Jupiter Limit Order V2

Smart Contract Security Assessment

April 2024

Prepared for:

Jupiter

Prepared by:

Offside Labs

Ronny Xing

Haijiang Xie

Zhipeng Xu

Contents

1 About Offside Labs

Offside Labs is a leading security research team, composed of top talented hackers from both academia and industry.

We possess a wide range of expertise in modern software systems, including, but not limited to, *browsers*, *operating systems*, *IoT devices*, and *hypervisors*. We are also at the forefront of innovative areas like *cryptocurrencies* and *blockchain technologies*. Among our notable accomplishments are remote jailbreaks of devices such as the **iPhone** and **PlayStation 4**, and addressing critical vulnerabilities in the **Tron Network**.

Our team actively engages with and contributes to the security community. Having won and also co-organized *DEFCON CTF*, the most famous CTF competition in the Web2 era, we also triumphed in the **Paradigm CTF 2023** within the Web3 space. In addition, our efforts in responsibly disclosing numerous vulnerabilities to leading tech companies, such as *Apple*, *Google*, and *Microsoft*, have protected digital assets valued at over **$300 million**.

In the transition towards Web3, Offside Labs has achieved remarkable success. We have earned over **$9 million** in bug bounties, and **three** of our innovative techniques were recognized among the **top 10 blockchain hacking techniques of 2022** by the Web3 security community.

https://offside.io/

https://github.com/offsidelabs

https://twitter.com/offside_labs

2 Executive Summary

Introduction

Offside Labs completed a security audit of *Jupiter Limit Order V2* project, starting on April 3rd, 2024, and concluding on April 3rd, 2024.

Project Overview

Jupiter Limit Order V2: Jupiter Limit Order provides users with the simplest way to place limit orders on Solana and receive tokens directly in the users' self-custody wallets when the order is filled. This V2 is its second brand-new version.

Audit Scope

The assessment scope contains mainly the smart contracts of the *limit-order-2* program and *keeper client* for the *Jupiter Limit Order V2* project.

The audit is based on the following specific branches and commit hashes of the codebase repositories:

- Jupiter Limit Order V2
  - Branch: main
  - Commit Hash: 34654f001af0b07b9b25ab8ea175a2a50eba2e91
  - Codebase Link

We listed the files we have audited below:

- Jupiter Limit Order V2
  - programs/limit-order-2/src/*.rs
  - keeper/src/*.ts

Findings

The security audit revealed:

- 0 critical issue
- 0 high issues
- 0 medium issues
- 2 low issues
- 5 informational issues

Further details, including the nature of these issues and recommendations for their remediation, are detailed in the subsequent sections of this report.

3 Summary of Findings

ID	Title	Severity	Status
01	Referral Token Accounts May Not Be Claimable	Low	Acknowledged
02	Keeper Does Not Use the Referral Token Accounts Correctly as the Fee Recipient	Low	Fixed
03	Maker Account Type Validation Is Inconsistent	Informational	Fixed
04	expired_at Check Conditions Are Inconsistent	Informational	Fixed
05	update_fee Ix Does Not Check the Fee Cap	Informational	Fixed
06	fee_authority Can Be Loaded From Ctx Directly	Informational	Fixed
07	flash_fill_order Instruction Does Not Check the output_mint	Informational	Fixed

4 Key Findings and Recommendations

4.1 Referral Token Accounts May Not Be Claimable

Severity: Low

Status: Acknowledged

Target: Smart Contract

Category: Data Validation

Description

If initializing an order with a referral, all fees will be sent to the referral token account.

The issue is that the `initialize_order` instruction's referral account only checks for `token::authority = REFERRAL_AUTHORITY` and the `output_mint`. This could allow a malicious user to input an un-claimable referral account.

Impact

This is a griefing attack, which will result in the admin being unable to withdraw the protocol fees.

Proof of Concept

We can find that the `referral_token_account` is a specific PDA account in:

```
    #[account(
        mut,
        seeds = [REFERRAL_ATA_SEED, referral_account.key().as_ref(), mint.key().as_ref()],
        bump,
        token::mint = mint,
        token::authority = project
    )]
    referral_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
```

programs/referral/src/instructions/claim.rs#L101-L108

A malicious user can initialize any other token account for `REFERRAL_AUTHORITY`.

Recommendation

Input the `ReferralAccount` of the referral program to check if the `referral_token_account` is claimable.

Mitigation Review Log

Jupiter Team: Acknowledged. Only partner will input this token account. There is no reason why they wan to input an un-claimable referral account, like no benefits for them at all.

Offside Labs: It's a griefing attack without profit. Even if users do not introduce a partner, there is still a minimum fee rate requirement. Therefore, users (attackers) passing in this token account has no impact on them, but it can cause damage to the protocol's revenue (fee transferred to an unclaimable address). I think we can reserve such a plan, so that if this issue really occurs with non-dust losses, we can directly upgrade the Referral program to retrieve these stuck fees.

4.2 Keeper Does Not Use The Referral Token Accounts Correctly as the Fee Recipient

Severity: Low	**Status: Fixed**
Target: Keeper Client	**Category: Logic**

Description

The keeper `flashFillOrder` function uses the following code to create ATA of the fee account by the `CreateMode::Idempotent` .

```
    preInstructions.push(
      createAssociatedTokenAccountIdempotentInstruction(
        taker,
        order.feeAccount,
        FEE_AUTHORITY,
        order.outputMint,
        order.outputTokenProgram
      )
    );
```

keeper/src/fillOrder.ts#L116-L124

But the `CreateAssociatedTokenAccount` instruction will still check if the owner of the ATA is the `FEE_AUTHORITY` , even if that ATA already exists.

Impact

If the order's `fee_account` originates from a referral, then this instruction will fail, causing the entire fill order transaction to consistently fail.

Proof of Concept

```
        if associated_token_account.base.owner != *wallet_account_info.key {
            let error = AssociatedTokenAccountError::InvalidOwner;
            msg!("{}", error);
            return Err(error.into());
        }
```

solana-labs/solana-program-library/associated-token-account/program/src/processor.rs#L97-L101

Recommendation

The owner of the `fee_account` could be either `FEE_AUTHORITY` or `REFERRAL_AUTHORITY` .

Mitigation Review Log

Jupiter Team: Commit c554bdf8bea9179c2f6d540d655daa39582a6b88

Offside Labs: Fixed.

4.3 Informational and Undetermined Issues

Maker Account Type Validation Is Inconsistent

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Logic

It uses `SystemAccount` to check the maker account in the `flash_fill_order` ix.

```
maker: SystemAccount<'info>,
```

programs/limit-order-2/src/instructions/flash_fill_order.rs#L98

But it uses `maker: UncheckedAccount<'info>` to bypass the case where the account is a PDA in the `cancel_order` ix.

Jupiter Team: Commit d776ef5c747f39303101f69114fdf69f5455ffb9

Offside Labs: Fixed.

expired_at Check Conditions Are Inconsistent

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Logic

```rust
    pub fn validate_pre_flash_fill(&self, making_amount: u64) -> Result<()> {
        ...
            require!(expired_at > now, LimitOrderError::OrderExpired);
        }
```

programs/limit-order-2/src/state.rs#L84

This `expired_at` in the `validate_pre_flash_fill` function should be `>=` instead of `>`, due to the inconsistency with the `now > self.expired_at` check in the `validate_cancel_order` function.

Jupiter Team: Commit 5af58d09174ac499c4156bde6f8160c73cd5a76f

Offside Labs: Fixed.

update_fee Ix Does Not Check the Fee Cap

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Code QA

```rust
    ctx.accounts.fee_authority.set_inner(Fee {
```

programs/limit-order-2/src/instructions/update_fee.rs#L9

It's better to add a fee cap check to restrict excessively high unreasonable config.

Jupiter Team: Commit dd06f878454bcf3dc76c8e3cff6fc471249f1eb1

Offside Labs: Fixed.

fee_authority Can Be Loaded From Ctx Directly

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Code QA

```rust
let (fee_authority, _) = Pubkey::find_program_address(&[FEE_SEED], &crate::ID);
```

programs/limit-order-2/src/instructions/initialize_order.rs#L52

`fee_authority` is also the `fee: Box<Account<'info, Fee>>` of the `InitializeOrder` (the current `ctx.accounts`).

Jupiter Team: Commit 6cdd864988fad3d4677b3c181fafd3b983081e36

Offside Labs: Fixed.

flash_fill_order Instruction Does Not Check the output_mint

Severity: Informational	Status: Fixed
Target: Smart Contract	Category: Data Validation

```
    output_mint: Box<InterfaceAccount<'info, Mint>>,
```

programs/limit-order-2/src/instructions/flash_fill_order.rs#L130

The `flash_fill_order` instruction does not sufficiently validate `output_mint` . Although `maker_output_mint_account` and `taker_output_mint_account` do check `output_mint` , both of these token accounts are provided by the taker and are not included in the `order` .

And if the `output_mint` is `spl_token::native_mint::ID` , the `order.fee_account` also does not validate `output_mint` .

However, this issue is NOT exploitable because the `Order::transfer_from_taker` will call `sync_native` on the `fee_account` , and if `fee_account` 's `is_native` is false, it will fail directly.

To ensure safety in future codes, its a good idea to add constraint to make sure the `output_mint` is equal to `order.output_mint` .

Jupiter Team: Commit eafaaf8ee725b320b1540622e9bcaf7ad00aea89

Offside Labs: Fixed.

5 Disclaimer

This audit report is provided for informational purposes only and is not intended to be used as investment advice. While we strive to thoroughly review and analyze the smart contracts in question, we must clarify that our services do not encompass an exhaustive security examination. Our audit aims to identify potential security vulnerabilities to the best of our ability, but it does not serve as a guarantee that the smart contracts are completely free from security risks.

We expressly disclaim any liability for any losses or damages arising from the use of this report or from any security breaches that may occur in the future. We also recommend that our clients engage in multiple independent audits and establish a public bug bounty program as additional measures to bolster the security of their smart contracts.

It is important to note that the scope of our audit is limited to the areas outlined within our engagement and does not include every possible risk or vulnerability. Continuous security practices, including regular audits and monitoring, are essential for maintaining the security of smart contracts over time.

Please note: we are not liable for any security issues stemming from developer errors or misconfigurations at the time of contract deployment; we do not assume responsibility for any centralized governance risks within the project; we are not accountable for any impact on the project's security or availability due to significant damage to the underlying blockchain infrastructure.

By using this report, the client acknowledges the inherent limitations of the audit process and agrees that our firm shall not be held liable for any incidents that may occur subsequent to our engagement.

This report is considered null and void if the report (or any portion thereof) is altered in any manner.