#HashEx
BLOCKCHAIN SECURITY

Grishmans Kombat

smart contracts final audit report

October 2024

hashex.org

contact@hashex.org

Contents

1. Disclaimer 3

2. Overview 4

3. Project centralization risks 6

4. Found issues 7

5. Contracts 9

6. Conclusion 19

Appendix A. Issues severity classification 20

Appendix B. Issue status description 21

Appendix C. List of examined issue types 22

Appendix D. Centralization risks classification 23

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org). HashEx has exclusive rights to publish the results of this audit on company's web and social sites.

2. Overview

HashEx was commissioned by the Grishmans token team to perform an audit of their smart contracts. The audit was conducted between 16/09/2024 and 18/09/2024.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at @Angels120/claim_smartContract Github repository and was audited after the commit 5c90b29.

Update. The fixes were implemented in a another @Angels120/merkleTree_airdrop repository and were audited after the commit 8677a86. The scope of the recheck includes the following file: airdrop.fc. The code was deployed to the TON network at address EQAK9rRdUcEAoB3DOYKjzywk08u5p-mF0U9JCeCEkdzAFNBc.

2.1 Summary

Project name	Grishmans Kombat
URL	https://t.me/Grishmans_Kombat
Platform	TON
Language	FunC
Centralization level	High
Centralization risk	High

2.2 Contracts

Name	Address
jetton-vault	
airdrop	EQAK9rRdUcEAoB3DOYKjzywk08u5p-mF0U9JCeCEkdzAFNBc

3. Project centralization risks

The owner of the airdrop contract can:

- update the addresses and amounts for users to claim;
- withdraw all tokens from the airdrop contract;
- change the owner of the contract.

4. Found issues

Critical	1 (11%)
High	2 (22%)
Low	4 (44%)
Info	2 (23%)

C45. jetton-vault

ID	Severity	Title	Status
C45I48	Critical	No validation of claimed amounts	Resolved
C45I49	High	Lack of testing	Resolved
C45I47	High	Unauthorized Initialization of jetton_wallet_address	Resolved
C45I4b	Low	Unused variables	Resolved
C45I4c	Low	Unnecessary impure modifier	Resolved
C45I4d	Low	Not optimal calculations	Resolved

C5a. airdrop

ID	Severity	Title	Status
C5aI7c	Low	Gas optimizations	Resolved

C5aI7d	Info	Updating the Merkle tree leads to unclaimed airdrops invalidation	Acknowledged
C5aI7b	Info	Code optimizations	Resolved

5. Contracts

C45. jetton-vault

Overview

The Vault contract is designed for jetton token distribution. The expected flow is as follows: an admin transfers jetton tokens to the Vault contract, and users can subsequently claim tokens from it. The purpose of this contract is to ensure that users cover the costs associated with the token transfer themselves.

The contract allows the following operations:

1. Receiving Jettons: Accepts Jetton tokens and tracks the available balance.
2. Claiming Jettons: Users can claim a specified amount of Jettons from the Vault.
3. Owner Withdrawal: The owner can withdraw all available Jettons from the Vault.
4. Change Admin: The owner can transfer ownership of the Vault to another address.

Issues

C45I48 No validation of claimed amounts

Critical Resolved

The `claim()` function does not impose any limits or validations on the amount a user can claim, other than ensuring the Vault has sufficient Jettons.

```
    if (op == op::claim()) {
        ;; update available jettons state
        int claim_amount = in_msg_body~load_coins();
        storage::available_jettons -= claim_amount;
        ;; exit if not enough jettons
        throw_unless(404, storage::available_jettons >= 0);

        cell msg_body = begin_cell()
                            .store_uint(op::transfer(), 32)
```

```
                            .store_uint(0, 64)
                            ;; .store_coins(storage::user_claim_amount)
                            .store_coins(claim_amount)
                            .store_slice(sender_address)
                            .store_slice(sender_address)
                            .store_uint(0, 1)
                            .store_coins(1)
                            .store_uint(0, 1)
                        .end_cell();

        var msg = begin_cell()
                    .store_uint(0x10, 6)
                    .store_slice(storage::jetton_wallet_address)
                    .store_coins(0)
                    .store_uint(1, 1 + 4 + 4 + 64 + 32 + 1 + 1)
                    .store_ref(msg_body);

        send_raw_message(msg.end_cell(), 64);

        save_data();

        return ();
    }
```

Users can claim excessive amounts, potentially depleting the Vault's Jetton balance unintentionally or maliciously.

Proof of concept

```
    it('anyone can claim tokens from the vault', async function() {
        const intialAliceBalance = await (await
userWallet(alice.address)).getJettonBalance()
        expect(intialAliceBalance).toBe(0n)

        const amountToMint = toNano('100');
        await jettonMinter.sendMint(owner.getSender(), jettonVault.address, amountToMint,
toNano('0.01'), toNano('0.1'))

        const vaultWallet = await userWallet(jettonVault.address);

        const vaultWalletBalance = await vaultWallet.getJettonBalance();
```

```
        expect(vaultWalletBalance).toBe(amountToMint);

        await jettonVault.sendClaim(alice.getSender(), toNano('0.1'), amountToMint)

        const aliceWalletBalance = await (await
userWallet(alice.address)).getJettonBalance()
        expect(aliceWalletBalance).toBe(amountToMint)
    })
```

Recommendation

Generate a Merkle tree using the available claimable balances. Set the Merkle root in the contract. Verify that the claimed amount is correctly recorded in the Merkle tree.

C45I49 Lack of testing

High Resolved

The provided code lacks any accompanying tests. Additionally, the token interaction scripts in the repository are outdated and were written for a previous version of the vault, making them incompatible with the current version.

Testing is critical to ensure the reliability and security of smart contracts. Given the immutable nature of blockchain, once deployed, contracts cannot be easily modified, making it essential to identify and resolve bugs or vulnerabilities beforehand. Comprehensive testing helps verify contract functionality, ensures it behaves as expected under various conditions.

Recommendation

Implement unit tests to ensure comprehensive coverage of all functionalities. Additionally, test the deployed token on the testnet to verify its performance.

C45I47 Unauthorized Initialization of jetton_wallet_address

High Resolved

The Vault sets `jetton_wallet_address` upon receiving the first `transfer_notification` message without verifying the sender's authenticity.

```
() recv_internal(int my_balance, int msg_value, cell in_msg_full, slice in_msg_body)
impure {
  ...
      if (op == op::transfer_notification()) {
        if (storage::jetton_wallet_address.preload_uint(2) != 0) {  ;; "if jetton wallet
address is already saved"
            throw_unless(403, equal_slices(sender_address,
storage::jetton_wallet_address));
        }
        storage::jetton_wallet_address = sender_address;
        storage::available_jettons += in_msg_body~load_coins();
        save_data();

        return ();
    }
}
```

An attacker can send a fake `transfer_notification` to the Vault before the legitimate Jetton wallet does, effectively hijacking the `jetton_wallet_address`. If an attacker sends his transfer before legitimate owner's transfer, the tokens sent from the owner would be locked on the contract. Neither owner, nor users would not be able to withdraw them.

Proof of concept

```
    it('test frontrun with another token', async function() {
        const alternativeMinterCode = await compile('JettonMinter');
        const alternativeWalletCode = await compile('JettonWallet');

        const amountToMint = toNano('0.5');

        const minterConfig = {
            admin: bob.address,
            content: Cell.EMPTY,
            wallet_code: alternativeWalletCode
        }
        const altJettonMinter =
blockchain.openContract(JettonMinter.createFromConfig(minterConfig,
alternativeMinterCode));

        await altJettonMinter.sendMint(bob.getSender(), jettonVault.address, amountToMint,
```

```
toNano('0.01'), toNano('0.1'))

        const newAmountToMint = toNano('5000')
        const tx = await jettonMinter.sendMint(owner.getSender(), jettonVault.address,
newAmountToMint, toNano('0.01'), toNano('0.1'))

        const vaultWallet = await userWallet(jettonVault.address)

        expect(tx.transactions).toHaveTransaction({
            from: jettonMinter.address,
            to: vaultWallet.address,
            success: true
        })

        //vault reverts transfer notification
        expect(tx.transactions).toHaveTransaction({
            from: vaultWallet.address,
            to: jettonVault.address,
            success: false,
            exitCode: 403
        })

        //vault initialized with fake token balance
        const {available_jettons} = await jettonVault.getVaultData();
        expect(available_jettons).toBe(amountToMint)

        //vault gets tokens, but there is no way to withdraw them
        const balanceOfRealToken = await vaultWallet.getJettonBalance();
        expect(balanceOfRealToken).toBe(newAmountToMint)
    })
```

Recommendation

Create a setter for the `jetton_wallet_address`. Ensure that the setter can be only called by the owner and only once.

C45I4b Unused variables

Low Resolved

Several variables are declared, but never used in the contract.

```
    const int tons_for_gas = 150000000;
    int min_tons_for_storage() asm "50000000 PUSHINT"; ;; 0.05 TON
    ...
    int fwd_fee = muldiv(cs~load_coins(), 3, 2); ;; we use message fwd_fee for estimation
of forward_payload costs
    ...
    int query_id = in_msg_body~load_uint(64);
```

C45I4c Unnecessary impure modifier

Low Resolved

The function `load_data()` is declared impure but does not modify the state.

```
() load_data() impure inline {
    slice ds = get_data().begin_parse();

    storage::owner_address = ds~load_msg_addr();
    storage::available_jettons = ds~load_coins();
    storage::jetton_wallet_address = ds~load_msg_addr();
}
```

C45I4d Not optimal calculations

Low Resolved

The `jetton_wallet_address` variable could be set only once, but not on every transfer notification.

```
() recv_internal(int my_balance, int msg_value, cell in_msg_full, slice in_msg_body)
impure {
  ...
    if (op == op::transfer_notification()) {
        if (storage::jetton_wallet_address.preload_uint(2) != 0) {  ;; "if jetton wallet
address is already saved"
            throw_unless(403, equal_slices(sender_address,
storage::jetton_wallet_address));
        }
        storage::jetton_wallet_address = sender_address;
        ...
    }
```

```
...
}
```

C5a. airdrop

Overview

The contract utilizes a Merkle tree for the implementation of a Jetton token airdrop. The tree is composed of account addresses and the corresponding amounts they are to receive. The contract is designated as the owner of the Jettons and receives messages from the `airdrop_helper` contract, sending tokens to users based on the data set in the Merkle root.

Issues

C5aI7c Gas optimizations

Low Resolved

The function `recv_internal()` makes an unnecessary call to `save_data()` within the `op::owner_withdraw_jetton` handler.

```
() recv_internal(int my_balance, int msg_value, cell in_msg_full, slice in_msg_body)
impure {
  ...
    elseif (context::op == op::owner_withdraw_jetton) {
        throw_unless(error::not_owner, equal_slices(context::sender,
data::owner_address));

        int withdraw_amount = in_msg_body~load_coins();

        cell msg_body = begin_cell()
                        .store_uint(op::jetton::transfer, 32)
                        .store_uint(0, 64)
                        .store_coins(withdraw_amount)
                        .store_slice(data::owner_address)
                        .store_slice(data::owner_address)
                        .store_uint(0, 1)
                        .store_coins(1)
```

```
                            .store_uint(0, 1)
                        .end_cell();

        var msg = begin_cell()
                    .store_uint(0x18, 6)
                    .store_slice(data::jetton_wallet)
                    .store_coins(0)
                    .store_uint(1, 1 + 4 + 4 + 64 + 32 + 1 + 1)
                    .store_ref(msg_body);

        send_raw_message(msg.end_cell(), 64);

        save_data();
    }
    ...
}
```

C5aI7d Updating the Merkle tree leads to unclaimed airdrops invalidation

● Info ✓ Acknowledged

When the owner updates the Merkle root, all unclaimed airdrop amounts from the previous Merkle tree become invalid. This results in users who have not yet claimed their tokens being unable to do so.

C5aI7b Code optimizations

● Info ✓ Resolved

The function `send_tokens()` could be used in the `op::owner_withdraw_jetton handler`. This will eliminate duplication of the code as the function and the handler use almost the same code.

```
() send_tokens(slice recipient, int amount) impure {
    send_raw_message(begin_cell()
        .store_uint(0x18, 6)
        .store_slice(data::jetton_wallet)
        .store_coins(0)
        .store_uint(1, 107)
        .store_ref(begin_cell()
```

```
            .store_uint(op::jetton::transfer, 32)
            .store_uint(context::query_id, 64)
            .store_coins(amount)
            .store_slice(recipient)
            .store_slice(recipient)
            .store_uint(0, 1)
            .store_coins(10000000)
            .store_uint(0, 1)
        .end_cell())
    .end_cell(), 64);
}


() recv_internal(int my_balance, int msg_value, cell in_msg_full, slice in_msg_body)
impure {
  ...
    elseif (context::op == op::owner_withdraw_jetton) {
        throw_unless(error::not_owner, equal_slices(context::sender,
data::owner_address));

        int withdraw_amount = in_msg_body~load_coins();

        cell msg_body = begin_cell()
                        .store_uint(op::jetton::transfer, 32)
                        .store_uint(0, 64)
                        .store_coins(withdraw_amount)
                        .store_slice(data::owner_address)
                        .store_slice(data::owner_address)
                        .store_uint(0, 1)
                        .store_coins(1)
                        .store_uint(0, 1)
                    .end_cell();

        var msg = begin_cell()
                    .store_uint(0x18, 6)
                    .store_slice(data::jetton_wallet)
                    .store_coins(0)
                    .store_uint(1, 1 + 4 + 4 + 64 + 32 + 1 + 1)
                    .store_ref(msg_body);

        send_raw_message(msg.end_cell(), 64);

        save_data();
```

```
    }
    ...
}
```

6. Conclusion

1 critical, 2 high, 4 low severity issues were found during the audit. 1 critical, 2 high, 4 low issues were resolved in the update. The reviewed contracts are highly dependent on the owner's account. See the centralization risks chapter.

Appendix A. Issues severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Info.** Issues that do not impact the contract operation. Usually, info severity issues are related to code best practices, e.g. style guide.

Appendix B. Issue status description

Resolved. The issue has been completely fixed.

Partially fixed. Parts of the issue have been fixed but the issue is not completely resolved.

Acknowledged. The team has been notified of the issue, no action has been taken.

Open. The issue remains unresolved.

Appendix C. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

Appendix D. Centralization risks classification

Centralization level

- **High.** The project owners can manipulate user's funds, lock user's funds on their will (reversible or irreversible), or maliciously update contracts parameters or bytecode.
- **Medium.** The project owners can modify contract's parameters to break some functions of the project contract or contracts, but user's funds remain withdrawable.
- **Low.** The contract is trustless or its governance functions are safe against a malicious owner.

Centralization risk

- **High.** Lost ownership over the project contract or contracts may result in user's losses. Contract's ownership belongs to EOA or EOAs, and their security model is unknown or out of scope.
- **Medium.** Contract's ownership is transferred to a contract with not industry-accepted parameters, or to a contract without an audit. Also includes EOA with a documented security model, which is out of scope.
- **Low.** Contract's ownership is transferred to a well-known or audited contract with industry-accepted parameters.

 contact@hashex.org

 @hashex_manager

 blog.hashex.org

 linkedin

 github

 twitter

#HashEx
BLOCKCHAIN SECURITY