Security Assessment Report

Kamino Limit Orders

January 29, 2025

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the Kamino Limit Orders smart contracts.

The artifact of the audit was the source code of the following programs, excluding tests, in a private repository.

The initial audit focused on the following versions and revealed 5 issues or questions.

program	type	commit
Kamino Limit Orders	Solana	`01348b5fee5b5bf1870ab16e5f8055dd65400ae9`

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
KAMINO LIMIT ORDERS		
[H-01] Potential DoS in "take_order" with wSOL-SOL conversion	High	Resolved
[M-01] Potential token discrepancy due to Token-2022 transfer fee extension	Medium	Resolved
[L-01] "verify_ata" function lacks Token-2022 compatibility	Low	Resolved
[L-02] Program whitelist can be bypassed	Low	Resolved
[I-01] Missing order type validation	Info	Resolved

Findings in Detail

KAMINO LIMIT ORDERS

[H-01] Potential DoS in "take_order" with wSOL-SOL conversion

In the current implementation of the "take_order" instruction, a feature is provided to simplify the process for users when the output mint is wSOL. This feature automatically converts wSOL to SOL and transfers it to the order maker. The implementation involves creating an "intermediary_output_token_account", transferring wSOL from the order taker to this account, and then closing the token account to convert the wSOL into SOL before transferring it to the order maker.

```
/* programs/limo/src/handlers/take_order.rs */
214 | let output_is_wsol = is_wsol(&ctx.accounts.output_mint.key());
215 | let output_destination_token_account = if output_is_wsol {
216 |     let intermediary_output_token_account = ctx
217 |         .accounts
218 |         .intermediary_output_token_account
219 |         .as_ref()
220 |         .ok_or(LimoError::IntermediaryOutputTokenAccountRequired)?;
221 |     let order_key = ctx.accounts.order.key();
222 |     let token_account_signer_seeds: &[&[u8]] =
223 |         intermediary_seeds!(ctx.bumps.intermediary_output_token_account, &order_key);
224 |     // Initialize intermediary output ATA
225 |     initialize_token_account_with_signer_seeds(
226 |         intermediary_output_token_account.to_account_info().clone(),
227 |         ctx.accounts.output_mint.to_account_info(),
228 |         ctx.accounts.output_token_program.to_account_info(),
229 |         ctx.accounts.pda_authority.to_account_info(),
230 |         ctx.accounts.rent.to_account_info(),
231 |         token_account_signer_seeds,
232 |         seeds,
233 |     )?;
234 |
235 |     intermediary_output_token_account.to_account_info()
236 | } else {
```

However, the creation of the "intermediary_output_token_account" is implemented using "system_instruction::create_account". This method fails if the target address already has a non-zero balance, as the instruction treats the address as already in use. As a result, a malicious actor could pre-calculate the address of the "intermediary_output_token_account" for an order with wSOL as the output token and send a small amount of lamports to the account in advance, caus-

ing a denial-of-service (DoS) attack.

```
/* programs/limo/src/token_operations.rs */
166 | let rent_exempt_balance = Rent::get()?.minimum_balance(TokenAccount::LEN);
167 |
168 | let create_account_ix = system_instruction::create_account(
169 |     authority.key,            // Payer of the account's rent
170 |     token_account.key,        // New WSOL token account address
171 |     rent_exempt_balance,      // Minimum rent-exempt balance
172 |     TokenAccount::LEN as u64, // Space needed for a token account
173 |     &spl_token::ID,           // Token program ID
174 | );
```

It is recommended to adopt the account creation mechanism used by Anchor for creating the "`intermediary_output_token_account`". Additionally, the current implementation does not account for Token-2022 when determining the length of the "`intermediary_output_token_account`". This should also be addressed to ensure compatibility.

Resolution

Fixed by commit "`21c397e`".

KAMINO LIMIT ORDERS

[M-01] Potential token discrepancy due to Token-2022 transfer fee extension

When creating an order, the current implementation records the amount and mint of the tokens provided by the user (input token) and the tokens they expect to receive (output token) in the order account. It then transfers the specified amount of input tokens to a vault owned by the PDA authority. However, this implementation does not account for the potential impact of the Token-2022 transfer fee extension. If the token provided by the order creator (order maker) has the transfer fee extension enabled and the fee rate is non-zero, the actual amount of input tokens received by the vault may be less than the amount recorded in the order account.

```
/* programs/limo/src/operations.rs */
053 | order.initial_input_amount = input_amount;
054 | order.remaining_input_amount = input_amount;
055 | order.expected_output_amount = output_amount;

/* programs/limo/src/handlers/create_order.rs */
035 | transfer_from_user_to_token_account(
036 |     ctx.accounts.maker_ata.to_account_info(),
037 |     ctx.accounts.input_vault.to_account_info(),
038 |     ctx.accounts.maker.to_account_info(),
039 |     ctx.accounts.input_mint.to_account_info(),
040 |     ctx.accounts.input_token_program.to_account_info(),
041 |     input_amount,
042 |     ctx.accounts.input_mint.decimals,
043 | )?;
```

This discrepancy can lead to issues when the order taker performs a take-order operation or when the order maker cancels the order. In these cases, the amount of tokens transferred out of the vault is determined by the amount recorded in the order account. As a result, the vault may end up with an insufficient token balance, preventing the successful completion of some requests.

Although the vault is restricted by the PDA to be created only by the admin, and the admin can manually avoid using mints with the transfer fee extension enabled, it is recommended to implement additional checks in the code to mitigate potential errors caused by admin oversight.

Resolution

Fixed by commit "e583af9".

KAMINO LIMIT ORDERS

[L-01] "verify_ata" function lacks Token-2022 compatibility

The program uses "`verify_ata`" to validate the ATA, but this verification only allows the SPL Token program's ATA.

```
/* programs/limo/src/utils/constraints.rs */
065 | pub fn verify_ata(wallet: &Pubkey, mint: &Pubkey, ata_account_key: &Pubkey) -> Result<()> {
066 |     // Derive the expected ATA address
067 |     let expected_ata = get_associated_token_address(wallet, mint);
068 |
069 |     // Verify the ATA's address
070 |     require_keys_eq!(
071 |         ata_account_key.key(),
072 |         expected_ata,
073 |         LimoError::InvalidAtaAddress
074 |     );
075 |
076 |     Ok(())
077 | }
```

It is recommended to use "`get_associated_token_address_with_program_id`" for the ATA verification to support Token-2022.

Resolution

Fixed by commit "`21c397e`".

KAMINO LIMIT ORDERS

[L-02] Program whitelist can be bypassed

The current implementation of the flash take order feature ensures that the program IDs for instructions before the start ix and after the end ix are included in the whitelist. However, due to the existence of the Token-2022 transfer hook, this design does not effectively guarantee that these instructions will not perform unintended operations.

```
/* programs/limo/src/utils/flash_ixs.rs */
140 | /// Instructions before or after the flash ixs are allowed only if they are part of following
↪  programs
141 | fn program_id_allowed(program_id: Pubkey) -> bool {
142 |     program_id == COMPUTE_BUDGET_PUBKEY
143 |         || program_id == spl_token::ID
144 |         || program_id == token_2022::ID
145 |         || program_id == associated_token::ID
146 | }
```

Resolution

Fixed by commits "`f80f2be`" and "`77dceaf`".

KAMINO LIMIT ORDERS

[I-01] Missing order type validation

In the "`create_order`" instruction, the current implementation allows users to provide an "`order_type`" parameter of type "`u8`". This parameter is directly written into the corresponding field of the newly created order account without validating whether the provided value represents a legitimate order type. Although the absence of such validation does not pose a security risk within the program, it could lead to issues in frontend or other related code.

```
/* programs/limo/src/handlers/create_order.rs */
011 | pub fn handler_create_order(
012 |     ctx: Context<CreateOrder>,
013 |     input_amount: u64,
014 |     output_amount: u64,
015 |     order_type: u8,
016 | ) -> Result<()> {
017 |     let order = &mut ctx.accounts.order.load_init()?;
018 |     let clock = Clock::get()?;
019 |
020 |     operations::create_order(
021 |         order,
022 |         ctx.accounts.global_config.key(),
023 |         ctx.accounts.maker.key(),
024 |         input_amount,
025 |         output_amount,
026 |         ctx.accounts.input_mint.key(),
027 |         ctx.accounts.output_mint.key(),
028 |         ctx.accounts.input_token_program.key(),
029 |         ctx.accounts.output_token_program.key(),
030 |         order_type,
031 |         ctx.bumps.input_vault,
032 |         clock.unix_timestamp,
033 |     )?;

/* programs/limo/src/operations.rs */
035 | pub fn create_order(
036 |     order: &mut Order,
037 |     global_config: Pubkey,
038 |     owner: Pubkey,
039 |     input_amount: u64,
040 |     output_amount: u64,
041 |     input_mint: Pubkey,
042 |     output_mint: Pubkey,
043 |     input_mint_program_id: Pubkey,
044 |     output_mint_program_id: Pubkey,
045 |     order_type: u8,
046 |     in_vault_bump: u8,
047 |     current_timestamp: i64,
048 | ) -> Result<()> {
```

```
049 |     require!(input_amount > 0, LimoError::OrderInputAmountInvalid);
050 |     require!(output_amount > 0, LimoError::OrderOutputAmountInvalid);
051 |     require!(input_mint != output_mint, LimoError::OrderSameMint);
052 |     order.global_config = global_config;
053 |     order.initial_input_amount = input_amount;
054 |     order.remaining_input_amount = input_amount;
055 |     order.expected_output_amount = output_amount;
056 |     order.number_of_fills = 0;
057 |     order.filled_output_amount = 0;
058 |     order.input_mint = input_mint;
059 |     order.input_mint_program_id = input_mint_program_id;
060 |     order.output_mint = output_mint;
061 |     order.output_mint_program_id = output_mint_program_id;
062 |     order.maker = owner;
063 |     order.status = OrderStatus::Active as u8;
064 |     order.order_type = order_type;
065 |     order.in_vault_bump = in_vault_bump;
066 |     order.last_updated_timestamp = current_timestamp.try_into().expect("Negative timestamp");
067 |
068 |     Ok(())
069 | }
```

Resolution

Fixed by commit "e583af9".

Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and the Client. This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

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

ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification. We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.