hexens x FUEL

Security Review Report for Fuel

August 2025

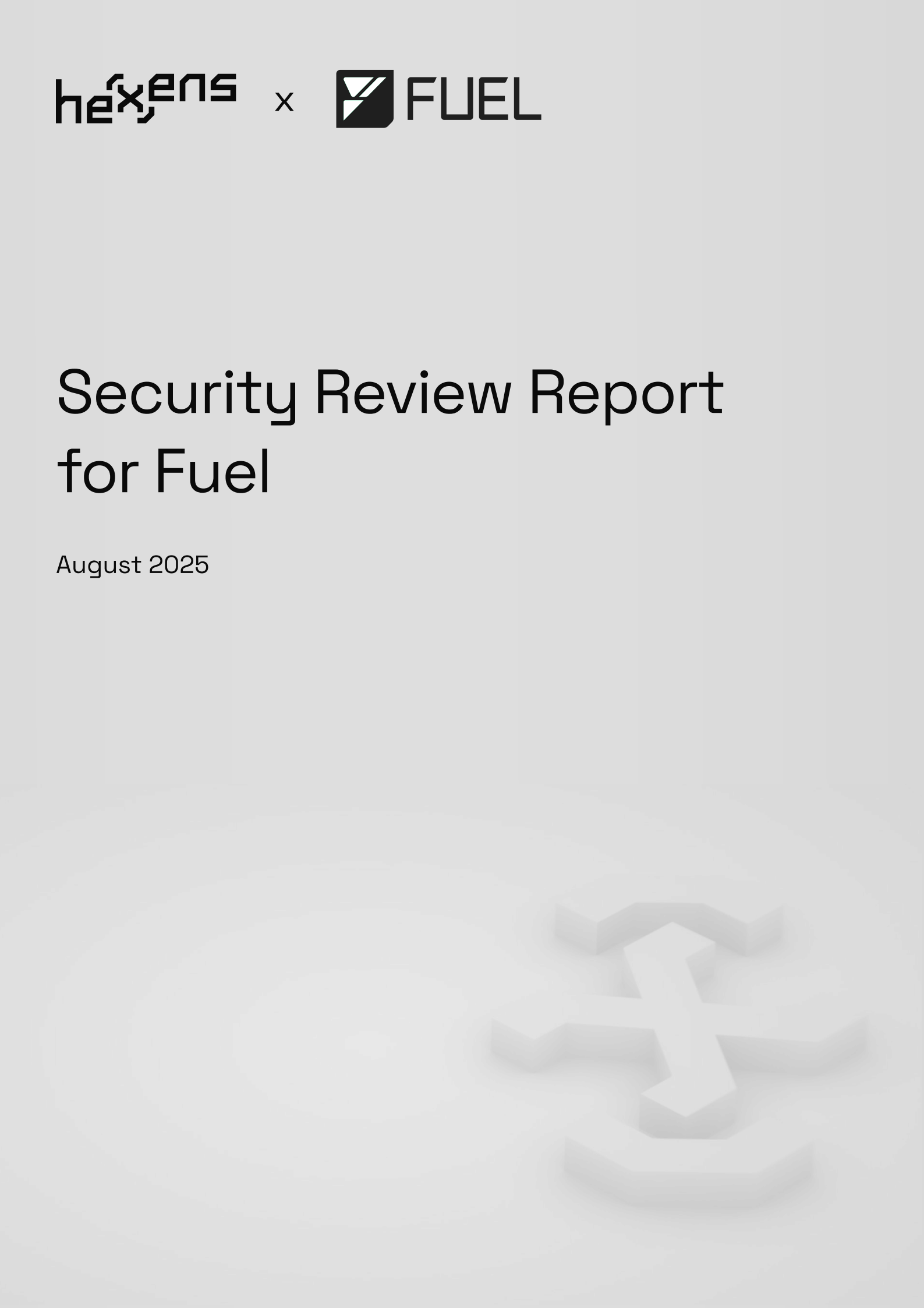


Table of Contents

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. Executive Summary

This report covers the security review 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 scope, spanning a total of 2 weeks.

During our review, we identified 3 high severity vulnerabilities, which could have resulted in loss of principal assets.

We also identified several minor severity vulnerabilities and code optimizations.

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.

3. Security Review Details

Review Led by

Kasper Zwijsen, Head of Audits

Scope

The analyzed resources are located on:

- https://github.com/FuelLabs/fuel-o2/tree/327cb0ca43c8d4b53f142c45f75d14d7137386f3/packages/sway-v2/contracts (2 weeks left)
- https://github.com/FuelLabs/fuel-o2/tree/8389a84910bb9a5c5588bf4eb981f49d0a1e497d/packages/sway-v2/contracts (1 week left)
- https://github.com/FuelLabs/fuel-o2/tree/0623976f4808e67f9ce3b32a94ffeef51ac3ecf0/packages/sway-v2/contracts (2 days left)

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

- https://github.com/FuelLabs/fuel-o2/tree/170b14e3a3caddfefbae19e975b3d32e5eb90273/packages/sway-v2/contracts

Changelog

11 August 2025	Audit start
25 August 2025	Initial report
3 September 2025	Revision received
9 September 2025	Final report

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.

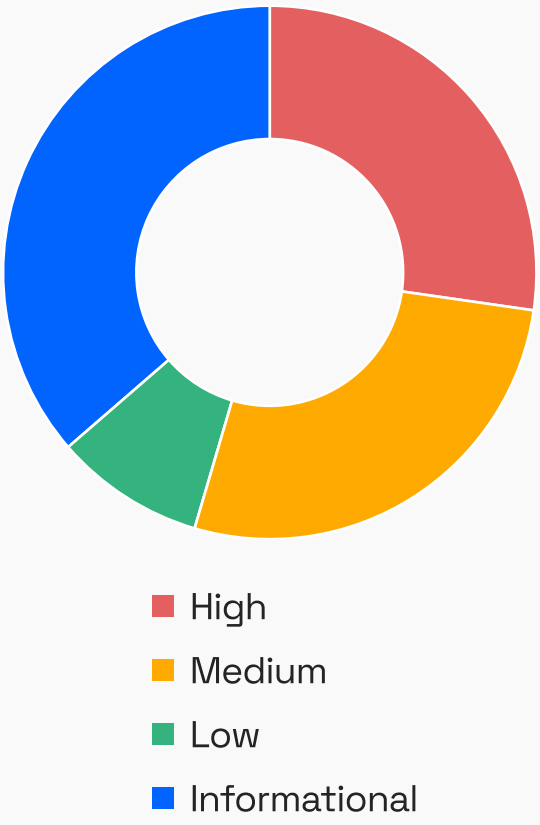
- **Issue Symbolic Codes**

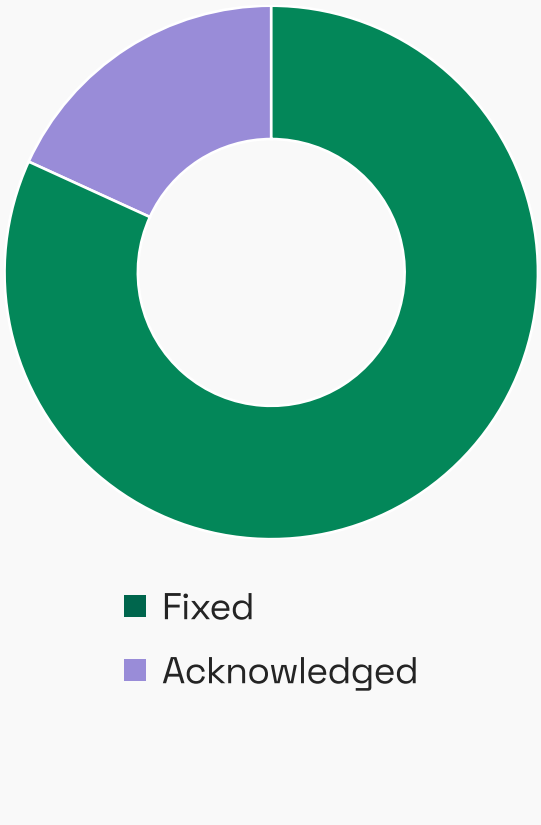
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

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	3
Medium	3
Low	1
Informational	4
Total:	**11**





6. Weaknesses

This section contains the list of discovered weaknesses.

FUEL10-1 | Incorrect transfer amount in match order

Fixed

Severity:	Probability:	Impact:
High	Unlikely	Critical

Path:

packages/sway-v2/contracts/order-book/src/main.sw:match_orders#L325-L329

Description:

During a manual match order through **match_orders**, it transfers funds to the seller (maker) after a trade, it transfers the quote asset to the seller using **fill_quantity / best_bid_price**.

However this is incorrect given the different decimals for the base asset (e.g. 9 decimals) and the quote asset (e.g. 6 decimals). This will result inaccurate fund transfers and could lead to theft of principals assets from the contract.

```
#[storage(read, write)]
fn match_orders() {
    require_not_paused();
    // Iterate overlaping prices until sells are greater than the buys
    while storage::orderbook::v1
        .buy_map
        .max()
        .is_some()
    && storage::orderbook::v1
        .sell_map
        .min()
        .is_some()
        && storage::orderbook::v1
            .buy_map
            .max()
            .unwrap() >= storage::orderbook::v1
            .sell_map
            .min()
            .unwrap() {
        let best_bid_price = storage::orderbook::v1.buy_map.max().unwrap();
```

```
            let best_bid_list =
storage::orderbook::v1.buys.get(best_bid_price);
            let (best_bid_key, best_bid_order_num) =
best_bid_list.front().unwrap();
            let mut best_bid = best_bid_key.read();
            let best_bid_order_id = asm(
                val: (best_bid_order_num, best_bid_price, 0u64,
best_bid.order_type),
            ) {
                val: b256
            };

            let best_ask_price =
storage::orderbook::v1.sell_map.min().unwrap();
            let best_ask_list =
storage::orderbook::v1.sells.get(best_ask_price);
            let (best_ask_key, best_ask_order_num) =
best_ask_list.front().unwrap();
            let mut best_ask = best_ask_key.read();
            let best_ask_order_id = asm(
                val: (best_ask_order_num, best_ask_price, 1u64,
best_ask.order_type),
            ) {
                val: b256
            };

            let fill_quantity = best_bid.quantity.min(best_ask.quantity);
            best_bid.quantity -= fill_quantity;
            best_ask.quantity -= fill_quantity;

            // Update bid book
            if best_bid.quantity == 0 {
                // Order completely filled
                let _ = best_bid_list.pop_front();

                // If there are no more buys at this price, update the price
heap
                if best_bid_list.is_empty() {
                    storage::orderbook::v1.buy_map.unset(best_bid_price);
                }
            } else {
                // Order filled partially, update to the new quantity
                best_bid_key.write(best_bid);
            }
```

```
            // Update ask book
            if best_ask.quantity == 0 {
                // Order completely filled
                let _ = best_ask_list.pop_front();

                // If there are no more sells at this price, update the price
heap
                if best_ask_list.is_empty() {
                    storage::orderbook::v1.sell_map.unset(best_ask_price);
                }
            } else {
                // Order filled partially, update to the new quantity
                best_ask_key.write(best_ask);
            }

            // Transfer traded assets
            transfer(
                best_ask
                    .trader_id,
                QUOTE_ASSET,
                fill_quantity / best_bid_price,
            );
            transfer(best_bid.trader_id, BASE_ASSET, fill_quantity);

            OrderMatchedEvent::new(
                best_bid_order_id,
                best_ask_order_id,
                fill_quantity,
                best_ask_price,
            )
                .log();

                // TODO: Determine if this is needed when the while loops
checks for `is_some()` anyway
                // if best_bid_list.is_empty() || best_ask_list.is_empty() {
                //     log("No more orders to match");
                //     break;
                // }
        }
    }
```

Remediation:

Use the existing helper function that correctly calculates the quote amount with decimals:

```
transfer(
    best_ask.trader_id,
    QUOTE_ASSET,
--  fill_quantity / best_bid_price,
++  quote_amount_from_quantity(fill_quantity, best_bid_price),
);
```

```
fn quote_amount_from_quantity(quantity: u64, price: u64) -> u64 {
    let total = quantity.as_u256() * price.as_u256();
    (total / BASE_DECIMALS.as_u256()).try_into().unwrap()
}
```

FUEL10-4 | Signature confusion due to EIP-712 non-compliance

Fixed

Severity:	Probability:	Impact:
High	Unlikely	Critical

Path:

packages/sway-v2/contracts/libs/src/signature.sw:only_proxy_owner_with_signature#L62-L87

Description:

The **trade-account** contract allows the owner to manage assets using the contract as a smart wallet. The contract uses both a session and direct signature validation to allow the owner to withdraw or call other contracts.

However, the **function only_proxy_owner_with_signature** that is used for signature validation, takes the signed data as bytes and does not correctly distinguish between domain and functional data, e.g. by adding a domain type hash and a functional type hash.

Because of this, it becomes possible for an attacker to take a signature of the owner from any other place (i.e. a **call_contract** signature into **withdraw**, or a signature for a completely different contract/chain) and submit it in **withdraw** or **call_contract**. The result of this signature confusion is an unauthorised and unintended action, potentially leading to direct principal asset loss.

```
#[storage(read, write)]
pub fn only_proxy_owner_with_signature<T>(signature: Option<Signature>, args:
T) -> Result<Identity, SignatureError>
where
    T: AbiEncode,
{
    let proxy_owner = _proxy_owner().owner().unwrap();
    match signature {
        Some(signature) => {
            if let Identity::Address(proxy_owner_address) = proxy_owner {
                let nonce = increment_nonce();
                require(
                    verify_address(proxy_owner_address, signature, (nonce,
args)),
                    "Caller is not the proxy owner",
                );
                return Ok(proxy_owner);
            }
            return Err(SignatureError::InvalidSignature);
```

```
            },
            None => {
                require(
                    proxy_owner == msg_sender().unwrap(),
                    "Caller is not the proxy owner",
                );
            }
        };
        Ok(proxy_owner)
    }
```

Remediation:

We highly recommend to make the signature validation compliant with EIP-712, which uses a domain type hash to distinguish between chains and contracts, as well as a functional type hash to distinguish between the functions within the same contract.

FUEL10-11 | Rounding errors can be exploited profitably

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

packages/sway-v2/contracts/order-book/src/main.sw:match_with_existing_orders#L908-L1122

Description:

In the function **match_with_existing_orders**, it matches the newly created Taker order with any existing Maker orders. Depending on whether the Taker order is a Buy or a Sell, the amounts will be calculated and rounded differently.

For example, if the Taker order is a Buy order, it means they want to buy the base asset and have provided a quote asset and corresponding price. The Maker had previously created their order as a Sell order and provided the base asset and price.

However, due to a combination of the rounding, low amount of decimals for highly valued tokens and cheap gas fees, it could become possible for a Taker to steal amounts from a Maker Sell order.

Most tokens on Fuel Network have 9 decimals, but there are some with 6 or 8. For example, FBTC has 8 decimals and is a highly valued token, which means that even a very small amount has monetary value.

Currently, **match_with_existing_orders** uses the helper function **quote_amount_from_quantity** to calculate the require amount of quote assets for the amount of base assets at some price. It always rounds down.

The rounding down will lead to a loss of the Sell side, because the quote amount would be rounded down. This can be exploited in the case of a Maker Sell order by a Taker Buy order.

For example, consider the case where FBTC is the base asset and SolvBTC is the quote asset. The assets are roughly equal, but there might be a small price difference depending on things such as withdrawal mechanisms (similar to **ETH/stETH** markets).

If the price is at 0.**999999**, then the price for 1 share of FBTC would be rounded down as 0 shares of SolvBTC. The taker can then repeatedly creates small 1 share orders to slowly drain the Maker's order for free.

This can be impactful due to the low gas fees of Fuel, as well as the high price of a single share of FBTC. For example, at a price of **$100.000**, then **$1** would be equal to only **1000** shares.

```
fn quote_amount_from_quantity(quantity: u64, price: u64) -> u64 {
    let total = quantity.as_u256() * price.as_u256();
    (total / BASE_DECIMALS.as_u256()).try_into().unwrap()
}
```

Remediation:

In order to remediate this issue, a minimum order or fulfillment amount has to be enforced. It is insufficient to for example always round in favour of the Maker or the Sell order, because it can still be exploited in the different direction (e.g. a Taker can sell at an increased price of **2.0** if the price is **1.0001**).

By enforcing a minimum order or fulfillment amount, the rounding errors can evened out or make it not profitable for an attacker. An order could for example be closed if the remaining quantity is less than the amount.

Besides this, it could also be considered to always round in favour of the Maker anyway. This adds a thin layer of protection, as exploitation as a Taker is easier than as a Maker.

FUEL10-2 | Anyone Can Register Arbitrary Contract IDs for Other Users' Trade Accounts

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

packages/sway-v2/types/contracts/O2Register.ts

Description:

In **O2Register** users can register a trade account for an **owner** via **register_trade_account**. It enforces that each owner can only have one trade account, this function is callable by anyone without authorization checks.

Anyone can register a **trade-account** for another user, even if that **trade-account** doesn't belong to or relate to the owner. This allows misrepresentation of the owner's trade account.

```
#[storage(read, write)]
fn register_trade_account(contract_id: ContractId, owner: Identity) ->
Result<(), O2RegisterError> {
    // Check if owner already has a Trade Account
    if storage.owner_trade_account.get(owner).try_read().is_some() {
        return Err(O2RegisterError::OwnerAlreadyHasTradeAccount);
    }

    // Store the mapping
    storage.owner_trade_account.insert(owner, contract_id);

    // Create and store the registration record
    let registration = TradeAccountRegistered::new(contract_id, owner);
    storage.trade_accounts.push(registration);

    // Log the event
    let event = TradeAccountRegistered::new(contract_id, owner);
    event.log();

    Ok(())
}
```

Remediation:

Consider using **msg.sender** as owner for the registration via **register_trade_account**.

FUEL10-12 | Missing template-based bytecode-root verification in trade-account registration

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

packages/sway-v2/contracts/trade-account-registry/src/main.sw

Description:

The **register_contract** flow verifies a child proxy by recomputing a root from the child's own bytecode and comparing it to **bytecode_root(child)**. This proves internal consistency of the child artifact but does not validate it against the factory's expected template root per SRC-12. As a result, a proxy with altered logic (while still returning the expected **proxy_target()** and correct **ORACLE_CONTRACT_ID**) can be registered.

```
        // Verify the trade account proxy bytecode
        let returned_root = bytecode_root(child_contract);
        let computed_root = compute_bytecode_root(vec_bytecode, configurables);
        if computed_root != returned_root {
            return Err("Invalid Contract");
        };
```

The contract exposes **factory_bytecode_root()** but does not use a template-derived expected root during registration.

```
    #[storage(read)]
    fn factory_bytecode_root() -> Option<b256> {
        Some(TRADE_ACCOUNT_PROXY_ROOT)
    }
```

Registration may accept proxies that diverge from the canonical template. Existing checks (oracle ID and target alignment) reduce but do not remove this risk.

Remediation:

- On registration, compute **expected_root = compute_bytecode_root(template_bytecode, configurables)** and require **bytecode_root(child) == expected_root**
- If deployments are identical (no configurables), require **configurables.is_none()** and compare with **TRADE_ACCOUNT_PROXY_ROOT**

FUEL10-14 | Last traded price constraint could be exploited into DoS of the order book

Acknowledged

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

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

Description:

The **create_order** function checks the state variable **last_traded_price** against the order's **price**. If the different in percentage is greater than the **PRICE_WINDOW**, it reverts. The **last_traded_price** is updated with **price** of the last executed order.

This could be exploited by creating a lot of tiny 1 share orders at ever increasing or decreasing prices, such that no new orders at normal prices could be created anymore. In the case of decreasing it could make the price go to 1 share, which could never increase anymore, unless the **price_window** is at 100%.

As such, this constraint poses a potential vector for a DoS attack.

```
        // Validate the price collar
        match storage::orderbook::v1.last_traded_price.try_read() {
            Some((last_traded_price, _)) => {
                if PRICE_WINDOW > 0 {
                    require(
                        order.price
                            .as_u256() >= (last_traded_price
                                .as_u256() * (100 - PRICE_WINDOW)
                                .as_u256()) / 100
                            .as_u256() && order.price
                            .as_u256() <= (last_traded_price
                                .as_u256() * (100 + PRICE_WINDOW)
                                .as_u256()) / 100
                            .as_u256(),
                        OrderCreationError::PriceExceedsRange,
                    );
                }
            },
            None => {},
        }
```

Remediation:

We would recommend to remove the **last_traded_price** constraint, as the risk of manipulation is too high. Instead, we recommend to consider using a price oracle and checking the **order.price** to be in a range of the price oracle's provided price.

Commentary from the client:

"We recognize the possible DoS exploit and have opted to set the price window to zero in trading pairs with low liquidity where we may see no buy orders."

FUEL10-3 | Owner initialization could be front-run

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

packages/sway-v2/contracts/trade-account-oracle/src/main.sw#L106-121
packages/sway-v2/contracts/order-book/src/main.sw#L635-649
packages/sway-v2/contracts/register/src/main.sw#L152-155

Description:

Several contracts use the **initialize_ownership()** function in order to initialize the owner of the contract to enforce authentication on several critical functions. The way it is currently done is by having an initialize function that the owner would need to call after the contract creation however it leaves room for a potential attacker to front-run the initialization function and set themselves as the owner of the contract as is warned by the official Fuel Sway docs:

*Please note that the example above does not apply any restrictions on who may call the **initialize()** function. This leaves the opportunity for a bad actor to front-run your contract and claim ownership for themselves.*

```
#[storage(read, write)]
fn initialize(owner: Identity, trade_account_impl: ContractId) {
    initialize_ownership(owner);

    require(
        asm(blob_id: trade_account_impl, len) {
            bsiz len blob_id;
            len: u64
        } != 0,
        "Blob does not exist",
    );

    storage::trading_account_oracle::v1
        .trading_account_implementation
        .write(Some(trade_account_impl));
}
```

```
#[storage(read, write)]
fn initialize(owner: Identity) {
    initialize_ownership(owner);
    OrderBookConfigEvent::new(
        BASE_ASSET,
        QUOTE_ASSET,
        BASE_DECIMALS,
        QUOTE_DECIMALS,
        MIN_BASE_TRADE,
        MIN_QUOTE_TRADE,
        MAKER_FEE,
        TAKER_FEE,
    )
        .log();
}
```

```
#[storage(read, write)]
fn initialize(owner: Identity) {
    initialize_ownership(owner);
}
```

Remediation:

In order to protect from that attack the official Fuel Sway docs recommends the following:

*To ensure the intended **Identity** is set as the contract owner upon contract deployment, use a **configurable** where the **INITIAL_OWNER** is the intended owner of the contract.*

FUEL10-5 | Redundant match orders function

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

packages/sway-v2/contracts/order-book/src/main.sw#L251-L254

Description:

The function **order-book::match_orders()** is used to manually trigger the order matching logic. However, this function is unnecessary because matching is already executed automatically when creating a new order. Therefore, there is no scenario in which a buy order with a higher price than a sell order could occur.

```
/// This method may not be necessary if matching is performed automatically
/// during order creation. Consider removing if automatic matching is
implemented.
#[storage(read, write)]
fn match_orders() {
```

Remediation:

Consider removing the function if not necessary.

FUEL10-7 | Spot and Limit orders behave the same

Acknowledged

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

The ABI exposes an **OrderType** with **Limit** and **Spot**, but the matching path does not branch on the type. Orders are processed using the provided **order.price** as the taker bound, and any unfilled remainder is stored on the book regardless of type. As a result, "Spot" orders do not enforce immediate-only semantics or slippage-specific behavior; they effectively act like limit orders.

OrderType definition exists:

```
/// Defines the execution type for an order.
/// Determines how the order should be matched in the order book.
pub enum OrderType {
    /// Limit order - executes only at the specified price or better
    /// Buy limit orders execute at the limit price or lower
    /// Sell limit orders execute at the limit price or higher
    Limit: (Price, Time),
    /// Spot order - executes immediately at the best available price
    /// Takes liquidity from the order book until filled or exhausted
    Spot: (),
}
```

Unfilled remainder is stored irrespective of type:

```
        match match_with_existing_orders(
            order.price,
            side,
            StorageOrder::new(trader, order.quantity, order.order_type),
        ) {
            OrderStatus::Unfilled(unfilled) => {
                // Add to list
                let order_number = match side {
                    Side::Buy => {
                        storage::orderbook::v1.buy_map.set(order.price);

storage::orderbook::v1.buys.get(order.price).push_back(unfilled.0)
                    },
```

```
                    Side::Sell => {
                        storage::orderbook::v1.sell_map.set(order.price);

storage::orderbook::v1.sells.get(order.price).push_back(unfilled.0)
                    },
                };
```

Remediation:

Explicitly branch on **OrderType** in matching.

- Spot: ignore **taker_price** (buy: ∞, sell: 0) or use caller-provided slippage bound (**max_price**/**min_price**)
- Limit: keep current price-bounded behavior; place unfilled remainder on book

FUEL10-8 | Redundant pausability in Register contract is implemented but not enforced

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

The contract implements pausability but does not check paused state in state-changing functions. As a result, the pause mechanism has no effect on actual behavior.

Pausable is implemented:

```
impl Pausable for Contract {
    #[storage(write)]
    fn pause() {
        only_owner();
        _pause();
    }

    #[storage(write)]
    fn unpause() {
        only_owner();
        _unpause();
    }

    #[storage(read)]
    fn is_paused() -> bool {
        _is_paused()
    }
}
```

Write functions do not enforce pause:

```
fn register_trade_account(contract_id: ContractId, owner: Identity) ->
Result<(), O2RegisterError> {
    // no require_not_paused()
    ...
}

fn register_order_book(contract_id: ContractId, market_id: MarketId) ->
Result<(), O2RegisterError> {
    only_owner();
```

```
    // no require_not_paused()
    ...
}
```

Remediation:

- If pausability is desired: add **require_not_paused()**; to mutating functions (e.g., **register_trade_account**, **register_order_book**), leave read-only getters unguarded
- If writes are owner-only by design and no public writes are planned: remove pausability (drop Pausable from ABI/impl and delete **pause**, **unpause**, **is_paused**), and update docs/tests accordingly

FUEL10-13 | Potential revert in register contract violates SRC-12

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

packages/sway-v2/contracts/trade-account-registry/src/main.sw

Description:

SRC-12 requires register_contract must not revert and should return Result::Err(str) on failure. The current implementation in packages/sway-v2/contracts/trade-account-registry/src/main.sw uses unwrap (and may use implicit revert paths), which can panic under malformed inputs, violating the standard and reducing robustness.

```
fn register_contract(
        child_contract: ContractId,
        configurables: Option<Vec<(u64, Vec<u8>)>>,
    ) -> Result<BytecodeRoot, str> {
        // Ensure valid configurables
        if configurables.is_none()
            || configurables.unwrap().len() != 2
        {
            return Err("Invalid Configuables");
        }

        // Convert configurable data into oracle and trading account owner
        let oracle_as_vec = configurables.unwrap().get(0).unwrap().1;
        let owner_as_vec = configurables.unwrap().get(1).unwrap().1;
        let oracle_contract = asm(bytes: oracle_as_vec.ptr()) {
            bytes: ContractId
        };
        let owner = asm(bytes: owner_as_vec.ptr()) {
            bytes: State
        }.owner().unwrap();

        // Check if owner already has a Trade Account and verify the oracle is
correct
        if oracle_contract != ORACLE_CONTRACT_ID {
            return Err("Invalid Oracle Contract");
        } else if
storage::registry::v1.owner_trade_account.get(owner).try_read().is_some()
```

```
        {
            return Err("OwnerAlreadyHasTradeAccount");
        }

        // Check the trade account's proxy target
        let oracle_blob_id_target = abi(Oracle,
ORACLE_CONTRACT_ID.bits()).get_trade_account_impl();
        let proxy_blob_id_target = abi(TradeAccountProxy,
child_contract.bits()).proxy_target();
        if oracle_blob_id_target != proxy_blob_id_target {
            return Err("Invalid Proxy Target Set");
        }

        // Load the bytecode of the trade account proxy
        let contract_size = asm(load_target: child_contract, length) {
            csiz length load_target;
            length: u64
        };
        let loaded_bytecode_ptr = asm(load_target: child_contract, length:
contract_size, ptr) {
            aloc length;
            move ptr hp;
            ccp ptr load_target zero length;
            ptr: raw_ptr
        };
        let mut vec_bytecode: Vec<u8> =
Vec::from(raw_slice::from_parts::<u8>(loaded_bytecode_ptr, contract_size));

        // Verify the trade account proxy bytecode
        let returned_root = bytecode_root(child_contract);
        let computed_root = compute_bytecode_root(vec_bytecode, configurables);
        if computed_root != returned_root {
            return Err("Invalid Contract");
        };

        // Create and store the registration record
        let registration = TradeAccountRegistered::new(child_contract, owner);
        storage::registry::v1.trade_accounts.push(registration);
        storage::registry::v1
            .owner_trade_account
            .insert(owner, child_contract);
        storage::registry::v1
            .trade_account_contracts
            .insert(child_contract, owner);

        // Log the event
```

```
        let event = TradeAccountRegistered::new(child_contract, owner);
        event.log();
        Ok(computed_root)
    }
```

Remediation:

- Replace all **unwrap**/revert paths with explicit checks returning **Err(str)**
- Validate **configurables** shape and **owner()** presence before use
- Ensure every failure path returns **Result::Err(str)**; no **require** in this function


hexens
x
FUEL