TOM PUMP
Audit Report

Mon Oct 21 2024

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

TonBit

TOM PUMP Audit Report

1 Executive Summary

1.1 Project Information

Description	A TON pump and DEX protocol
Type	DEX
Auditors	TonBit
Timeline	Thu Sep 19 2024 - Mon Oct 21 2024
Languages	FunC
Platform	Ton
Methods	Architecture Review, Unit Testing, Manual Review
Source Code	https://github.com/TOM-PUMP/func-part
Commits	5bf631225f6e26ed2894576d60a92a0c5f73c905 9c6770f874c1a3d207dafa222ded468683309924 d28532a44ae1cdc59d8f1d63e69f2a036db189a5

1.2 Files in Scope

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

ID	File	SHA-1 Hash
JWA	contracts/jetton_wallet.fc	41159b22677e818569afde7fe9021f61bc9b42be
DWA	contracts/dex_wallet.fc	bac05a5aaffe333276a3bf9ac9f20b988467cae3
MAT	contracts/imports/math.fc	43ea019e971e8d011a3d99e5378edad93bdfbba5
OCA	contracts/imports/op-codes-amm.fc	9ccb51b8db9f46e31be2e3c0dd8ecb68eb9cc68b
PAR	contracts/imports/params.fc	e87b4e91cebbed58e6a23de768e2b63131639a91
AMU	contracts/imports/amm-minter-utils.fc	1087721d8daa26659798c5b3867533557232394e
STD	contracts/imports/stdlib.fc	48ba5be2230d6db462adb890e7b15ff0b36b90de
OCO	contracts/imports/op-codes.fc	7ef90bd084811dbc89046a1468498ca4a9dbce21
DPA	contracts/imports/discovery-params.fc	8c7a35f4878a074828d8eb43761c1765e3c9fa51
STO	contracts/imports/store.fc	0e39f549eabcd85da12d079662ca03accf397cc0
UTI	contracts/imports/utils.fc	cc7d2af21612c1306cb5cf067e19d63cb734f723

CON	contracts/imports/constants.fc	1a072c090130b98f010ad74bed33d6a42b199a2a
JUT	contracts/imports/jetton-utils.fc	971a7946ff8d80d8e1344e51eed8e31c2d8dfca1
DMI	contracts/dex_minter.fc	b69b26b7805961c3ea7619a377029b39c4ac0f05
PFU	contracts/pump_fun.fc	b3b69e6457521c4b29b3accb2be6812af878d472
JMI	contracts/jetton_minter.fc	85b1cd8c58afebd7dfd1f472060f50e21e9d1e8c

1.3 Issue Statistic

Item	Count	Fixed	Acknowledged
Total	15	5	10
Informational	1	0	1
Minor	1	1	0
Medium	5	1	4
Major	8	3	5
Critical	0	0	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 TOM to identify any potential issues and vulnerabilities in the source code of the TOM PUMP 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 15 issues of varying severity, listed below.

ID	Title	Severity	Status
AMU-1	When `token_reserves == 0` , the User's Share Calculation is Incorrect	Major	Acknowledged
AMU-2	The `total_supply` is Updated incorrectly when the First User Adds Liquidity	Medium	Acknowledged
DMI-1	Removing Liquidity Lacks Slippage Protection	Major	Acknowledged
DMI-2	Fee Omission Issue in `dex_minter` Swap Operations	Major	Acknowledged
DMI-3	The User cannot Receive all the Extra Assets	Medium	Acknowledged
DMI-4	Strict Liquidity Conditions Cause Transaction Failures	Medium	Acknowledged
JMI-1	Proposed Fee Deductions for Minting	Major	Fixed
JMI-2	Refunding Excess TON During Minting	Major	Acknowledged

JMI-3	When `op = burn`, the Protocol's Calculation of `reserve_return` is Incorrect	Major	Fixed
JMI-4	Minting and Burning Lack Slippage Protection	Major	Fixed
JMI-5	Centralization Risk	Major	Acknowledged
JMI-6	Single-step Ownership Transfer Can be Dangerous	Medium	Acknowledged
JMI-7	The Calculation of Tokens during Minting is Incorrect	Medium	Fixed
JMI-8	Precision Loss	Minor	Fixed
JMI-9	Storage in Jetton_minter Protocol	Informational	Acknowledged

3 Participant Process

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

Admin

- Admin can upgrade the contract via the message `op == OP_CODE_UPGRADE`.
- Admin can change the admin via the message `op == 3`.
- Admin can change the content via the message `op == 4`.
- Admin can perform a withdraw operation via the message `op == op::withdraw`.
- Admin can perform a deposit operation via the message `op == op::deposit`.

User

- Users can remove liquidity via the message `op == OP_BURN_NOTIFICATION`.
- Users can swap TON to Jetton via the message `op == OP_SWAP_TON`.
- Users can add liquidity or swap Jetton to TON via the message `op == OP_TRANSFER_NOTIFICATION`.
- Users can mint Jetton via the message `op == op::mint`.
- Users can burn Jetton to exchange for TON via the message `op == op::burn_notification`.

4 Findings

AMU-1 When `token_reserves == 0`, the User's Share Calculation is Incorrect

Severity: Major

Status: Acknowledged

Code Location:

contracts/imports/amm-minter-utils.fc#150

Descriptions:

In the `calculate_new_lp()` function, when `store::token_reserves == 0`, the calculation for `new_liquidity` is incorrectly defined as:

```
new_liquidity = square_root(ton_amount * token_amount) / MINIMUM_LIQUIDITY
```

Instead, `new_liquidity` should be calculated as:

```
new_liquidity = square_root(ton_amount * token_amount)  - MINIMUM_LIQUIDITY
```

This error results in the user receiving less liquidity than they should.

Suggestion:

It is recommended to change it to:

```
new_liquidity = square_root(ton_amount * token_amount)  - MINIMUM_LIQUIDITY
```

AMU-2 The `total_supply` is Updated incorrectly when the First User Adds Liquidity

Severity: Medium

Status: Acknowledged

Code Location:

contracts/imports/amm-minter-utils.fc#144-163

Descriptions:

In the `calculate_new_lp()` function, if `token_reserves == 0`, the protocol calculates the user's liquidity as follows:

```
new_liquidity = square_root(ton_amount * token_amount) / MINIMUM_LIQUIDITY
```

Then, it updates:

```
store::total_supply += new_liquidity
```

The issue here is that `store::total_supply` should also add `MINIMUM_LIQUIDITY`. This approach is intended to prevent the first liquidity provider from manipulating shares. In Uniswap V2, the liquidity obtained by the user is calculated after deducting `MINIMUM_LIQUIDITY`, with the protocol minting `MINIMUM_LIQUIDITY` to the zero address. Thus, when the first user adds liquidity, their total liquidity becomes:

```
liquidity = totalSupply - MINIMUM_LIQUIDITY
```

Suggestion:

It is recommended to add `MINIMUM_LIQUIDITY` to `store::total_supply` when the first user adds liquidity.

DMI-1 Removing Liquidity Lacks Slippage Protection

Severity: Major

Status: Acknowledged

Code Location:

contracts/dex_minter.fc#142-179

Descriptions:

In the `dex_minter` contract, when `op == OP_BURN_NOTIFICATION`, the protocol calls `remove_liquidity()` to remove liquidity and obtains `ton` and `token`. However, the protocol does not implement any slippage protection measures.

```
() remove_liquidity(
  int jetton_amount,
  slice from_address,
  int query_id,
  slice sender,
  int msg_value,
  int fwd_fee) impure inline {

  int ton_to_remove = muldiv(jetton_amount, store::ton_reserves, store::total_supply);
  int token_to_remove = muldiv(jetton_amount, store::token_reserves,
store::total_supply);

  throw_unless(ERROR::WRONG_JETTON_SENDER_ADDRESS,
     equal_slices(
        calculate_user_jetton_wallet_address(from_address,my_address(),
           store::jetton_wallet_code
        )
  , sender)
  );
```

In Uniswap V2, the protocol includes slippage protection by checking that the amount of tokens the user receives is greater than or equal to the user's expected minimum token amount. https://github.com/Uniswap/v2-periphery/blob/master/contracts/UniswapV2Router02.sol#L117-L118

```
    function removeLiquidity(
        address tokenA,
        address tokenB,
        uint liquidity,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) public virtual override ensure(deadline) returns (uint amountA, uint amountB) {
        address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
        IUniswapV2Pair(pair).transferFrom(msg.sender, pair, liquidity); // send liquidity to
pair
        (uint amount0, uint amount1) = IUniswapV2Pair(pair).burn(to);
        (address token0,) = UniswapV2Library.sortTokens(tokenA, tokenB);
        (amountA, amountB) = tokenA == token0 ? (amount0, amount1) : (amount1,
amount0);
        require(amountA >= amountAMin, 'UniswapV2Router: INSUFFICIENT_A_AMOUNT');
        require(amountB >= amountBMin, 'UniswapV2Router: INSUFFICIENT_B_AMOUNT');
    }
```

Suggestion:

It is recommended to implement slippage protection measures.

DMI-2 Fee Omission Issue in `dex_minter` Swap Operations

Severity: Major

Status: Acknowledged

Code Location:

contracts/dex_minter.fc#90-130

Descriptions:

In the `dex_minter` contract, during the swap operation, there are scenarios where no fees are charged, which could lead to financial losses for the protocol.

1. In `op == OP_SWAP_TON`, if slippage protection is triggered, the funds are refunded to the user. However, in this case, no gas fee or forward fee is charged, and the cost of refunding TON is covered by the protocol.

```
if ((amount_out < min_amount_out) | (amount_out > trgt_resvers)) {
    if (is_ton_src == true) {
        send_grams(sender, in_amount);
    } else {
        transfer_token(query_id, sender, in_amount, msg_value);
    }
    return ();
}

send_raw_message(msg, 1 + 2);
```

2. The same issue exists in `sub_op == OP_SWAP_TOKEN`.

Suggestion:

It is recommended to deduct the gas and forward fees during the swap operation.

DMI-3 The User cannot Receive all the Extra Assets

Severity: Medium

Status: Acknowledged

Code Location:

contracts/dex_minter.fc#116-123

Descriptions:

In the add liquidity process, the protocol transfers extra assets to `jetton_sender` under the condition that either `extra_jeton` or `extra_ton` is greater than `ADD_LIQUIDITY_DUST`.

```
else {
        int extra_ton = ton_liquidity - optimal_ton;
        int extra_jeton = jetton_amount - optimal_jetton;

        ;; return extra's
        if extra_jeton > ADD_LIQUIDTY_DUST {
            transfer_token(query_id, jetton_sender, extra_jeton, fwd_fee * 2 );  ;; TODO send
0.1TON use fwd
            ton_liquidity = optimal_ton;
        }
    elseif extra_ton > ADD_LIQUIDTY_DUST {
            send_grams(jetton_sender ,extra_ton);
            ;; token_liquidity = optimal_token;
        }
    }
```

However, the code uses an `if-elseif` structure, meaning the protocol will either transfer `jeton` to `jetton_sender` or `ton` to `jetton_sender`, but not both. This could result in a loss of funds for the user, as they may not receive all the extra assets they are entitled to.

Suggestion:

It is recommended to transfer all extra assets to the user.

DMI-4 Strict Liquidity Conditions Cause Transaction Failures

Severity: Medium

Status: Acknowledged

Code Location:

contracts/dex_minter.fc#101-124

Descriptions:

In the `dex_minter` contract, the condition checks for adding liquidity are overly strict, which may cause valid transactions to fail.

```
if ((optimal_ton <= ton_liquidity_min) | (optimal_jetton <= jeton_liquidity_min)) {
    revert_add_liquidity(
        ton_liquidity,
        jetton_amount,
        jetton_sender,
        query_id,
        msg_value,
        fwd_fee
    );
    should_revert = 1;
} else {
    int extra_ton = ton_liquidity - optimal_ton;
    int extra_jeton = jetton_amount - optimal_jetton;

    // Return excess liquidity
    if (extra_jeton > ADD_LIQUIDTY_DUST) {
        transfer_token(query_id, jetton_sender, extra_jeton, fwd_fee * 2); // TODO: Send
0.1TON using fwd
        ton_liquidity = optimal_ton;
    } else if (extra_ton > ADD_LIQUIDTY_DUST) {
        send_grams(jetton_sender, extra_ton);
        // token_liquidity = optimal_token;
    }
}
```

You can refer to Uniswap's implementation for reference: https://github.com/Uniswap/v2-periphery/blob/0335e8f7e1bd1e8d8329fd300aea2ef2f36dd19f/contracts/UniswapV2Router02.sol#L

L57.

Suggestion:

It is recommended to modify the code logic by referencing Uniswap’s approach.

JMI-1 Proposed Fee Deductions for Minting

Severity: Major

Status: Fixed

Code Location:

contracts/jetton_minter.fc#182-221

Descriptions:

In the Jetton_minter contract, when minting Jetton tokens, a protocol fee of 1/10 TON is charged based on the user's TON used to mint the Jetton tokens through `msg_value`. However, the forward fee (`fwd_fee`) and gas fee (`gas_fee`) are not deducted. If the gas fee and forward fee are higher than the protocol fee, the protocol may struggle to be profitable.

```
int fee = muldiv(1, const::nano(), 10);
throw_unless(75, msg_value - fee > muldiv(3, const::nano(), 10));
int buy_value = msg_value - fee
```

Suggestion:

It is recommended to deduct the forward fee (`fwd_fee`) and gas fee (`gas_fee`) during the minting process.

Resolution:

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

JMI-2 Refunding Excess TON During Minting

Severity: Major

Status: Acknowledged

Code Location:

contracts/jetton_minter.fc#117-136

Descriptions:

In the `jetton_minter` contract, the minting of Jetton tokens is calculated as follows:

```
(int, int, int) get_token_amount_for_ton(
    int msg_value,
    int virtual_total_supply,
    int virtual_ton_total,
    int virtual_initial_ton_amount,
    int target_liquidity,
    int max_supply
) {
    int k = const::nano() * const::nano() * virtual_initial_ton_amount;
    virtual_ton_total += (msg_value / 100) * 99;
    int tokens = (virtual_total_supply - (k / virtual_ton_total));
    int liquidity_pool = abs(virtual_total_supply - max_supply);

    if (liquidity_pool <= tokens) {
        tokens = liquidity_pool;
    }

    virtual_total_supply -= tokens;
    return (tokens, virtual_ton_total, virtual_total_supply);
}
```

If the value of `abs(virtual_total_supply - max_supply)` is zero or very small, the amount of Jetton tokens minted by the user will be minimal, despite the user spending a significant amount of TON.

Suggestion:

It is recommended to refund the excess TON to the user.

JMI-3 When `op = burn`, the Protocol's Calculation of `reserve_return` is Incorrect

Severity: Major

Status: Fixed

Code Location:

contracts/jetton_minter.fc#226-227

Descriptions:

When `op == op::mint()`, the protocol uses `get_token_amount_for_ton()` to calculate tokens.

```
(int, int, int) get_token_amount_for_ton(
    int msg_value,
    int virtual_total_supply,
    int virtual_ton_total,
    int virtual_initial_ton_amount,
    int target_liquidity,
    int max_supply
) {
    int k = const::nano() * const::nano() * virtual_initial_ton_amount;
    virtual_ton_total += (msg_value / 100) * 99;
    int tokens = (virtual_total_supply - (k / virtual_ton_total));
    int liquidity_pool = abs(virtual_total_supply - max_supply);

    if (liquidity_pool <= tokens) {
        tokens = liquidity_pool;
    }

    virtual_total_supply -= tokens;
    return (tokens, virtual_ton_total, virtual_total_supply);
}
```

When `op == op::burn_notification()`, the protocol first calculates `current_price`, then computes `reserve_return` using `jetton_amount * current_price`.

```
    if (op == op::burn_notification()) {
        int jetton_amount = in_msg_body~load_coins();
        slice from_address = in_msg_body~load_msg_addr();
```

```
    int current_price = get_jetton_price();
    int reserve_return = muldiv(jetton_amount, current_price, const::nano());
```

The calculation in `get_jetton_price()` is defined as `price = virtual_ton_total / virtual_token_total_supply`.

```
(int) get_jetton_price() method_id {
  (
    int total_supply,
    int max_supply,
    int target_liquidity,
    int virtual_token_total_supply,
    int virtual_token_max_supply,
    int virtual_ton_total,
    int virtual_initial_ton_amount,
    int reserve_balance,
    int reserve_ratio,
    slice admin_address,
    cell content,
    cell jetton_wallet_code,
    _) = load_data();

  return muldiv(virtual_ton_total, const::nano(), virtual_token_total_supply);
}
```

This method for calculating `reserve_return` differs from the approach used to compute tokens during minting, indicating a potential inconsistency in the token valuation process. This algorithm can allow users to mint and then directly burn to profit.

Suggestion:

It is recommended to ensure consistency between the two calculation methods.

Resolution:

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

JMI-4 Minting and Burning Lack Slippage Protection

Severity: Major

Status: Fixed

Code Location:

contracts/jetton_minter.fc#173-258

Descriptions:

The protocol is developed based on the x*y = k economic model, similar to Uniswap V2. Users receive LP tokens when they mint and can withdraw the corresponding tokens when they burn.

```
if (op == op::burn_notification()) {
    int jetton_amount = in_msg_body~load_coins();
    slice from_address = in_msg_body~load_msg_addr();
    int current_price = get_jetton_price();
    int reserve_return = muldiv(jetton_amount, current_price, const::nano());
    throw_unless(74,
      equal_slices(calculate_user_jetton_wallet_address(from_address, my_address(),
jetton_wallet_code), sender_address)
    );

    var msg = begin_cell()
      .store_uint(0x10, 6)
      .store_slice(from_address)
      .store_coins(reserve_return)
      .store_uint(0, 1 + 4 + 4 + 64 + 32 + 1 + 1);
    send_raw_message(msg.end_cell(), 0);

    slice response_address = in_msg_body~load_msg_addr();
    if (response_address.preload_uint(2) != 0) {
      var msg = begin_cell()
            .store_uint(0x10, 6) ;; nobounce - int_msg_info$0 ihr_disabled:Bool
bounce:Bool bounced:Bool src:MsgAddress -> 011000
            .store_slice(response_address)
            .store_coins(0)
            .store_uint(0, 1 + 4 + 4 + 64 + 32 + 1 + 1)
            .store_uint(op::excesses(), 32)
            .store_uint(query_id, 64);
```

```
        send_raw_message(msg.end_cell(), 2 + 64);
    }

    save_data(
        total_supply - jetton_amount,
        max_supply,
        target_liquidity,
        virtual_token_total_supply + jetton_amount,
        virtual_token_max_supply,
        virtual_ton_total - reserve_return,
        virtual_initial_ton_amount,
        reserve_balance - reserve_return,
        reserve_ratio,
        admin_address,
        content,
        jetton_wallet_code,
        jetton_minter_code);
    return ();
}
```

However, both minting and burning lack slippage protection, making users susceptible to sandwich attacks. In contrast, Uniswap V2 has implemented appropriate slippage protection measures within its protocol. https://github.com/Uniswap/v2-periphery/blob/master/contracts/UniswapV2Router02.sol#L46-L58

Suggestion:

It is recommended to implement slippage protection measures.

Resolution:

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

JMI-5 Centralization Risk

Severity: Major

Status: Acknowledged

Code Location:

contracts/jetton_minter.fc#339-410

Descriptions:

Centralization risk was identified in the smart contract. The admin can withdraw funds from the contract and modify its content.

```
    if (op == op::withdraw()) {
        throw_unless(73, equal_slices(sender_address, admin_address));
        slice address = in_msg_body~load_msg_addr();
        var msg = begin_cell()
          .store_uint(0x10, 6) ;; nobounce - int_msg_info$0 ihr_disabled:Bool bounce:Bool
bounced:Bool packages:MsgAddress -> 011000
          .store_slice(address)
          .store_coins(my_balance - const::min_tons_for_storage())
          .store_uint(0, 1 + 4 + 4 + 64 + 32 + 1 + 1);
        send_raw_message(msg.end_cell(), 0);
        return ();
    }
```

Suggestion:

It is recommended to take ways to reduce the risk of centralization.

JMI-6 Single-step Ownership Transfer Can be Dangerous

Severity: Medium

Status: Acknowledged

Code Location:

contracts/jetton_minter.fc#296-315

Descriptions:

Single-step ownership transfer means that if a wrong address was passed when transferring ownership or admin rights it can mean that role is lost forever. If the admin permissions are given to the wrong address within this function, it will cause irreparable damage to the contract. Below is the official documentation explanation from OpenZeppelin

https://docs.openzeppelin.com/contracts/4.x/api/access

Ownable is a simpler mechanism with a single owner "role" that can be assigned to a single account. This simpler mechanism can be useful for quick tests but projects with production concerns are likely to outgrow it.

```
if (op == 3) { ;; change admin
    throw_unless(73, equal_slices(sender_address, admin_address));
    slice new_admin_address = in_msg_body~load_msg_addr();
    save_data(
        total_supply,
        max_supply,
        target_liquidity,
        virtual_token_total_supply,
        virtual_token_max_supply,
        virtual_ton_total,
        virtual_initial_ton_amount,
        reserve_balance,
        reserve_ratio,
        new_admin_address,
        content,
        jetton_wallet_code,
        jetton_minter_code);
    return ();
}
```

Suggestion:

It is recommended to use a two-step ownership transfer pattern.

JMI-7 The Calculation of Tokens during Minting is Incorrect

Severity: Medium

Status: Fixed

Code Location:

contracts/jetton_minter.fc#117-136

Descriptions:

In the `get_token_amount_for_ton()` function, the protocol calculates tokens using the following method.

```
(int, int, int) get_token_amount_for_ton(
    int msg_value,
    int virtual_total_supply,
    int virtual_ton_total,
    int virtual_initial_ton_amount,
    int target_liquidity,
    int max_supply
) {
    int k = const::nano() * const::nano() * virtual_initial_ton_amount;
    virtual_ton_total += (msg_value / 100) * 99;
    int tokens = (virtual_total_supply - (k / virtual_ton_total));
    int liquidity_pool = abs(virtual_total_supply - max_supply);

    if (liquidity_pool <= tokens) {
        tokens = liquidity_pool;
    }

    virtual_total_supply -= tokens;
    return (tokens, virtual_ton_total, virtual_total_supply);
}
```

From the test code, we find the configuration information as follows.

```
const data: Cell = beginCell()
        .storeCoins(0) // total supply
        .storeCoins(toNano(800000000)) // max supply
        .storeCoins(toNano(1100)) // target liquidity
```

```
    .storeCoins(toNano(1000000000)) // virtual token total supply
    .storeCoins(toNano(800000000)) // virtual token max supply
    .storeCoins(toNano(400)) // virtual TON total
    .storeCoins(toNano(400)) // virtual initial TON amount
    .storeCoins(0) // reserve balance
    .storeUint(0, 32)
    .storeAddress(admin)
    .storeRef(contentCell)
    .storeRef(tokenCode)
    .endCell();
const dataCell: Cell = new Cell({ bits: data.bits, refs: data.refs });
```

Given `msg.value` as `100 * 1e9`, the calculations proceed as follows:

1. (k = 1e9 * 1e9 * 400 * 1e9 = 400000000000000000000000000000)
2. (virtual_ton_total = 400 * 1e9 + 100 * 1e9/100 * 99 = 499000000000)
3. The tokens are calculated as (tokens = 1000000000 * 1e9 - 400000000000000000000000000000/499000000000 = 198396793587174349).

When `buy_value` is also `100 * 1e9`, receiving `198396793587174349` tokens appears unreasonable.

Suggestion:

It is recommended to adjust the value of k.

Resolution:

This issue has been fixed by removing a `const::nano()` multiplier when calculating k.

JMI-8 Precision Loss

Severity: Minor

Status: Fixed

Code Location:

contracts/jetton_minter.fc#126

Descriptions:

The line `virtual_ton_total += (msg_value / 100) * 99;` may cause amplified precision loss due to performing division before multiplication.

Suggestion:

It is recommended to resolve this issue by multiplying first and then dividing, as follows:

```
virtual_ton_total += (msg_value * 99) / 100;
```

Resolution:

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

JMI-9 Storage in Jetton_minter Protocol

Severity: Informational

Status: Acknowledged

Code Location:

contracts/jetton_minter.fc#33-63

Descriptions:

In the Jetton_minter protocol, what is the significance of `target_liquidity`, `virtual_token_max_supply`, and `reserve_ratio`? The protocol does not include any evaluations or conditions regarding `target_liquidity` and other parameters.

Suggestion:

It is recommended to use or remove these parameters.

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.