Phoenix Limit Order Book Program Audit Report

01.11.2023

– Mad Shield

Introduction

Ellipsis Labs commissioned Mad Shield to audit the Phoenix program. The audit took approximately ~5 weeks to complete starting from March 2nd and ending on April 5th 2023. This report briefly covers the program's workflow along with a short description of our general findings.

Overall, we are glad to confirm that no critical vulnerabilities were found in the code leading to loss of funds or protocol insolvency. However, there were some vulnerabilities with Medium and Low levels of severity in the program that have since been communicated to the team and addressed accordingly.

Overview

Phoenix is a Central-Limit Order Book (CLOB) decentralized exchange (DEX) protocol built on the Solana blockchain, supporting markets for spot assets. Leveraging the high-speed and low transaction fees of Solana, it intends to be the main liquidity pool for market makers to provision limit orders for digital asset markets and for DeFi traders to access the deep liquidity and efficiency of traditional order book trading venues.

Technical Specifications

Here, we go through some of the technical intricacies of Phoenix that are most relevant to the usability and security of the program.

1. **Account Structure**

   The layout of accounts on the Solana blockchain is the most important part of a program's design. A significant detail of Phoenix's account layouts is the extensive use of Red-Black trees. This data structure allows Phoenix to function without the need for an off-chain cranker.

   The implementation of all the data structures in Phoenix is separate from the implementation of the CLOB. This is great from the perspective of security analysis as the business logic for the DEX can be analyzed independently from the logic of the underlying data operations.

In fact, The Ellipsis Labs team has built Sokoban, a custom library that supports multiple compact efficient data structures. The Red-Black tree in Phoenix is instantiated using this library.

All Bids and Asks in Phoenix are stored in separate Red-Black trees in the market account, keeping a balanced representation of the order book.

2. **Interface Design**

   Phoenix has a flexible interface to represent orders that supports multiple scenarios and order types. In addition, the matching engine logic is order agnostic, solely targeted to the optimal matching of the in-flight order.

   Market makers can utilize this flexibility in the order schema, to develop more customized trading strategies and devise safety measures to protect against extreme market events or chain conditions.

   The matching engine being a sovereign component in the design further adds to the verifiability of the program's core logic allowing for more robust performance guarantees and security evaluation.

3. **Typed Units**

   Phoenix defines custom types for all the measurement units across various quantities utilized within the program. For example, amounts of Quote and Base assets are measured in QuoteLots and BaseLots types.

   Although below this abstraction layer, all these units ultimately rely on Rust primitive types like u64, this approach offers two notable advantages.

   First, it provides a convenient framework where the contributors can recognize the units of different values instantly and do mindful conversions whenever necessary without concerns about exploits involving type confusions.

   Second, the compilation process would fail if there were any operations on incompatible types such as comparison between BaseLots and QuoteLots. This adds an additional security layer to the program especially in the matching engine where it guarantees the consistency of the units employed.

Scope and Objectives

The scope of the audit included Phoenix and the Sokoban library. Ellipsis Labs delivered these two programs to Mad Shield at these github repositories:

Phoenix – https://github.com/Ellipsis-Labs/phoenix-v1
Sokoban – https://github.com/Ellipsis-Labs/sokoban

The git commits at which each repository was at the start of the audit are as follows:

Phoenix – 11eda7a3637b5e106d8b01cd8033dfea68d36c82
Sokoban – 7a2b711fae49a10be75ab9340f6dc4075e75312b

The main objectives of the audit are defined as:

- Minimizing the possible presence of any critical vulnerabilities in the program. This would include detailed examination of the code and edge case scrutinization to find as many vulnerabilities.

- 2-way communication during the audit process. This included for Mad Shield to reach a perfect understanding of the design of Phoenix and the goals of the team. Ellipsis Labs in return would expect a clear and thorough explanation of all vulnerabilities discovered during the process with potential suggestions and recommendations for fixes and code improvements.

- Clear attention to the documentation of the vulnerabilities with an eventual publication of a comprehensive audit report to the public audience for all stakeholders to understand the security status of the programs.

Methodology

After the initial contact from the Ellipsis Labs team, we did a quick read through of the source code to evaluate the scope of the work and recognize early potential footguns. As Phoenix relied on Sokoban as the library for its data structures, for the sake of simplicity and separation of duties, we took the approach of treating the Red-Black tree as a black-box tool with correct behavior and turned our focus primarily to find any vulnerabilities within the CLOB itself. This proved to be an effective technique as we uncovered a number of issues where the implementation dithered from the program's specification.

Throughout this process, we engaged in extensive communication with the team to gain an in-depth understanding of the program and its limitations in the context of the Solana runtime. During one of these sessions, we found issue **SHIELD_PHNX_001** where the order book could end up in an invalid crossed state as there couldn't be infinite matching of orders.

Phoenix doesn't use Anchor, the common framework for program development on Solana. Hence, we made a significant attempt to understand the custom account validation to ensure that no account substitutions were possible. This revealed **SHIELD_PHNX_002** where an unauthorized authority could manipulate another market's list of approved market makers. Following this access control violation, we took extra care to confirm that the program is resilient against other authority/signer mischecks. To the best of our knowledge, there are no further such breaches involving different market authorities. In addition, the program utilizes a Squad multisig wallet as the upgrade authority which is the best possible solution to manage program upgrades on Solana.

As described in the technical specification section, the order schema on Phoenix is very flexible; however, most of the order packet information is abstracted out inside the matching engine logic. Therefore, we took the same approach of examining the two components individually on their own and then in relation to each other.

While scrutinizing the different order formats, we encountered an unreachable execution path rigorously explained in **SHIELD_PHNX_003.** The matching engine logic was impressively immune to our multiple efforts to unveil potential footguns which demonstrated the considerable effort the team has put into the implementation of this module. We were however, able to find **SHIELD_PHNX_004 ,** an overly aggressive order cancellation bug in the intersection of the two logics concerning the different modes of self trade behavior.

We next shifted our focus towards Sokoban. Within this library, a generic Node Allocator forms the basis for constructing various data structures, including the Red-Black tree. Rigorous validation of primitive operations, such as addition and deletion, within this allocator confirmed their integrity. Notably, this module cleverly leverages the 0 index as a substitute for the NULL value. Thus, we exercised extra caution to make sure that there would be no issues with index collisions when accessing nodes. The library aptly handles conversions between the 1-indexed and 0-based representations.

The Red-Black tree is a complex data structure that involves separate procedures for common tree operations depending on the selected node. Sokoban appropriately adopts a proprietary, battle-tested open source implementation. Furthermore, the library comes with a valuable fuzzy testing tool with an exhaustive suite of unit tests.

In our analysis, we successfully verified that there were no deviations from the reference implementation. Moreover, we utilized the fuzzy testing tool to add extra unit tests of potentially vulnerable corner cases. In summary, our assessment affirms that Sokoban is a well-crafted library, offering Phoenix with a robust implementation of its interface.

Findings & Recommendations

In this section, we enumerate some of the findings and issues we discovered and explain their implications and resolutions.

Severity	Count
Critical	0
High	0
Medium	2
Low	2

Tab 1. Breakdown Of The Key Findings

We communicated all findings through our channels to the team and worked closely to recommend fixes and review the patches before final integration and deployment.

This report is a valid document for the status of the Phoenix codebase between

commit - 11eda7a3637b5e106d8b01cd8033dfea68d36c82 and

commit - ee9cc0333fbab6ee20861e51201951b3e9c38dc9

With resolving patches for all findings.

1. SHIELD_PHNX_001 - Limit Order Crossing *[Medium]*

The process with which the Market Makers place limit orders has a couple of delicacies. First, The order comes with a match limit which determines how many orders on the opposite side of the book would match with the current in-flight order. This is in case the order price crosses the orderbook e.g. an incoming buy order with a higher price than the current lowest price. In case not all of the incoming order is filled, then the program would return the remaining unfilled amount and would put an order at the specified price.

This could lead to a corrupted crossed orderbook state where a Bid would be placed with a higher price than the current Ask price. To avoid this situation, the program should check if the current price of the order is valid and only then proceed to place the order. Interestingly, this check was in place for PostOnly orders and the violating order would be amended to the best price on the opposite side of the book. However, this was not the case for Limit orders.

A patch was introduced to resolve this issue which you can follow in the code snippet below. Essentially, the program checks whether the order price crosses the best price on the other side and it would skip the order if that is the case without placing it.

During this analysis, we also noticed an efficiency boost could be introduced. This was specifically related to the TIF (Time In Force) feature. The program would reject the order even if the price was crossing an expired order. The code was improved to only check for active crossings of the orderbook and not prevent order placement due to a stale order.

```rust
let limit_order_crosses = if matches!(order_packet, OrderPacket::PostOnly { .. }) {
   false
} else {
   let best_price_on_opposite_book = self
       .get_book(side.opposite())
       .iter()
       .find(|(_, resting_order)| {
           !resting_order.is_expired(current_slot, current_unix_timestamp)
               && resting_order.num_base_lots > BaseLots::ZERO
       })
       .map(|(o_id, _)| o_id.price_in_ticks)
       .unwrap_or_else(|| match side {
           Side::Bid => Ticks::MAX,
           Side::Ask => Ticks::ZERO,
       });
   match side {
       Side::Bid => order_packet.get_price_in_ticks() >= best_price_on_opposite_book,
       Side::Ask => order_packet.get_price_in_ticks() <= best_price_on_opposite_book,
   }
};
```

2. **SHIELD_PHNX_002 - Market Maker Seat Access Control Violation *[Medium]***

Currently, each market on Phoenix only allows for market making by traders that have been approved by the market authority. In this permissioned setup, market makers can request to be assigned a seat which then needs to be approved by the market authority for them to start placing limit orders. Upon requesting a seat by calling the RequestSeat instruction, a seat account is created for the trader with the following seeds for the PDA derivation:

```
["seat", market_key, trader]
```

By default, the status of this account is set to NotApproved. To approve this seat request the market authority needs to invoke the ChangeSeatStatus instruction. For this operation to be safe, it is necessary to assert two conditions. First, one has to check that the seat account is owned by the phoenix program and that its address matches the PDA derived by the associated seeds. Secondly, it must be checked that the seeds used to derive the PDA match the current context in which the ChangeSeatStatus is called i.e. the market address and the trader account stored in the Seat account are indeed the market account and trader account passed to the instruction.

The program, however, would only check the first condition: that the Seat account address matches the PDA derived by the stored market and trader accounts. But it would jump over the second condition: that the market account used to validate the PDA derivation is indeed the market account in the instruction context.

This could lead to a scenario where a seat intended to be permissible only by market A to also be registered in market B enabling a trader, otherwise unauthorized, to start market making for market B. Even worse, the malicious authority for market A could change the seat status for the market makers of market B, illegally preventing market B traders from continuing their operations. They could further make themselves the sole market maker on market B giving them freedom to manipulate the market.

To prevent this privilege escalation, a patch was introduced to include an additional verification step. This step checks whether the seat account belongs to the relevant market within the instruction's context, as demonstrated in the following code snippet.

```
let (seat_address, _) = get_seat_address(market, &seat.trader);
assert_with_msg(
    seat.market == *market,
    ProgramError::InvalidAccountData,
    "Market on seat does not match market in instruction",
)?;
```

3. SHIELD_PHNX_003 - FOK Orders Fail In The No Deposit/Withdrawal Mode *[Low]*

FOK (Fill Or Kill) orders on phoenix are a subset of IOC (Immediate Or Cancel) orders where the order amount has to be equal to the minimum amount to fill (both parameters specified in the order schema). The order fails if the matched amount after order execution is less than the minimum required.

There is also an additional option in phoenix where the trader can execute trades solely based on the amount they have previously deposited into their phoenix accounts. Under such circumstances, no deposit or withdrawal i.e token transfers would happen. Instead the free fund amounts in the trader accounts would be updated to reflect the result of the trade.

To enforce FOK behavior, the program checks that neither of the matched quote or base amounts are less than the minimum amounts.

On the other hand, with the no deposit/withdrawal option enabled, the available funds in the trader state account are updated according to the matched amount and the matched amount itself is set to ZERO.

We noticed that the deposit/withdrawal event would occur before the FOK condition check i.e. by the time the matched order amounts were compared against the required minimum values, they had already been zeroed out. As a result, Phoenix would incorrectly assume that the criteria for the FOK order has not been met and therefore reject this program path from ever being reached.

A simple fix was applied swapping the execution order of these conditionals, thereby, accommodating for deposit/withdrawal free FOK orders to be possible.

4. SHIELD_PHNX_004 - Self Trade With DecrementTake Cancels Orders Aggressively *[Low]*

During order matching, if a market maker's own orders hit, they have the option to define three distinct behaviors to manage the order execution. First is to Abort, which means the market maker does not want to trade against any of their own orders. The second and third scenarios are more interesting and are CancelProvide and DecrementTake respectively.

In the case of CancelProvide, the self trade order is taken off the book but the matched amount is not affected. This facilitates automatic cancellation of limit orders while maximizing the filled take amount resulting in both order book clean-up and optimal performance.

For the DecrementTake scenario, in theory, the market maker should effectively trade against themselves, reducing the matched amount by the posted amount of their limit order. Additionally, the market should attempt to cancel the order as in CancelProvide if the remaining amount of the self trade order is zero. But if the remaining order amount is non-zero, the expected behavior is to reduce the order amount without removing it.

In practice, Phoenix treated both cases with the identical logic, leading to an inconsistency where the order was taken off the book, even if the engine did not match the entirety of the take order.

The code snippet below highlights the recommended changes to alleviate the issue.

```
SelfTradeBehavior::DecrementTake => {
   let base_lots_removed = inflight_order
       .base_lot_budget
       .min(
           inflight_order
               .adjusted_quote_lot_budget
               .unchecked_div::<QuoteLotsPerBaseUnit, BaseLots>(
                   order_id.price_in_ticks
                       * self.tick_size_in_quote_lots_per_base_unit,
               ),
       )
       .min(num_base_lots_quoted);
   self.reduce_order_inner(
       current_trader_index,
       &order_id,
       inflight_order.side.opposite(),
       Some(base_lots_removed),
       false,
       false,
       record_event_fn,
   )?;
   ..
}
```

Conclusion

In conclusion, our audit of the Phoenix program and Sokoban library has been a thorough effort to enhance security. While we identified medium and low-severity vulnerabilities, we're pleased to report no critical issues jeopardizing user funds. The identified vulnerabilities and proposed resolutions detailed in this report will enhance the reliability of Phoenix.

The audit underscores our focus on security within the expanding Solana DeFi landscape, emphasizing the ongoing need for evaluations and transparent documentation. We would like to acknowledge the Ellipsis Labs team for their cooperative approach throughout the process, which has contributed to the overall effectiveness of this collaboration.