce: true,
            value: 0,
            mode: SendRemainingValue | SendIgnoreErrors,
            body: SellInternal{
                sender_from: sender(),
                amount: msg.amount
            }.toCell(),
            data: init.data,
```

```
        code: init.code
    });
}

receive(msg: SellInternalReply){
    let ctx: Context = context();
    let init: StateInit = initOf JettonDefaultWallet(msg.sender_from, myAddress());
    require(ctx.sender == contractAddress(init), "contract sender is not from child");
    send(SendParameters{
        to: contractAddress(init),
        bounce: true,
        value: 0,
        mode:  SendRemainingValue | SendIgnoreErrors,
        body: TokenTransfer{
            query_id: 0,
            amount: msg.amount,
            destination: myAddress(),
            response_destination: null,
            custom_payload: null,
            forward_ton_amount: ton("0.05"),
            forward_payload: beginCell().endCell().asSlice()
        }.toCell(),
        code: init.code,
        data: init.data
    });
}
```

Suggestion:

It is recommended to perform the calculations on the front-end and then directly initiate transfers to the wallet held by `BondCurveJetton`.

Resolution:

This issue has been fixed. The client has removed `Sell` and `SellInternalReply` messages.

BCJ-6 Missing Parameter Validation in `SetTradeConfig` Function

Severity: Minor

Status: Fixed

Code Location:

sources/BondCurveJetton.tact#27-301

Descriptions:

During the initialization of the contract, the `trade_config` parameter undergoes validation to ensure that critical addresses such as `swap_router` and `router_pton_wallet` are not set to zero addresses. However, the subsequent receive function that sets a new `trade_config` does not perform similar checks.

```
receive(msg: SetTradeConfig){
    require(sender() == self.owner, "only owner can set trade config");
    self.trade_config = msg.trade_config;
}
```

Suggestion:

It is recommended to add validation checks for the `SetTradeConfig` parameters when accepting the `SetTradeConfig` message.

Resolution:

This issue has been fixed. The client has added validation checks for the `SetTradeConfig` parameters when accepting the `SetTradeConfig` message.

BCJ-7 Unused Function

Severity: Minor

Status: Fixed

Code Location:

sources/BondCurveJetton.tact#345-348

Descriptions:

The function `add_liquidity_stonfi` is currently not being called anywhere. This function is intended to add liquidity by calling two separate functions: `sendMemeToWalletV4` and `sendTonToWalletV4`. However, since it is not utilized anywhere in the code, it serves no purpose and only adds unnecessary bloat to the contract.

```
fun add_liquidity_stonfi(token0: Address, token1: Address, amt0: Int, amt1: Int,
min_lp_out: Int) {
    self.sendMemeToWalletV4(token0, token1, amt0, min_lp_out);
    self.sendTonToWalletV4(amt1, min_lp_out);
}
```

Suggestion:

It is recommended to remove the `add_liquidity_stonfi` function.

Resolution:

This issue has been fixed. The client has removed the `add_liquidity_stonfi` function.

JET-1 Refund of Excess Gas Fees

Severity: Minor

Status: Fixed

Code Location:

sources/jetton.tact#336-342

Descriptions:

In jetton.tact, when handling Check `TransferReply` , if it is found that user-to-user transfers are not allowed, the excess gas fee should be refunded.

```
receive(msg: CheckTransferReply) {
    let ctx: Context = context();
    require(ctx.sender == self.master, "transfer_check_from is not jetton master");
    self.transferable = msg.res;
    if (msg.res) {
        self.token_transfer(msg.token_transfer);
    }
}
```

Suggestion:

It is recommended to refund the excess gas fees to the user.

Resolution:

This issue has been fixed. The client has refunded the excess gas fees to the user.

PAY-1 Depletion of Storage Fees

Severity: Major

Status: Fixed

Code Location:

sources/Payouts.tact#133-153

Descriptions:

In the `PayoutsMaster` contract, when a user performs a `withdraw` operation, the following check is executed:

```
require(myBalance() > ton("1.0") + self.storageReserve + ok.value, "Insufficient balance on master");
```

This is intended to verify that the master contract has sufficient balance. However, the `myBalance` function returns the contract's balance before the transaction, plus the `ctx.value` sent with the message. Given that the message is sent with the `SendRemainingValue` mode, if the `ctx.value` is relatively large, it could cause the `withdraw` operation to fail or potentially lead to the depletion of storage fees.

Suggestion:

It is recommended to ensure accurate balance verification, use `myBalance() - ctx.value` when checking the contract's balance.

Resolution:

This issue has been fixed. The client has used `myBalance() - context().value` when checking the contract's balance.

PAY-2 Signature Replay

Severity: Major

Status: Fixed

Code Location:

sources/Payouts.tact#59-74

Descriptions:

There is a potential risk of signature replay in the `Payout` contract. Let's simulate an attack scenario:

The balance of `PayoutsMaster` is `5`. If a user simultaneously initiates the following two messages:

1. `WithdrawTicket.nonce` = `2`, withdraw `4`, `PayoutBeacon` executes `TryPayout` successfully, TryPayout.nonce becomes `3`, PayoutsMaster has not yet executed PayoutOk.
2. `WithdrawTicket.nonce` = `3`, withdraw `2`, `PayoutBeacon` executes `TryPayout` successfully, `TryPayout.nonce` becomes `4`.

Eventually, the transaction with `nonce=2` will fail (insufficient balance), and `PayoutBeacon` will roll back the nonce to `3`.

If the user then uses the same signature to initiate `WithdrawTicket.nonce=3`, withdraw `2`, this transaction will execute successfully.

```
bounced(msg: bounced<PayoutOk>) {
    // Reset nonce if our message bounced
    self.nonce -= 1;
    self.forward(self.owner, "Payout failed".asComment(), false, null);
}
```

Suggestion:

It is recommended that when handling the `bounced<PayoutOk>` message, the nonce should not be decremented by 1.

Resolution:

This issue has been fixed. The client has removed `self.nonce -= 1;`.

PAY-3 Replace + with Bitwise OR (|)

Severity: Medium

Status: Fixed

Code Location:

sources/Payouts.tact#197

Descriptions:

In the `receive("Destroy") {}` function, the protocol sets the mode to `SendRemainingBalance + SendDestroyIfZero`.

```
receive("Destroy") {
    self.requireOwner();
    send(SendParameters{
        value: 0,
        mode: SendRemainingBalance + SendDestroyIfZero,
        to: self.owner,
        bounce: false,
        body: "Contract destroyed".asComment()
    });
}
```

However, according to the official documentation, it is recommended to use the bitwise OR operator (|) instead.

```
Note, that while adding (+) base modes together with optional flags is possible, it is discouraged due to the possibility of excess values. Use the bitwise OR (|) instead, as it's designed to work with such flag and bit manipulations of the mode.
```

Suggestion:

It is recommended to use the bitwise OR (|) instead.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

Appendix 1

Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.