BlockApex

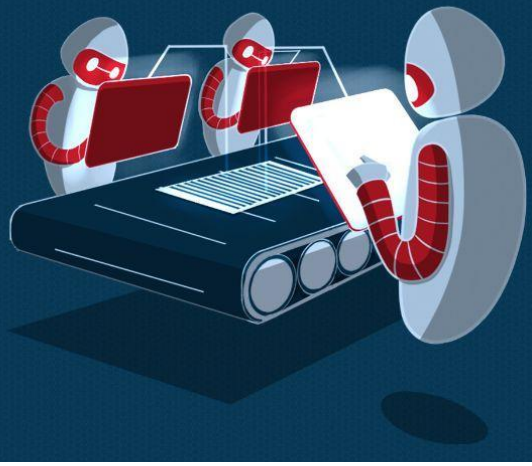
SMART CONTRACT SECURITY ANALYSIS REPORT

```
pragma solidity 0.7.0;
contract Contract {

    function hello() public returns (string) {
        return "Hello World!";
    }

    function findVulnerability() public returns (string) {
        return "Finding Vulnerability";
    }

    function solveVulnerability() public returns (string) {
        return "Solve Vulnerability";
    }
}
```



PREFACE

Objectives

The purpose of this document is to highlight any identified bugs/issues in the provided codebase. This audit has been conducted in a closed and secure environment, free from influence or bias of any sort. This document may contain confidential information about IT systems/architecture and intellectual property of the client. It also contains information about potential risks and the processes involved in mitigating/exploiting the risks mentioned below. The usage of information provided in this report is limited, internally, to the client. However, this report can be disclosed publicly with the intention to aid our growing blockchain community; under the discretion of the client.

Key understandings

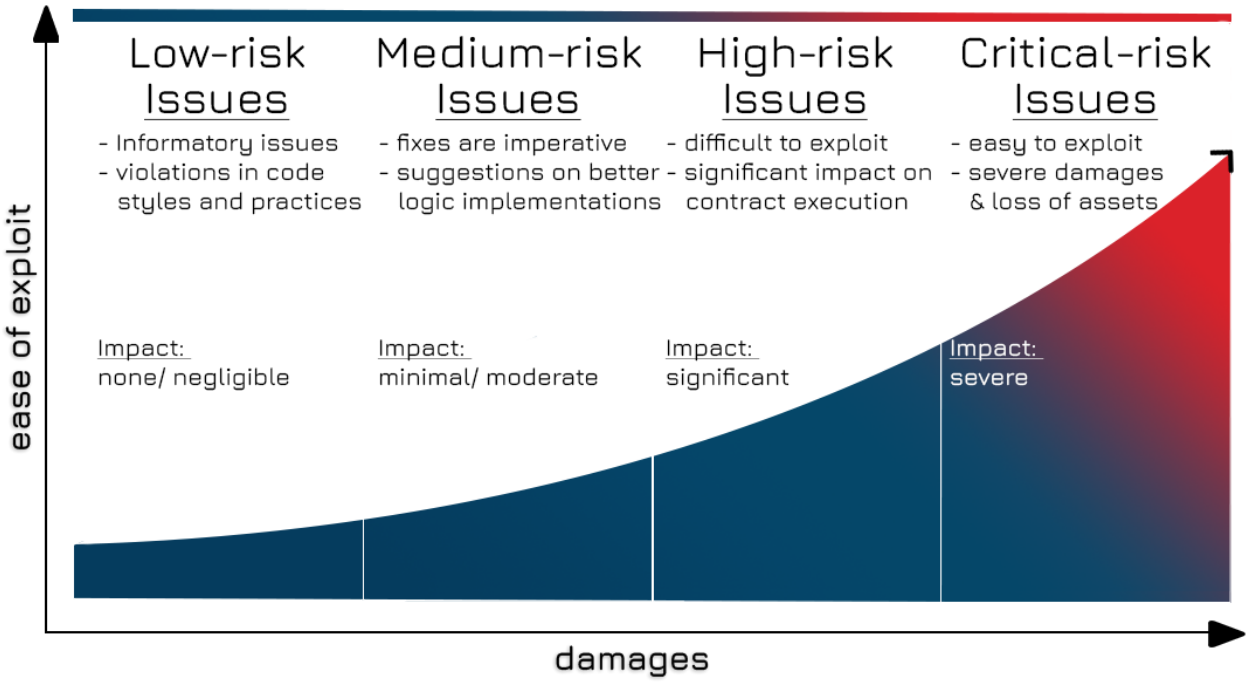


TABLE OF CONTENTS

INTRODUCTION

BlockApex (Auditor) was contracted by Spin.Finance (Client) for the purpose of conducting a Smart Contract Audit/Code Review. This document presents the findings of our analysis which started on 31st May 2022.

Name
Spin.Finance Decentralized Exchange - Order Book Market Place | Swap Functionality
Auditors
Moazzam Arif | Abdul Sami Jawed | Faizan Nehal | Muhammad Jariruddin
Platform
NEAR Protocol | Rust
Type of review
Manual Code Review | Automated Tools Analysis
Methods
Architecture Review | Functional Testing | Computer-Aided Verification | Manual Review
Git repository/ Commit Hash
549bce99171d0fe5473075937f76310348b2dca2
White paper/ Documentation
Docs | Medium | Spin Intern Guides
Document log
Initial Audit Completed: June 15th, 2022
Final Audit Completed: July 10th, 2022

Scope

The git-repository shared was checked for common code violations along with vulnerability-specific probing to detect **major issues/vulner abilities**. Some specific checks are as follows:

Code review		Functional review
Reentrancy	Unchecked external call	Business Logics Review
Ownership Takeover	ERC20 API violation	Functionality Checks
Timestamp Dependence	Unchecked math	Access Control & Authorization
Gas Limit and Loops	Unsafe type inference	Escrow manipulation
DoS with (Unexpected) Throw	Implicit visibility level	Token Supply manipulation
DoS with Block Gas Limit	Deployment Consistency	Asset's integrity
Transaction-Ordering Dependence	Repository Consistency	User Balances manipulation
Style guide violation	Data Consistency	Kill-Switch Mechanism
Costly Loop		Operation Trails & Event Generation

Project Overview

Spin is a DeFi derivative infrastructure built on NEAR Protocol, a reliable and scalable L1 solution. The on-chain order book solution offered by Spin provides a CEX-competitive experience to DeFi users.

Founded in June 2021, Spin was the first product to offer an on-chain order book solution on NEAR Protocol. The advantages of the order book model include better user experience compared to AMM, flexible liquidity, easy access for institutional traders, secure and transparent on-chain verification, opportunity to price different types of instruments, and trading robots interoperability.

System Architecture

Central Limit Order Book Model

Spin uses a single-asset pool, the liquidity of which is sent for market making to the order book. Thus, traders will always have enough liquidity for comfortable trading, and investors can profit from the sophisticated market-making mechanism developed by the Spin team.

Spot trading

The current version of the Spin spot DEX on NEAR is the first order book implementation that supports on-chain order matching and NEAR wallet connection. Currently, the Spin spot DEX is already live on mainnet: https://trade.spin.fi. Spin also provides users with an opportunity to make instant token swaps at the market price in a single click. Spin boasts lower fees compared to AMMs.

On Spin, for example, on USN/USDC, the taker fee is 0.04% and the maker's rebate is -0.02%. At the same time, NEAR Protocol's largest AMM Ref Finance charges 0.3% from swappers.

Methodology & Scope

The codebase was audited using a filtered audit technique. A band of four (4) auditors scanned the codebase in an iterative process spanning over a time of two (2) weeks.

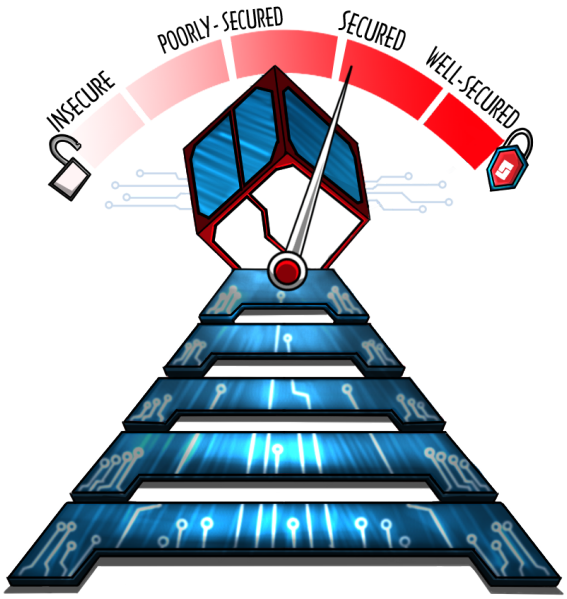
Starting with the recon phase, a basic understanding was developed and the auditors worked on developing presumptions for the developed codebase and the relevant documentation/whitepaper. Furthermore, the audit moved on with the manual code reviews with the motive to find logical flaws in the codebase complemented with code optimizations, software and security design patterns, code styles, best practices and identifying false positives that were detected by automated analysis tools.

AUDIT REPORT

Executive Summary

The analysis indicates that the contracts under scope of audit are **working properly** excluding swap functionality which contains one recent issue.

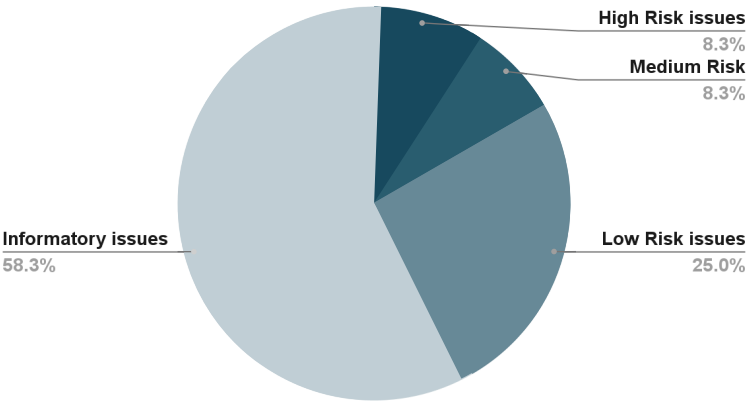
Our team performed a technique called “Filtered Audit”, where the contract was separately audited by four individuals. After a thorough and rigorous process of manual testing, an automated review was carried out using cargo-audit & cargo-tarpaulin for static analysis and cargo-fuzz for fuzzing invariants. All the flags raised were manually reviewed and re-tested to identify the false positives.



Our team found:

# of issues	Severity of the risk
0	Critical Risk issue(s)
5	High Risk issue(s)
1	Medium Risk issue(s)
4	Low Risk issue(s)
9	Informatory issue(s)

Proportion of Vulnerabilities



Key Findings

#	Findings	Risk	Status
1.	*Swap:* Logic Flaw in set_availability() will lead wrong state of the market been set	High	Acknowledged
2.	Order expiration time can be exploited	High	Fixed
3.	Potential MultiSig Failure/ Phishable Deployment	High	Fixed
4.	Potentially dangerous drop_state() method	High	Fixed
5.	Inessential parameter optional visibility	High	Fixed
6.	Unoptimized loop with redundant ops	Medium	Acknowledged
7.	Impossible ownership transfer	Low	Acknowledged
8.	Illegal Max Fee Possibility	Low	Fixed
9.	Ineffectual availability of market	Low	Acknowledged
10.	Inexistent math checks	Low	Acknowledged
11.	Inconsistent asserts and panic	Informatory	Acknowledged
12.	Unoptimized error message pattern	Informatory	Acknowledged
13.	Inconsistent arguments type	Informatory	Acknowledged
14.	Inexplicable balances variable	Informatory	Fixed
15.	Inconsistent functions to handle errors	Informatory	Acknowledged

16.	Unhandled whitelisting removal	Informatory	Acknowledged
17.	Quality of Test Cases	Informatory	Partially Fixed
18.	Incomplete condition evaluation	Informatory	Acknowledged
19.	Misidentified default order behavior	Informatory	Acknowledged

Detailed Overview

Critical-risk issues

No issues were found.

High-risk issues

1. **Logic Flaw in set_availability() will lead wrong state of the market been set**

File: In the marketplace/src/market.rs

Description:
In the latest commit a mechanism for **set_availability()** function is changed and an **assert!()** statement is included to make sure that markets are not set when "placing is on and canceling is off", the problem here is if both the placement and canceling is off then this **assert!()** statement will return true because of **!availability.allow_place** and will eventually set the market in the **set_market_option()** function of marketplace.rs contract. So because of lack of implementation of proper checks, the market will be set.

```
   pub fn set_availability(&mut self, availability: MarketAvailability) {
       assert!(
           !availability.allow_place || availability.allow_cancel,
           "Market state when placing is on and cancelling is off is not
possible."
       );
       self.availability = availability;
   }
```

Remedy:
There should be two checks, the first one will verify whether both the allow_place and allow_cancel are not false at the same thing. After that the current assert!() statement will be called. Adding a statement like above the already present assert!() statement will solve it.

```
       assert!(
```

```
            availability.allow_place || availability.allow_cancel,
            ""
        );
```

Status:
Acknowledged.

Dev Response:

Spin consider "allow_place==false and allow_cancel==false" a valid market state. Moreover, that's required for a market to be created with such a configuration, so we can
1. ensure market was created with correct base-quote pairs
2. ensure the fees and limits were set correctly

2. Order expiration time can be exploited

File: In the marketplace/src/marketplace.rs

Description:
`order ttl` can be exploited by delaying or keeping users' transactions in the mempool. As the expiry time is calculated by **`current time_stamp + ttl`**. **Current_time** is calculated at the time the transaction is included in the block. Interested parties can keep the order in waiting queues or just do not include it in the block. When the prices are favorable they can execute the order. Although to pull this kind of attack requires high technical resources, it still can be exploited.

```
fn make_order(&mut self, order: PlaceOrder, meta: PlaceOrderMeta, signer:
AccountId) -> Order {
        let expiration_time = if let Some(ttl) = order.ttl {
            // to nanoseconds
            meta.timestamp + (ttl * 1_000_000_000)
        } else {
            Timestamp(0)
        };
```

```
        self.order_id_sequence += 1;

        Order {
            id: self.order_id_sequence,
            acc: signer,
            price: order.price,
            average_price: U128(0),
            quantity: order.quantity,
            remaining: order.quantity,
            updated_at: Timestamp(0),
            created_at: meta.timestamp,
            expiration_time,
            o_type: order.order_type,
            client_order_id: order.client_order_id,
        }
    }
```

Remedy:

```
let expiration_time = ttl;
```

Status:
Fixed

3. Potential MultiSig Failure/ Phishable Deployment

File: In the spot/src/lib.rs

Description:
`env::signer_account_id()` is the account that originates a transaction on the NEAR protocol. In case of a multisig governance account management may not work and the deployer can be phished to interact with malicious contracts. Since the below contracts check on the transaction originator, this check can be bypassed. For further info please take a look at this SWC .

```
        pub fn create_market(&mut self, base: AccountId, quote: AccountId) ->
MarketID {
```

```
            let _session = self.make_session();
            let signer = env::signer_account_id();

            self.marketplace
                .create_market(signer.as_str(), base.to_string(),
quote.to_string())
        }
```

Remedy:
Use the predecessor's account so that a call coming through a multisig governance contract could be handled appropriately.

```
let caller = env::predecessor_account_id().to_string();
```

Status:
Fixed

4. Potentially dangerous drop_state() method

File: In the spot/src/lib.rs

Description:
The drop_state() function should be omitted as it is already a very unsafe method. Moreover it is calling the clear() function from currency.rs contract. The clear() function is just removing all the markets at once.

```
pub fn drop_state(&mut self, keys: Vec<String>, keep_balances: bool) {
       let _session = self.make_session();
       let signer = env::signer_account_id();
       self.marketplace
           .drop_state(signer.as_str(), keys, keep_balances)
   }

   /// Set contract root state.
   /// Used to change the state of the contract when updating the contract.
   pub fn set_root_state(&mut self, state: String) {
```

```
        let signer = env::signer_account_id();
        self.marketplace.ensure_root(signer.as_str());
        self.marketplace.ensure_markets_are_stopped();

        Promise::new(current_account_id()).function_call(
            "set_root_state_callback".to_string(),
            json!({ "state": state }).to_string().into_bytes(),
            0,
            SINGLE_CALL_GAS,
        );
    }
```

Remedy:

The drop_state() method should be discarded

Status:
Fixed

5. Inessential parameter optional visibility

File: In the spot/src/lib.rs

Description:
The parameter **`ttl`** in the **`place_ask()`** and **`place_bid()`** function should not be optional and there must be a mandatory limit for how long a specific order exists. Not giving the order a time-to-live will open it for many different attack scenarios, and the order would be at risk of MEV attacks.

```
pub fn place_bid(
        &mut self,
        market_id: MarketID,
        price: Price,
        quantity: Quantity,
        ttl: Option<u64>,
        market_order: bool,
```

```
        client_order_id: Option<u32>,
    ) -> Option<OrderID> {
        self.place_order(
            PlaceOrder {
                price,
                quantity,
                ttl,
                market_order,
                order_type: OrderSide::Bid,
                client_order_id,
            },
            market_id,
        )
    }
```

Remedy:
It is our recommendation from the security perspective that each order should always have a time-to-live.

Status:
Fixed

Medium-risk issues

6. Unoptimized loop with redundant ops

File: In the marketplace/src/market.rs

Description:

Inside the **`handle_execution_effects()`** function of market.rs contract, a continue; statement should be placed at the end of the first loop, currently it is also executing the whole function even if the particular order is empty. This is a waste of resources and gas, to make the code more optimized and secure from redundancy a continue statement should be placed.

```
fn handle_execution_effects(
        &mut self,
        balances: &mut Balances<P>,
        effects: &ExecutionEffects,
    ) -> Price {
        let execution_price = effects.applied_price();
        let mut taker_base_amount = U128(0);
        let mut taker_quote_amount = U128(0);
        for match_effect in &effects.apply_to {
            if match_effect.order.is_empty() {
                Accounts::<P>::remove_order(
                    &match_effect.order.acc,
                    self.id,
                    match_effect.order.id,
                );
            }
```

Remedy:
So it is advised to use a continue statement here and end the loop for the particular order.

Status:
Acknowledged

Low-risk issues

7. Impossible ownership transfer

Description:
There exists no function or implementation for transfer ownership or renounce ownership. These two functions are very necessary for any protocol so that a trust in potential multisig governance is built.

Remedy:
We recommend that these functions should be added in the codebase to make it more usable and resistant in the long term.

Status:
Acknowledged

8. Illegal Max Fee Possibility

File: In the marketplace/src/market.rs

Description:

Inside the **`set_fees()`** function, there should be a maximum cap on how much the fees could go up. Or what is the max limit on how much the fees could be set. Because currently it is just checking whether or not the taker_fee is greater than or equal to zero, and if the **`maker_fee`** is less than or equal to **`taker_fee`**. In the current implementation a fee of more than 100% could be set up, which is an indicator of rug-pull for the users.
Moreover there should also be a lower cap on the **`maker_fee`**. This will increase the credibility of your code.

```
pub fn set_fees(&mut self, fees: MarketFeesInput) {
        assert!(
            fees.taker_fee >= 0,
            "Taker fee should not be less than zero."
        );
        assert!(
            fees.taker_fee >= fees.maker_fee.abs(),
            "Taker fee should be greater than or equal to maker fee."
        );
        self.fees = MarketFees::new(fees);
    }
```

Remedy:

There should be a proper check for setting the max upper and lower limit for the fees and if the provided fees exceed the limit then it should discard it.

Status:
Fixed

9. Ineffectual availability of market

File: In the marketplace/src/market.rs

Description:
In the market.rs contract of order book there is a function is_running(). It will check whether or not the market is running or not, and even if a single user is available then it will return true. The current condition that it is checking is **self.availability.allow_place || self.availability.allow_cancel**
There is an inconsistency here that needs to be addressed, if the **`allow_cancel`** for any market is false and **`allow_place`** is true then it would return true as the return value for **`is_running()`** function.

```
pub fn is_running(&self) -> bool {
        self.availability.allow_place || self.availability.allow_cancel
    }
```

Remedy:
It is therefore recommended that setting the availability of the marketplace should be restricted so that whenever **`allow_cancel = false`** it should also be that **`allow_place = false`** , hence a consistent pattern of availability is achieved. In fact, it would be highly inconsistent if the canceling orders is turned off but placing orders is turned on. The **`allow_place`** could be set as false independently of **`allow_cancel`**, but the vice versa should not be true.

Status:
Acknowledged

10. Inexistent math overflows checks

Description:
In the overall code, there is a lack of underflow and overflow checks inside the cargo.toml.

Remedy:
In all the cargo.toml files, underflow and overflow checks should be defined.

Status:
Acknowledged

Informatory issues and Optimizations

11. Inconsistent asserts and panics

File: In the spot/src/lib.rs

Description:
On multiple occasions e.g. as the one below an inconsistent checking pattern is observed where the function asserts for an invariant and also uses panics in if blocks to panic the code execution. This is an anti-pattern check in case of panics and can be replaced with using asserts.

```
assert_eq!(sender_id.as_str(), signer.as_str(), "Invalid signer");
      if amount.0 == 0 {
          Near::panic("Attached deposit balance must be greater than 0.");
      };
```

Remedy:
Specifically, in the above mentioned case the statement **`amount.0 == 0`** is purely antipattern and can be replaced with the assertion **`assert_ne!`** following appropriate checks.

Status:
Acknowledged

12. Unoptimized error message pattern

File: In the spot/src/lib.rs

Description:
No standardized error codes were found in the codebase. The error handling is not upto the mark at this stage of implementation following any best practice or coding style.

```
Err(_) => Near::panic("Failed to get currency data."),
```

Remedy:
Create an error.rs file and define all the error messages as standardized error codes with explanatory messages, part of optimized best practices.

```
// Signatures and access
#[msg("Access denied")]
AccessDenied,
```

Status:
Acknowledged

13. Inconsistent arguments type

File: In the spot/src/ft.rs

Description:
In the ft.rs file, the functions **`storage_deposit()`** and **`ft_metadata()`** implement a mismatching argument type deviating from the code style and best practices. The arguments received by both functions consist of empty vectors but the ways are different in both.

```
  /// Storage deposit.
pub fn storage_deposit(currency: AccountId) {
   Promise::new(currency).function_call(
       "storage_deposit".to_string(),
"{}".as_bytes().to_vec(),
       Decimal::new(125.into(), 5).scale(NEAR_DECIMALS).value().0,
       SINGLE_CALL_GAS * 2,
   );
}
```

```
/// Returns promise for ft_metadata.
pub fn ft_metadata(address: AccountId) -> Promise {
   Promise::new(address).function_call("ft_metadata".to_string(), vec![], 0,
SINGLE_CALL_GAS)vec![]
}
```

Remedy:
It is therefore recommended that the functions should implement a singular pattern of sending arguments to a low level function call so that consistency is maintained and readability is improved.

Status:
Acknowledged

14. Inexplicable balances variable

File: In the spot/src/lib.rs

Description:
In the lib.rs file in the spot directory, the drop_state() function asks for the **keep_balances** boolean variable to confirm whether to keep balances before dropping and clearing a market state. This check is assumed to be an obvious pattern as setting it negative will cost in losing the user's internal balance collections.

```
pub fn drop_state(&mut self, keys: Vec<String>, keep_balances: bool) {
      let _session = self.make_session();
      let signer = env::signer_account_id();
      self.marketplace
          .drop_state(signer.as_str(), keys, keep_balances)
  }
```

Remedy:
It is suggested that the param be removed from the function. If the need is necessary then the following check should be modified to only remove non-zero balances.

```
for key in keys {
          let key = base64::decode(key).expect("Invalid key");
          if keep_balances && key.starts_with(&balances_key) {
              continue;
```

```
            }
            P::remove(&key);
        }
```

Status:
Fixed

15. Inconsistent functions to handle errors

File: In the marketplace/src/marketplace.rs

Description:

In the **`get_orders()`** function, a **`if Some`** and **`else`** statement should be added so that if none of the orders are found for users in a particular market, then it should throw a **`panic()`** statement. This will make the code more consistent with the error handling. The panic statement was implemented in the get_order(0 function, so it should also be implemented in **`get_order()`**.

```
pub fn get_order(
        &self,
        market_id: MarketID,
        account_id: AccountId,
        order_id: OrderID,
    ) -> Order {
        let market = self.markets.get(&market_id).expect("Market not found.");
        if let Some(order) = market.get_order(account_id, order_id) {
            order
        } else {
            P::panic("Order not found.");
        }
    }
```

```
    /// Returns list of user [Order].
    pub fn get_orders(&self, market_id: MarketID, account_id: AccountId) ->
Vec<Order> {
        let market = self.markets.get(&market_id).expect("Market not found.");
        market.get_user_orders(&account_id)
    }
```

Remedy:

The Panic statement should also be implemented in the **`get_orders()`** function just like it was implemented in **`get_order()`**.

Status:
Acknowledged

16. Unhandled whitelisting removal

File: In the marketplace/src/marketplace.rs

Description:
Inside this contract, there are functions to set the whitelist and fetch them but there is no function to remove the whitelist. In the long run this could prove to be worrisome as it will keep on accumulating the whitelist but there will be no way to remove the old and unnecessary whitelist. It will also be effective on the storage as data that is useless will take up resources.

Remedy:

A method that would be used to remove the whitelist in the marketplace.rs contract should be created.

Status:
Acknowledged

17. Quality of Test Cases

Description:
The test cases that are written are only negative test cases and there is no positive test coverage for the provided code. A full positive test coverage should be provided for the functionalities.

Remedy:

Positive test cases should be properly defined.

Status:
Acknowledged

18. Incomplete condition evaluation

File: In the marketplace/src/order_book/order.rs

Description:

In the **from(b: u8)** function it will return **OrderSide: Bid** if **b** will be **0** and for all the other numbers it will return **OrderSide: Ask**.

```
impl From<u8> for OrderSide {
    fn from(b: u8) -> Self {
        if b == 0 {
            OrderSide::Bid
        } else {
            OrderSide::Ask
        }
    }
```

Remedy:
Our recommendation is an **`else if`** statement should be created that will check whether **b==1** then it will return OrderSide: Ask and for the else it will throw an error.

Status:
Acknowledged

19. Misidentified default order behavior

File: In the marketplace/src/order_book/order.rs

Description:

Inside the **default()** function it is returning the **OrderSide: Ask**. The default behavior should not be OrderSide: Ask.

```
impl Default for OrderSide {
    fn default() -> Self {
        OrderSide::Ask
    }
}
```

Remedy:

There should not be any default OrderSide, instead all the OrderSide should be defined accordly at runtime.

Status:
Acknowledged

DISCLAIMER

The smart contracts provided by the client for audit purposes have been thoroughly analyzed in compliance with the global best practices till date w.r.t cybersecurity vulnerabilities and issues in smart contract code, the details of which are enclosed in this report.

This report is not an endorsement or indictment of the project or team, and they do not in any way guarantee the security of the particular object in context. This report is not considered, and should not be interpreted as an influence, on the potential economics of the token, its sale or any other aspect of the project.

Crypto assets/tokens are results of the emerging blockchain technology in the domain of decentralized finance and they carry with them high levels of technical risk and uncertainty. No report provides any warranty or representation to any third-Party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third-party should rely on the reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project.

Smart contracts are deployed and executed on a blockchain. The platform, its programming language, and other software related to the smart contract can have its vulnerabilities that can lead to hacks. The scope of our review is limited to a review of the Solidity code and only the Solidity code we note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks.

This audit cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.