Security Audit Report for EasyCoin

Date: October 8, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	EasyCoin.AI
Target	EasyCoin

Version History

Version	Date	Description
1.0	October 8, 2024	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-10-09 02:19:13+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 audit focuses on EasyCoin by EasyCoin.AI [1], which enables users to follow the trading strategies of others in a secure, non-custodial manner. The audit is limited to files within the `programs/easycoin` folder of the repository, excluding all other files. External dependencies, such as the Solana development framework `Anchor` [2], are assumed to be reliable in terms of functionality and security, and are not within the scope of this audit.

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
EasyCoin	Version 1	b30bdf6775dcaae076805eee4627adca5fe8f994
	Version 2	9ea4a6d3562a4b3d7ac345bc248f22c401ff4e8e

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] The audit for EasyCoin was conducted in a private, undisclosed repository. Once all issues and recommendations were resolved, the project maintainers created a new repository at https://github.com/RealEasyCoin/easycoin-program/ containing the source code for `Version 2`.

[2] https://www.anchor-lang.com/

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 [3] and Common Weakness Enumeration [4]. 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.

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

[4] 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 **one** note.

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

ID	Severity	Description	Category	Status
1	Medium	Potentially unclaimable user fee	DeFi Security	Fixed
2	-	Add explicit checks to verify swap output	Recommendation	Fixed
3	-	Potential centralization risks	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potentially unclaimable user fee

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `EasyCoin` contract is designed to charge fees for each user swap after the swap is completed on `Jupiter`. However, as shown in the following code segment, the contract only checks if the remaining balance (after deducting the required rent) exceeds *the fee for the current swap*, rather than *the accumulated fees for the user's account*. This could result in the contract undercharging users.

```
let required_rent: u64 = Rent::get()?.minimum_balance(user_account.data_len());
let user_account_balance = user_account.lamports() - required_rent;
require!(
    user_account_balance >= trade_fee,
    AgentError::UserAccountBalanceNotEnough
);

// record swap fee
ctx.accounts
    .owner_account
    .add_user_account_due_fee(args.user_account_nonce, trade_fee)?;

emit!(SwapOnJupiterEvent {
    user_account: ctx.accounts.user_account.key(),
    result: true,
});
```

Listing 2.1: src/instructions/swap_on_jupiter.rs

Impact User accounts may have insufficient balances to cover the accumulated fees for all swaps.

Suggestion Add checks to ensure that the accumulated fees can be properly charged.

2.2 Additional Recommendation

2.2.1 Add explicit checks to verify swap output

Status Fixed in `Version 2`

Introduced by `Version 1`

Description When the operator swaps token on behalf of a user, the destination account for the call to Jupiter can be set to `Jupiter::id()`. By default, a Jupiter swap transfers the output tokens to the user account when this destination account is assigned. However, to accommodate potential logic upgrades, the program should implement explicit checks to ensure that the output tokens are transferred to the user account.

```rust
fn validate_destination_token_account(
    user_account: SystemAccount,
    route_type: JupiterRouteType,
    remaining_accounts: &'info [AccountInfo<'info>],
) -> Result<InterfaceAccount<'info, TokenAccount>> {
    match route_type {
        JupiterRouteType::Route => {
            let user_destination_token_account_info = remaining_accounts
                .get(3)
                .ok_or(AgentError::NotJupiterRoute)?;
            #[cfg(feature = "enable-log")]
            msg!(
                "user_destination_token_account_info: {}",
                user_destination_token_account_info.key().to_string()
            );

            let user_destination_token_account: InterfaceAccount<'info, TokenAccount> =
                InterfaceAccount::try_from(user_destination_token_account_info)?;
            require!(
                user_destination_token_account.owner == user_account.key(),
                AgentError::JupiterRouteDestinationInvalid
            );

            #[cfg(feature = "enable-log")]
            msg!("destination token account authority is valid");

            let destination_token_account_info = remaining_accounts
                .get(4)
                .ok_or(AgentError::NotJupiterRoute)?;
            // if equal to Jupiter::id(), destination_token_account is user_destination_token_account
            require!(
                destination_token_account_info.key() == Jupiter::id()
                    || destination_token_account_info.key()
```

```
                        == user_destination_token_account.key(),
                    AgentError::JupiterRouteDestinationInvalid
                );
                Ok(user_destination_token_account)
            }
            JupiterRouteType::SharedAccountRoute => {
                return err!(AgentError::NotJupiterRoute);
            }
        }
    }
```

Listing 2.2: src/instructions/swap_on_jupiter.rs

Impact N/A

Suggestion Add checks to the swap result after executing the swap on Jupiter.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description The program operator has privileged access to modify system configurations and swap tokens on behalf of users, which poses centralization risks. If the privileged account were compromised, it could disrupt the program's entire functionality. However, the operator cannot directly steal funds from users, which mitigates this risk.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS