Security Audit Report for Meme Launchpad

Date: August 15, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Shitzu Apes
Target	Meme Launchpad

Version History

Version	Date	Description
1.0	August 15, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-08-16 02:07:00+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Rust
Approach	Semi-automatic and manual verification

The target of this audit is the code repository of Meme Launchpad[1] of Shitzu Apes. Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include `meme-launchpad` folder contract only. Specifically, the files covered in this audit include:

```
crates/meme-token/src/lib.rs
```

Listing 1.1: Audit Scope for this Report

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
Meme Launchpad	Version 1	0b4ead45a93ddad80a65fe7e85dfaf81215a7fd3
	Version 2	be0fb1c94a6a6203bb5a37f34f8ec8c75b21343f

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

[1] https://github.com/Shitzu-Apes/meme-launchpad/tree/main/crates/meme-token

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **one** potential security issue. Besides, we have **one** recommendation and **two** notes.

- Medium Risk: 1
- Recommendation: 1
- Note: 2

ID	Severity	Description	Category	Status
1	Medium	Lack of state rollback for the failure of cross contract invocation	DeFi Security	Fixed
2	-	Lack of check in function `new()`	Recommendation	Confirmed
3	-	Potential centralization risk	Note	-
4	-	Ensure `token_id` is registered in `ref_contract` before deploying token	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Lack of state rollback for the failure of cross contract invocation

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `lib.rs` file, the function `new()` is used for initiating and configuring a token. In this process, the function will first directly mint the corresponding amount of tokens to the `deployer` and the `ref_contract`, and then invoke a cross-contract call to function `ft_on_transfer()` in the `ref_contract` to account for the deployer. However, if the cross-contract call fails (e.g., the `ref_contract` is paused), the minted tokens will still be retained in the `ref_contract` but will not be recorded in the `deployer`'s account. This leads to the assets being unable to be retrieved.

```
pub fn new(
    name: String,
    symbol: String,
    icon: String,
    decimals: u8,
    total_supply: U128,
    ref_contract_id: AccountId,
    pool_amount: U128,
) -> Self {
    let deployer = env::predecessor_account_id();
    let mut this = Self {
        name,
        symbol,
```

```
            icon,
            decimals,
            token: FungibleToken::new(b"t".to_vec()),
        };


        this.token.internal_register_account(&deployer);
        this.token
            .internal_deposit(&deployer, total_supply.0 - pool_amount.0);
        FtMint {
            owner_id: &deployer,
            amount: U128(total_supply.0 - pool_amount.0),
            memo: None,
        }
        .emit();


        this.token.internal_register_account(&ref_contract_id);
        this.token.internal_deposit(&ref_contract_id, pool_amount.0);
        FtMint {
            owner_id: &ref_contract_id,
            amount: U128(pool_amount.0),
            memo: None,
        }
        .emit();


        // this simulates an 'ft_transfer_call' to Ref contract for a token deposit
        ext_ft_receiver::ext(ref_contract_id.clone())
            .with_unused_gas_weight(1)
            .ft_on_transfer(deployer, pool_amount, "".to_string());


        this
    }
```

Listing 2.1: lib.rs

Impact Assets deposited into the `ref_contract` cannot be retrieved.

Suggestion Revise the logic to ensure that the state is correctly rolled back when a cross-contract invoke fails.

2.2 Additional Recommendation

2.2.1 Lack of check in function `new()`

Status Confirmed

Introduced by `Version 1`

Description During the initialization process, the contract directly mints all of the `total_supply` to the `deployer`, except for the amount to be transferred to the `ref_contract`. There is a lack of validation for the size of the two parameters here. Specifically, if the `pool_amount` is greater than the `total_supply`, an underflow error will occur.

```rust
pub fn new(
    name: String,
    symbol: String,
    icon: String,
    decimals: u8,
    total_supply: U128,
    ref_contract_id: AccountId,
    pool_amount: U128,
) -> Self {
    let deployer = env::predecessor_account_id();
    let mut this = Self {
        name,
        symbol,
        icon,
        decimals,
        token: FungibleToken::new(b"t".to_vec()),
    };


    this.token.internal_register_account(&deployer);
    this.token
        .internal_deposit(&deployer, total_supply.0 - pool_amount.0);
    FtMint {
        owner_id: &deployer,
        amount: U128(total_supply.0 - pool_amount.0),
        memo: None,
    }
    .emit();


    this.token.internal_register_account(&ref_contract_id);
    this.token.internal_deposit(&ref_contract_id, pool_amount.0);
    FtMint {
        owner_id: &ref_contract_id,
        amount: U128(pool_amount.0),
        memo: None,
    }
    .emit();


    // this simulates an ‘ft_transfer_call‘ to Ref contract for a token deposit
    ext_ft_receiver::ext(ref_contract_id.clone())
        .with_unused_gas_weight(1)
        .ft_on_transfer(deployer, pool_amount, "".to_string());


    this
}
```

Listing 2.2: lib.rs

Suggestion Add a check to ensure `pool_amount` is less than the `total_supply`.

Feedback from the project Yes this is true. However the `pool_amount` will always be less than the `total_supply` according to the cross contract function call by the token factory.

2.3 Notes

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description The contract has a centralization risk, at the time of deployment, the `deployer` owns the entire `total_supply`. Additionally, the deployer holds the account's full access key. If the `deployer`'s private key is lost or maliciously used, they could upgrade the contract, causing losses to users.

Feedback from the project No access key is created, since it's deployed by a factory contract. No access key exists ever for the token contracts. And this is working as intended. The factory contract (which was not part of this audit) will be the one to deploy these token contracts. It will keep track of all accounts which can claim their respective token share. It is not possible to directly send all tokens to the respective users, since it would exceed the max allowed amount of gas per transaction.

2.3.2 Ensure `token_id` is registered in `ref_contract` before deploying token

Introduced by `Version 1`

Description During the token initialization process, a portion of the assets is deposited into the `ref_contract`, and the function `ft_on_transfer()` within the `ref_contract` records who deposited these assets. The function `ft_on_transfer()` checks if the token has already been registered in the `ref_contract`. Therefore, it is crucial to ensure that the function `register_tokens()` of the `ref_contract` has been invoked before creating the token. Otherwise, `ft_on_transfer()` will not execute properly.

Feedback from the project Tokens are created through a factory contract. During the entire creation process, the function `register_tokens()` of the `ref_contract` is invoked first, followed by the creation of the token contract.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS