hexens x FUEL

Security Review Report for Fuel

October 2025

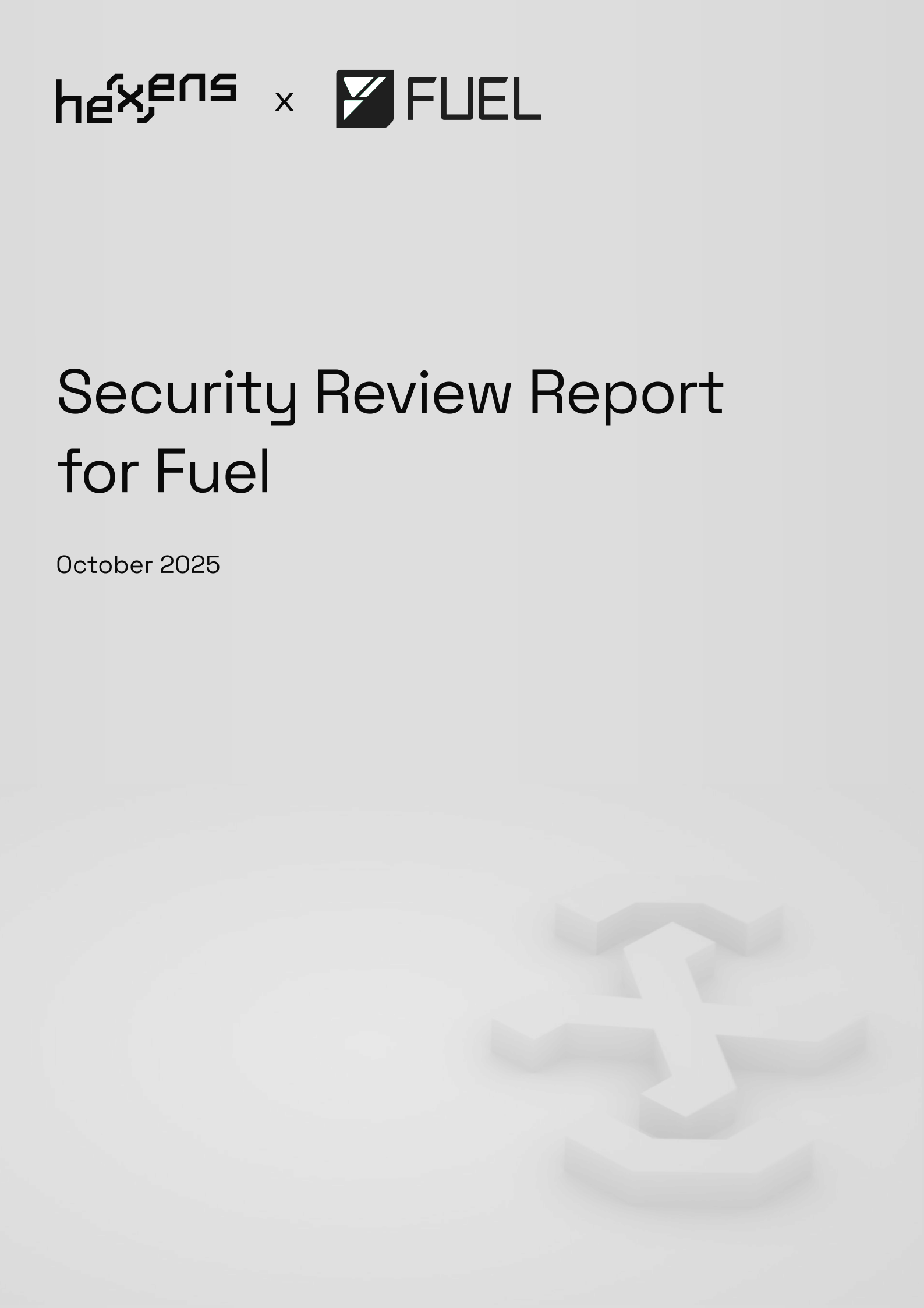


Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Security Review Details

Review Led by

Jahyun Koo, Lead Security Researcher

Scope

The analyzed resources are located on:

- https://github.com/FuelLabs/fuel-o2/tree/081e12e474149763bf1919bf1001d85a746e794d/packages/sway-v2/contracts

The issues described in this report were fixed in the following commits:

- https://github.com/FuelLabs/fuel-o2/pull/1226
- https://github.com/FuelLabs/fuel-o2/pull/1230
- https://github.com/FuelLabs/fuel-o2/pull/1110

Changelog

30 September 2025	Audit start
13 October 2025	Initial report
21 October 2025	Revision received
30 October 2025	Final report

3. Executive Summary

This report covers the security review of Fuel O2, an orderbook developed by Fuel Labs in Sway for Fuel Network. The orderbook has automatic matching upon order creation and allows users to create trade accounts.

Our security assessment was a full review of the code, spanning a total of 2 weeks.

During our review, we identified 1 high severity that could have lead to asset loss for single users when interacting with the protocol.

We also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

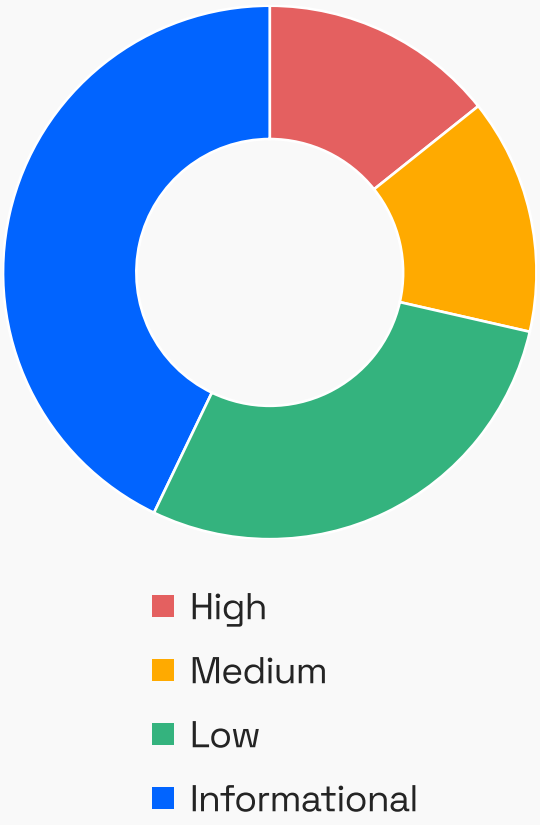
- **Issue Symbolic Codes**

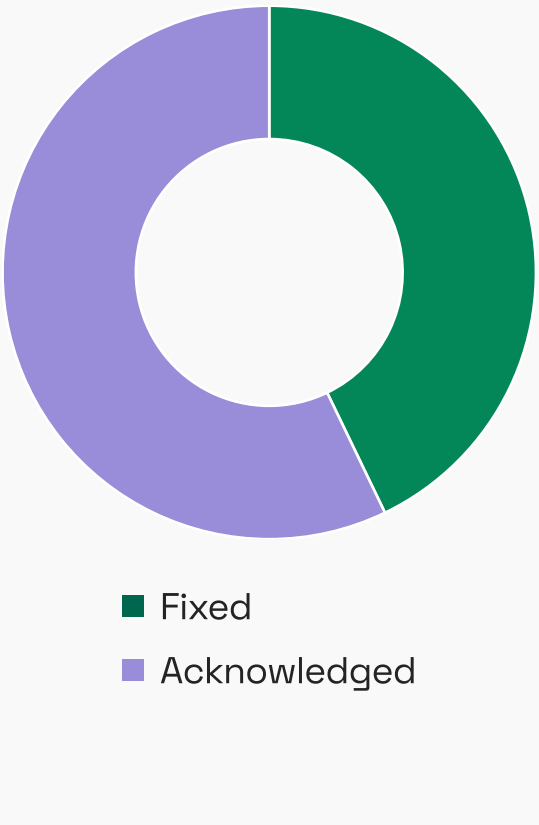
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	0
High	1
Medium	1
Low	2
Informational	3
Total:	**7**





6. Weaknesses

This section contains the list of discovered weaknesses.

FUEL11-7 | No amount out protection on Market order

Fixed

Severity:	Probability:	Impact:
High	Unlikely	Critical

Path:

packages/sway-v2/contracts/order-book/src/main.sw:create_order

Description:

The Fuel O2 order book has a **Market** order that allows users to create orders at the current best price on the market. Even though this feature is very useful for doing easy swaps for users, it is still important to have safeguards in place.

However, currently the order book simply takes the best price as the **min** for a buy order and the **max** for a sell order:

```
fn get_best_price(taker_price: Option<u64>, side: Side) -> Option<u64> {
    match side {
        Side::Buy => {
            // Get the minimum sell price
            match (storage::orderbook::v1.sell_map.min(), taker_price) {
                (Some(maker_price), Some(taker_price)) => {
                    // If the minium sell price is less than the taker price, match with it
                    if maker_price < taker_price {
                        Some(maker_price)
                    } else {
                        Some(taker_price)
                    }
                },
                (Some(maker_price), None) => Some(maker_price),
                (None, Some(taker_price)) => Some(taker_price),
                (None, None) => None,
            }
        },
        Side::Sell => {
            // Get the maximum buy price
```

```
            match (storage::orderbook::v1.buy_map.max(), taker_price) {
                (Some(maker_price), Some(taker_price)) => {
                    // If the maximum buy price is greater than the taker price, match with it
                    if maker_price > taker_price {
                        Some(maker_price)
                    } else {
                        Some(taker_price)
                    }
                },
                (Some(maker_price), None) => Some(maker_price),
                (None, Some(taker_price)) => Some(taker_price),
                (None, None) => None,
            }
        },
    }
}
```

Though this is technically correct, it comes with the possibility of exploitation.

More specifically, an attacker can watch for a Market order and front-run the market to change the current market price to something that leaves the victim with nothing.

For example, consider an empty pool:

1. The attacker posts a Limit order at a good price (say selling 1 ETH at 1000 USDC).
2. A victim creates a Market order and sends 1000 USDC.
3. The attacker front-runs and fills their own Limit order, while also creating a new Limit order selling 0.0001 ETH for 1000 USDC.
4. The victim's Market order sees the new Limit order as the best price and fills all 1000 USDC for nearly nothing.

Remediation:

We recommend adding a parameter to the **Market** order enum to allow a user to set a min/max price or a minimum amount out value. This value should be checked against at the end when it is filled.

FUEL11-1 | Blacklisted users can cancel orders via cancel_order and receive funds

Acknowledged

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/order-book/src/main.sw#L1725-L1818

Description:

The **cancel_order** function does not enforce blacklist restrictions. Normally, blacklisted trader's their orders should be canceled via **cancel_blacklist_orders**, which increases **settled_balances** but does not transfer funds, keeping them effectively frozen.

However, blacklisted traders can call **cancel_order** directly and receive their funds, bypassing this restriction.

```
#[storage(read, write)]
fn cancel_order(order_id: OrderId, cancel_type: CancelType) -> bool {
    let (order_num, price, side, _order_type) = match decode_order_id(order_id) {
        Some((n, p, s, t)) => (n, p, s, t),
        None => return false,
    };

    let sender = msg_sender().unwrap();
    let deque = match side {
        Side::Buy => storage::orderbook::v1.buys.get(price),
        Side::Sell => storage::orderbook::v1.sells.get(price),
    };

    let order = deque.get(order_num);
    if order.is_none() {
        // Avoid failing when canceling orders that do not exist just return false
        // This is done to avoid issues when orders have been executed before the
        // cancel action was executed
        return false;
    }
    let order = order.unwrap();

    match cancel_type {
```

```
    CancelType::Default => require(order.trader_id == sender, OrderCancelError::NotOrderOwner),
    CancelType::Blacklist => require(
        balance_of(
            BLACK_LIST_CONTRACT
                .unwrap(),
            AssetId::new(BLACK_LIST_CONTRACT.unwrap(), order.trader_id.bits()),
        ) > 0,
        OrderCancelError::TraderNotBlacklisted,
    ),
    CancelType::ForceCancel => {},
}

// Get the coins which are owed.
let coins_to_return = match side {
    Side::Buy => {
        // TODO: Handle dust
        // if deque.unsafe_get_head_index() == order_num {
        //     // If the canceled order is the head, it might have been partially matched and have a remainder
        //     let remaining = deque.unsafe_read_remainder();
        //     if remaining > 0 {
        //         remaining
        //     } else {
        //         quote_coins_from_quantity(order.quantity, price)
        //     }
        // } else {
            quote_coins_from_quantity(order.quantity, price)
        // }
    },
    Side::Sell => order.quantity,
};

let _ = deque.remove(order_num);
// Update the minimum and maximum prices
if deque.len().try_read().unwrap_or(0) == 0 {
    match side {
        Side::Buy => {
            storage::orderbook::v1.buy_map.unset(price);
        },
        Side::Sell => {
            storage::orderbook::v1.sell_map.unset(price);
        },
    }
```

```
    }

    match cancel_type {
        CancelType::Default => {
            match side {
                Side::Buy => transfer(order.trader_id, QUOTE_ASSET, coins_to_return),
                Side::Sell => transfer(order.trader_id, BASE_ASSET, coins_to_return),
            };
            OrderCancelledEvent::new(order_id).log();
        },
        _ => {
            let trader_balance =
storage::orderbook::v1.settled_balances.get(order.trader_id).try_read().unwrap_or((0, 0));
            match side {
                Side::Buy => storage::orderbook::v1.settled_balances.insert(
                    order.trader_id,
                    (trader_balance.0, trader_balance.1 + coins_to_return),
                ),
                Side::Sell => storage::orderbook::v1.settled_balances.insert(
                    order.trader_id,
                    (trader_balance.0 + coins_to_return, trader_balance.1),
                ),
            };
            OrderCancelledInternalEvent::new(order_id).log();
        }
    }

    true
}
```

Remediation:

Check if the user is blacklisted at the beginning of the function **cancel_order**.

Commentary from the client:

"Acknowledged, but will remain as is. This is intended functionality, blacklist is only ended to prevent the creation of orders."

FUEL11-5 | Market Registration Logic Allows Duplicate or Illogical Asset Pairs

Fixed

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

packages/sway-v2/contracts/schema/src/register.sw#L29-L47

Description:

The order-book-registry contract's register_order_book function is susceptible to two related issues regarding asset pairs:

1. **Duplicate Markets with Flipped Assets:** The contract uses the **AssetPair** tuple **(base, quote)** as a key for market registration. Because tuples are order-sensitive, **(ASSET_A, ASSET_B)** and **(ASSET_B, ASSET_A)** are treated as distinct keys. This allows for the registration of two different **order-book** contracts for what is logically the same trading pair, which could lead to fragmented liquidity and user confusion.
2. **Markets with Identical Assets:** There is no validation to prevent an **AssetPair** where the base and quote assets are the same (e.g., **(ASSET_A, ASSET_A)**). Registering such a market is illogical and could lead to undefined behavior or locked funds within the associated **order-book** contract, as its logic may not be designed to handle identical base and quote assets.

This can fragment liquidity and confuse integrations.

```
impl MarketId {
    /// Creates a new MarketId from base and quote assets.
    ///
    /// # Arguments
    ///
    /// * `base_asset` - The asset being traded
    /// * `quote_asset` - The asset used for pricing
    ///
    /// # Returns
    ///
    /// * `MarketId` - A new market identifier
    pub fn new(base_asset: AssetId, quote_asset: AssetId) -> Self {
        Self {
            base_asset,
            quote_asset,
        }
```

```
    }
}
```

Remediation:

- Enforce a Canonical Asset Pair Order: Before using the **asset_pair** as a storage key, normalize it by enforcing a consistent ordering (e.g., by comparing the **b256** values of the two **AssetId**s). This will ensure that **(ASSET_A, ASSET_B)** and **(ASSET_B, ASSET_A)** are treated as the same market.
- Prevent Identical Assets: Add a check to ensure that the two **AssetIds** within the **asset_pair** are not identical before allowing a market to be registered.

FUEL11-2 | Maker Quote Dust Not Accounted For When Fractional Prices Are Allowed

Acknowledged

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

packages/sway-v2/contracts/order-book/src/main.sw#L1699-L1723

Description:

When the **ALLOW_FRACTIONAL_PRICE** configurable is set to **true**, the contract permits partial fills where the quote amount per match is calculated via **floor(quantity * price / BASE_DECIMALS)**. Across multiple fills, the sum of these individual floored amounts can be less than the single floored amount for the total quantity, leaving a "quote dust" remainder.

Current implementation does not track or settle this remainder for buy-side makers:

- The per-price deque remainder field exists but is not updated (TODOs present).
- A maker-dust settlement helper exists but is not invoked.
- The cancellation path ignores head remainder on the buy side.

As a result, under repeated partial fills at prices not perfectly aligned with **BASE_DECIMALS**, a small residual quote amount can remain uncredited to the maker and locked in the contract. This behavior is configuration-gated and does not occur when **ALLOW_FRACTIONAL_PRICE** is **false**.

```
#[storage(read, write)]
fn settle_maker_dust(
    maker_trade: StorageOrder,
    ref mut maker_remainder: u64,
    quote_coins: u64,
    ref mut maker_list: StorageKey<SparseDeque>,
) {
    // The maker is buying and providing quote
    let total_remainder = maker_remainder - quote_coins;
    if total_remainder > 0 {
        // Add the remainder to maker's settled balances
        let maker_balance =
storage::orderbook::v1.settled_balances.get(maker_trade.trader_id).try_read().unwrap_or((0, 0));
        storage::orderbook::v1
            .settled_balances
            .insert(
```

```
                maker_trade
                    .trader_id,
                (maker_balance.0, maker_balance.1 + total_remainder),
            );
        // Reset the remainder
        maker_list.unsafe_write_remainder(0);
    }
    maker_remainder = 0;
}
```

Remediation:

- Track buy-side maker remainder at the head of each price level during partial fills
- Settle any tracked remainder to the maker when the order is fully filled or removed
- Update cancellation to include the tracked remainder when canceling a buy-side head order with partial fills.

Commentary from the client:

“Acknowledged, but will remain as is. Intentionally removed.”

FUEL11-3 | Market Orders Over-Constrained by Price Validation

Acknowledged

Severity:	Probability:	Impact:
Informational	Likely	Informational

Path:

packages/sway-v2/contracts/order-book/src/main.sw#L179-L289

Description:

The **create_order** function requires all orders, including **OrderType::Market**, to pass price-related validations (**PRICE_PRECISION**, **quote_would_truncate**, **PRICE_WINDOW**). However, the provided price for a market order is ignored during execution, which instead uses the best available price from the book.

This design forces callers to supply a valid but ultimately unused price and can cause legitimate market orders to be unexpectedly rejected by constraints like **PRICE_WINDOW**.

```
#[storage(read, write), payable]
fn create_order(order_args: OrderArgs) -> OrderId {
    let tx_start_gas = global_gas();

    require_not_paused();

    // Ensure the order args are valid
    require(
        order_args
            .quantity != 0 && order_args
            .price != 0,
        OrderCreationError::InvalidOrderArgs,
    );
    let mut order_id = b256::zero();
    let msg_asset = msg_asset_id();
    let trader = msg_sender().unwrap();

    // Only check whitelist contract is one is set
    if WHITE_LIST_CONTRACT.is_some() {
        // Ensure the whitelist contract has the asset with this trader's id as the SubId
        require(
            balance_of(
                WHITE_LIST_CONTRACT
```

```
                .unwrap(),
            AssetId::new(WHITE_LIST_CONTRACT.unwrap(), trader.bits()),
        ) > 0,
        OrderCreationError::TraderNotWhiteListed,
    );
}

if BLACK_LIST_CONTRACT.is_some() {
    require(
        balance_of(
            BLACK_LIST_CONTRACT
                .unwrap(),
            AssetId::new(BLACK_LIST_CONTRACT.unwrap(), trader.bits()),
        ) == 0,
        OrderCreationError::TraderBlackListed,
    );
}

// Determine the trade side and validate the asset.
let side = if msg_asset == QUOTE_ASSET {
    Side::Buy
} else if msg_asset == BASE_ASSET {
    Side::Sell
} else {
    // Configurables do not support match statements in Sway, so we need to have an if statement.
    revert_with_log(OrderCreationError::InvalidAsset);
    Side::Buy // This will never be reached.
};

// Check to ensure price is truncated
// NOTE: If the price falls below the PRICE_PRECISION, the contract will no longer accept orders.
// For example, if price precision is $0.01, any orders under $0.01 will fail.
require(
    order_args
        .price % PRICE_PRECISION == 0,
    OrderCreationError::PricePrecision,
);

// Verify the price provided does not result in a fractional order
if !ALLOW_FRACTIONAL_PRICE {
    require(
        !quote_would_truncate(order_args.quantity, order_args.price),
        OrderCreationError::FractionalPrice,
```

```
            );
        }

        // Validate the input amount against the order args.
        let msg_amount = msg_amount();
        match side {
            Side::Buy => require(
                msg_amount == quote_coins_from_quantity(order_args.quantity, order_args.price) && msg_amount
>= MIN_ORDER,
                OrderCreationError::InvalidInputAmount,
            ),
            Side::Sell => require(
                msg_amount == order_args
                    .quantity && quote_coins_from_quantity(order_args.quantity, order_args.price) >= MIN_ORDER,
                OrderCreationError::InvalidInputAmount,
            ),
        }

        // Validate the price collar
        match storage::orderbook::v1.last_traded_price.try_read() {
            Some((last_traded_price, _)) => {
                if PRICE_WINDOW > 0 {
                    // Upscale to u256 to avoid overflows
                    let last_price = asm(r: (0u64, 0u64, 0u64, last_traded_price)) {
                        r: u256
                    };
                    let price = asm(r: (0u64, 0u64, 0u64, order_args.price)) {
                        r: u256
                    };
                    let price_window = asm(r: (0u64, 0u64, 0u64, PRICE_WINDOW)) {
                        r: u256
                    };

                    // 0x64u256 = 100u64 - Sway does not support 100u256 so hex must be used.
                    // Price must be +/- a percentage of the price window.
                    // Asserts that price is greater than (last_traded_price * (100 - price_window) / 100)
                    // Asserts that price is less than (last_traded_price * (100 + price_window) / 100)
                    require(
                        price >= (last_price * (0x64u256 - price_window)) / 0x64u256 && price <= (last_price *
(0x64u256 + price_window)) / 0x64u256,
                        OrderCreationError::PriceExceedsRange,
                    );
                }
```

```
        },
        None => {},
    }
    ...
```

Remediation:

- For OrderType::Market, bypass price-based checks; use msg_amount as spend cap and derive fills from book, applying MIN_ORDER to executed amounts.
- Optionally support a slippage bound.

Commentary from the client:

“Acknowledged, but will remain as is.”

FUEL11-4 | QUANTITY_PRECISION Configurable is Not Enforced

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

packages/sway-v2/contracts/order-book/src/main.sw#L718-L721

Description:

The **QUANTITY_PRECISION** configurable is defined but, unlike **PRICE_PRECISION**, is not enforced in **create_order**. This may result in small amounts of base asset dust and inconsistent behavior with what the UI displays or expects.

```
fn get_quantity_precision() -> u64 {
    QUANTITY_PRECISION
}
```

Remediation:

- Enforce quantity % QUANTITY_PRECISION == 0 for limit orders at creation.
- For market orders, enforce only when the order will be stored.

Commentary from the client:

"Acknowledged, but will remain as is. Required by the API."

FUEL11-6 | Debug Log Statement in Production Code

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

packages/sway-v2/contracts/libs/src/heap.sw#L165-L177

Description:

The **MaxHeap<T>::peek()** function contains a debug log statement **log(1)** of **heap.sw**. This leftover from development emits an unnecessary event when called, adding minor gas overhead.

```
// Peek at the maximum element without removing it
#[storage(read)]
pub fn peek(self) -> Option<T> {
    let data = self.as_vec();
    match data.get(0) {
        Some(key) => {
            log(1);
            Some(key.read())
        },
        None => None,
    }
}
```

Remediation:

Remove the **log(1)**; statement from **heap.sw**.


hexens
x
FUEL