Audit Report

Fuzion Plasma OTC

v1.1

February 24, 2023

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Fuzion to perform a security audit of Fuzion Plasma OTC.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following GitHub repository:

https://github.com/ATLO-Labs/fuzion-otc/

Commit hash: `8fe7542a1d20ab7f568cee0e6031b9d5eae1dd13`

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base’s intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line by line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

Fuzion Plasma OTC is a protocol built on the Kujira chain that enables users to execute OTC coin swaps. It allows both public and private OTC coin swaps with different options, such as the presence of an arbiter, the creation of vesting accounts, and partial swaps.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending, Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	**Low-Medium**	-
Code readability and clarity	**Medium-High**	-
Level of documentation	**Medium**	No technical documentation was provided.
Test coverage	**High**	95.7% reported test coverage.

Summary of Findings

No	Description	Severity	Status
1	Attackers can drain funds from contract	Critical	Resolved
2	Malicious arbiters can steal funds by overwriting the recipient address	Critical	Resolved
3	Funds are locked if the recipient has deposited and the arbiter does not approve the escrow before the deadline	Critical	Resolved
4	Escrows can be completed without recipient's interaction	Major	Resolved
5	Escrow and vesting end times should be enforced to be in the future	Minor	Resolved
6	Division by zero error if creator or receiver fee percentage is set to zero	Minor	Resolved
7	Missing address validation during contract instantiation and configuration update	Minor	Resolved
8	Fee collectors' percentage sum should not exceed 100	Minor	Resolved
9	Contract-defined admins can lock user funds	Minor	Resolved
10	`OTC_ACTIVE_PAIRS_COUNT` is not decrement when public `OTC` is modified into private	Minor	Resolved
11	Missing validation to ensure that `creator_balance` is not equal to `asking_price`	Informational	Resolved
12	Admin can update `fee_collectors` to an empty vector	Informational	Resolved
13	Redundant partial fill validation	Informational	Acknowledged
14	Incorrect comment for `vesting_end_time`	Informational	Resolved
15	Loss of escrow information after the execution of `Refund` messages	Informational	Acknowledged
16	Custom access controls implementation	Informational	Acknowledged
17	“Migrate only if newer” pattern is not followed	Informational	Resolved

18	Overflow checks are not enabled	**Informational**	**Resolved**

Detailed Findings

1. Attackers can drain funds from contract

Severity: Critical

In `src/commands.rs:245`, the `execute_set_recipient` function sets the escrow status to `EscrowStatus::Active` even if the escrow is already completed and its funds were transferred.

This allows an attacker to reactivate completed escrows to drain funds from the contract by executing `SetRecipient` and `Approve` messages repeatedly.

Additionally, since in `src/commands.rs:610` the function `send_tokens` returns an empty vector when the balance is empty, even if the receiver did not deposit funds, the sender can still perform the attack draining funds from the contract.

Please see the `steal_funds` test case to reproduce the issue.

Recommendation

We recommend ensuring that the escrow status is `EscrowStatus::Pending` when executing the `SetRecipient` message and returning an error if the balance is not positive, similarly to how Cosmos SDK does it in the `Bank` module.

Status: Resolved

2. Malicious arbiters can steal funds by overwriting the recipient address

Severity: Critical

In `src/commands.rs:408-415`, an arbiter needs to approve the escrow before processing the trade.

A malicious arbiter can execute the `SetRecipient` message to overwrite the recipient address to another one under its control and then call the `Approve` message to steal the original recipient’s funds.

Please see the `arbiter_steal_funds` test case to reproduce the issue.

Recommendation

We recommend ensuring the escrow's `recipient_balance` vector is not empty when executing the `SetRecipient` message.

Status: Resolved

3. Funds are locked if the recipient has deposited and the arbiter does not approve the escrow before the deadline

Severity: Critical

In `src/commands.rs:408-415`, the arbiter is required to approve the escrow to process the trade if it is created providing an arbiter and recipient.

If the arbiter does not approve the escrow before the deadline, funds will be stuck in the contract. In fact, both `Approve` and `Refund` messages will revert respectively with `Expired` and `RecipientAlreadyDeposited` contract errors.

Due to the impossibility of the arbiter rejecting an escrow and refunding both parties, this issue is likely to happen when the arbiter opposes the trade.

Please see the `funds_stuck_arbiter_after_deadline` test case to reproduce the issue.

Recommendation

We recommend implementing functionality for the arbiter to refund funds to both parties, even after the deadline.

Status: Resolved

4. Escrows can be completed without recipient's interaction

Severity: Major

In `src/commands.rs:255`, the `execute_approve` function does not verify the recipient's balance is not empty when processing the trade, allowing the trade to be completed without any deposits from the recipient.

This is problematic because the recipient could collude with the arbiter to approve the escrow and steal the creator's funds. Consequently, the creator's funds will be sent to the recipient, while the creator receives nothing in return.

Please see the `approve_without_deposit` test case to reproduce this issue.

Recommendation

We recommend ensuring the escrow's `recipient_balance` vector is not empty when executing the `Approve` message.

Status: Resolved

5. Escrow and vesting end times should be enforced to be in the future

Severity: Minor

During the execution of `Create` messages, in both `src/commands.rs:126-129` and `src/commands.rs:196`, the `vesting_end_time` and `end_time` timestamps are not validated to be in the future.

This could lead to scenarios where the newly created escrow is already expired, or the vested tokens are immediately redeemable.

Recommendation

We recommend validating `vesting_end_time` and `end_time` timestamps to be in the future.

Status: Resolved

6. Division by zero error if creator or receiver fee percentage is set to zero

Severity: Minor

During contract instantiation, zero percentage fees for creator or receiver are accepted.

However, since this value is used as a denominator in the `calc_minimum_balance` function in `src/commands:696` during the handling of `Create` messages, a zero value will lead to a division by zero error.

Additionally, if any fee collectors are misconfigured to receive zero fees, the `transfer_tokens_messages` function will fail because the contract will try to send zero amount of fees, which will be prevented by Cosmos SDK.

Recommendation

We recommend modifying the `assert_valid_percentage` function to reject zero percentage amounts.

Status: Resolved

7. Missing address validation during contract instantiation and configuration update

Severity: Minor

During contract instantiation in `src/contract.rs:28-54` as well as execution of `UpdateConfig` messages in `src/commands.rs:16-51`, the `fee_collectors` and `admins` addresses are not validated.

Storing invalid addresses could lead to unexpected behavior like errors when completing escrows because of the failure of `Bank` messages directed to an invalid fee collector address.

Recommendation

We recommend validating addresses before storing them.

Status: Resolved

8. Fee collectors' percentage sum should not exceed 100

Severity: Minor

In both `src/contract.rs:40-42` and `src/commands.rs:35-40`, the `fee_collectors` vector provided by the admin is set in the `CONFIG` struct.

Since this vector represents a weighted list of addresses that should receive fees, the sum of the percentages of all the vector's elements should not exceed 100 to avoid charging more fees than expected.

Additionally, duplicate addresses are allowed in the vector, potentially leading to higher fees than intended.

We classify this issue as minor since only the contract admin can cause it.

Recommendation

We recommend validating the `fee_collectors` vector to ensure that the sum of the elements' percentages does not exceed 100 and deduping the fee collector addresses to prevent any address from collecting fees more than once.

Status: Resolved

9. Contract-defined admins can lock user funds

Severity: Minor

In `src/commands.rs:655-691`, the `calc_fees` function iterates over all fee collectors for each `balance` coin in order to send fees through `Bank` messages.

Since no upper limit for `fee_collectors` is enforced, admins could store a large `fee_collectors` vector in order to let the contract run out of gas during the `calc_fees` function execution.

Consequently, all the messages that involve the sending of funds will fail, leading to the lock of user funds in the contract.

We classify this issue as minor since it can only be caused by admins who are community-trusted entities.

Recommendation

We recommend defining and enforcing an upper limit to the number of entries of the `fee_collector` vector.

Status: Resolved

10. `OTC_ACTIVE_PAIRS_COUNT` is not decrement when public `OTC` is modified into private

Severity: Minor

In `src/commands.rs:230`, the `execute_set_recipient` function does not decrease the active `OTC` pairs when a public `OTC` trade becomes private.

Consequently, this causes the `MarketEscrowActivePairsCount` query to return incorrect values.

Please see the `public_to_private_does_not_decrease_active_otc` test case to reproduce the issue.

Recommendation

We recommend executing the `active_otc_pair_decrement` function if a public `OTC` trade is updated to a private trade.

Status: Resolved

11. Missing validation to ensure that `creator_balance` is not equal to `asking_price`

Severity: Informational

In `src/commands.rs:96`, the function `execute_create` is not checking if the `creator_balance` is equal to the `asking_price`.

If an escrow with the same provided and asked coin's denom and amount is created, this could generate a potentially unwanted arbitrage opportunity and make the creator lose funds on the trade.

Recommendation

We recommend adding validation to ensure that both the provided and asked coins are not equal.

Status: Resolved

12. Admin can update `fee_collectors` to an empty vector

Severity: Informational

In `src/commands.rs:35-40`, during an update of `fee_collectors`, `assert_has_fee_collectors` is not called to ensure the fee collectors vector is not empty.

This is inconsistent with the contract instantiation phase in `src/contract.rs:36`.

Recommendation

We recommend executing the `assert_has_fee_collectors` function during the `execute_update_config` phase.

Status: Resolved

13. Redundant partial fill validation

Severity: Informational

In `src/commands.rs:461-463`, the `if` statement ensures the escrow allows partial fills in the `execute_receiver_partial_deposit` function. This check can be removed as the same validation is performed in `src/commands.rs:333`.

Recommendation

We recommend removing the redundant validation to increase the efficiency of the code.

Status: Acknowledged

14. Incorrect comment for `vesting_end_time`

Severity: Informational

In `src/commands.rs:90`, the comment documents that `vesting_end_time` represents the vesting start time. This is incorrect, as the variable represents the vesting's end time.

Recommendation

We recommend modifying the comment into "*Vesting end time*".

Status: Resolved

15. Loss of escrow information after the execution of `Refund` messages

Severity: Informational

During the handling of `Refund` messages, the `execute_refund` function removes the selected `escrow` data from the storage.

While this is not a security issue, it could degrade the user experience since `escrow` information is not queriable anymore, and all the involved parties have no more reference to their operations.

Additionally, in the case of partially filled escrows, the removal of the original escrow could lead to the impossibility of retrieving data from its `partial_fill_of_id`.

Recommendation

We recommend implementing a final state for refunded `escrows` instead of removing them.

Status: Acknowledged

16. Custom access controls implementation

Severity: Informational

The contract implements custom access controls. Although no instances of broken controls or bypasses have been found, using a battle-tested implementation reduces potential risks and the complexity of the codebase.

Also, the access control logic is duplicated across the handlers of each function, which negatively impacts the code's readability and maintainability.

Recommendation

We recommend using a well-known access control implementation such as `cw_controllers::Admin` (https://docs.rs/cw-controllers/0.14.0/cw_controllers/struct.Admin.html).

Status: Acknowledged

17. “Migrate only if newer” pattern is not followed

Severity: Informational

The contract is currently migrated without regard to the versioning. This can be improved by adding validation to ensure that the migration is only performed if the supplied version is newer.

Recommendation

It is recommended to follow the migrate “only if newer” pattern defined in the CosmWasm documentation.

Status: Resolved

18. Overflow checks are not enabled

Severity: Informational

The contracts crate does not have `overflow-checks` enabled in `Cargo.toml`.

This profile configuration is useful to enforce control on possible overflows in the contract code.

Recommendation

We recommend enabling `overflow-checks` in `Cargo.toml`.

Status: Resolved

Appendix: Test Cases

1. Test case for "[Attackers can drain funds from contract]"

```rust
#[test]
fn steal_funds() {
    let mut deps = mock_dependencies();
    let attacker = String::from("attacker");

    // init contract
    let instantiate_msg = default_instantiate_msg();
    let info = mock_info(&attacker, &[]);
    instantiate(deps.as_mut(), mock_env(), info, instantiate_msg).unwrap();
    let asking_price = coin(1000, "earth");

    let create = CreateMsg {
        arbiter: None,
        recipient: Some(attacker.clone()),
        title: None,
        end_time: None,
        description: None,
        asking_price: asking_price.clone(),
        vesting_account: None,
        vesting_end_time: None,
        vesting_delayed: None,
        partial_fills_allowed: None,
        partial_fill_minimum: None,
        partial_fill_maximum: None,
    };

    // create an escrow
    let creator_balance = coins(1000, "tokens");
    let info = mock_info(&attacker, &creator_balance);
    let env = mock_env();
    let msg = ExecuteMsg::Create(create.clone());
    execute(deps.as_mut(), env.clone(), info, msg).unwrap();

    // receiver deposit funds into escrow
    // NOTE: In order to trigger the attack that is enabled
    // by not returning an error on the send_tokens function, this
    // step can be ignored.
    let info = mock_info(&attacker, &[asking_price.clone()]);
    let env = mock_env();
    let msg = ExecuteMsg::ReceiverDeposit {
        id: 1,
        vesting_account: None,
    };
    execute(deps.as_mut(), env.clone(), info.clone(), msg).unwrap();
```

```
    // set recipient
    let msg = ExecuteMsg::SetRecipient {
        id: 1,
        recipient: attacker.clone(),
    };
    execute(deps.as_mut(), mock_env(), info.clone(), msg).unwrap();

    // approve escrow
    execute(deps.as_mut(), mock_env(), info.clone(), ExecuteMsg::Approve { id: 1
}).unwrap();

    // repeat to steal funds
    let msg = ExecuteMsg::SetRecipient {
        id: 1,
        recipient: attacker,
    };
    execute(deps.as_mut(), mock_env(), info.clone(), msg).unwrap();
    execute(deps.as_mut(), mock_env(), info.clone(), ExecuteMsg::Approve { id: 1
}).unwrap();

}
```

2. Test case for "[Malicious arbiters can steal funds by overwriting the recipient address]"

```rust
#[test]
fn arbiter_steal_funds() {
    let mut deps = mock_dependencies();

    let creator = String::from("creator");
    let arbiter = String::from("arbiter");
    let recipient = String::from("recipient");

    // init contract
    let instantiate_msg = default_instantiate_msg();
    let info = mock_info(&creator, &[]);
    instantiate(deps.as_mut(), mock_env(), info, instantiate_msg).unwrap();

    let asking_price = coin(1_000, "earth");

    let create = CreateMsg {
        arbiter: Some(arbiter.clone()),
        recipient: Some(recipient.clone()),
        title: None,
        end_time: None,
        description: None,
        asking_price: asking_price.clone(),
        vesting_account: None,
        vesting_end_time: None,
        vesting_delayed: None,
        partial_fills_allowed: None,
        partial_fill_minimum: None,
        partial_fill_maximum: None,
    };

    // create an escrow
    let creator_balance = coins(1_000, "tokens");
    let info = mock_info(&creator, &creator_balance);
    let msg = ExecuteMsg::Create(create.clone());
    execute(deps.as_mut(), mock_env(), info, msg).unwrap();

    // recipient deposit funds
    let info = mock_info(&recipient, &[asking_price.clone()]);
    let msg = ExecuteMsg::ReceiverDeposit { id: 1, vesting_account: None };
    execute(deps.as_mut(), mock_env(), info.clone(), msg.clone()).unwrap();

    // escrow pending to be approved by arbiter
    let res = query_escrow(deps.as_ref(), &mock_env(), 1).unwrap();
    assert_ne!(res.status, EscrowStatus::Completed);
    assert_eq!(res.recipient.unwrap(), recipient);

    // instead of approving, the arbiter first overwrites the recipient to
```

```
themselves
    let info = mock_info(&arbiter, &[]);
    let msg = ExecuteMsg::SetRecipient { id: 1, recipient: arbiter.clone()};
    execute(deps.as_mut(), mock_env(), info.clone(), msg).unwrap();

    // arbiter then approve the funds
    let info = mock_info(&arbiter, &[]);
    let msg = ExecuteMsg::Approve { id: 1 };
    execute(deps.as_mut(), mock_env(), info.clone(), msg).unwrap();

    // recipient gets forcefully modified into arbiter
    let res = query_escrow(deps.as_ref(), &mock_env(), 1).unwrap();
    assert_eq!(res.status, EscrowStatus::Completed);
    assert_eq!(res.recipient.unwrap(), arbiter);
}
```

3. Test case for "Funds are locked if the recipient has deposited and the arbiter does not approve the escrow before the deadline"

```rust
#[test]
fn funds_stuck_arbiter_after_deadline() {
    let mut deps = mock_dependencies();

    let creator = String::from("creator");
    let arbiter = String::from("arbiter");
    let recipient = String::from("recipient");

    // init contract
    let instantiate_msg = default_instantiate_msg();
    let info = mock_info(&creator, &[]);
    instantiate(deps.as_mut(), mock_env(), info, instantiate_msg).unwrap();

    let asking_price = coin(1_000, "earth");
    let mut test_env = mock_env();
    test_env.block.time = Timestamp::from_seconds(101);

    let create = CreateMsg {
        arbiter: Some(arbiter.clone()),
        recipient: Some(recipient.clone()),
        title: None,
        end_time: Some(102),
        description: None,
        asking_price: asking_price.clone(),
        vesting_account: None,
        vesting_end_time: None,
        vesting_delayed: None,
        partial_fills_allowed: None,
        partial_fill_minimum: None,
        partial_fill_maximum: None,
    };

    // create an escrow
    let creator_balance = coins(1_000, "tokens");
    let info = mock_info(&creator, &creator_balance);
    let msg = ExecuteMsg::Create(create.clone());
    execute(deps.as_mut(), test_env.clone(), info, msg).unwrap();

    // recipient deposit funds
    let info = mock_info(&recipient, &[asking_price.clone()]);
    let msg = ExecuteMsg::ReceiverDeposit { id: 1, vesting_account: None };
    execute(deps.as_mut(), test_env.clone(), info.clone(),
msg.clone()).unwrap();

    let res = query_escrow(deps.as_ref(), &test_env.clone(), 1).unwrap();

    // verify recipient deposited
```

```
    assert_eq!(res.recipient_balance, vec![asking_price.clone()]);

    // escrow haven't end because waiting arbiter
    assert_ne!(res.status, EscrowStatus::Completed);
    assert_ne!(res.status, EscrowStatus::Expired);

    // arbiter did not Approve escrow
    // escrow expired
    test_env.block.time = Timestamp::from_seconds(103);

    // verify escrow expired
    let res = query_escrow(deps.as_ref(), &test_env.clone(), 1).unwrap();
    assert_eq!(res.status, EscrowStatus::Expired);

    // funds are now locked for both creator and recipient
    // even if the arbiter comes in to refund, the tx will fail
    let info = mock_info(&arbiter, &[]);
    let msg = ExecuteMsg::Refund { id: 1 };
    let res = execute(deps.as_mut(), test_env.clone(), info.clone(),
msg).unwrap_err();
    assert_eq!(res, ContractError::RecipientAlreadyDeposited {});

    // approving fails too
    let msg = ExecuteMsg::Approve { id: 1 };
    let res = execute(deps.as_mut(), test_env.clone(), info.clone(),
msg).unwrap_err();
    assert_eq!(res, ContractError::Expired {});

}
```

4. Test case for "Escrows can be completed without recipient's interaction"

```rust
#[test]
fn arbiter_collude_with_recipient() {
    let mut deps = mock_dependencies();

    let user = String::from("user");
    let arbiter = String::from("arbiter");
    let recipient = String::from("recipient");

    // init contract
    let instantiate_msg = default_instantiate_msg();
    let info = mock_info(&user, &[]);
    instantiate(deps.as_mut(), mock_env(), info, instantiate_msg).unwrap();

    let create = CreateMsg {
        arbiter: Some(arbiter.clone()),
        recipient: Some(recipient.clone()),
        title: None,
        end_time: None,
        description: None,
        asking_price: coin(1000, "earth"),
        vesting_account: None,
        vesting_end_time: None,
        vesting_delayed: None,
        partial_fills_allowed: None,
        partial_fill_minimum: None,
        partial_fill_maximum: None,
    };

    // creator creates an escrow
    let creator_balance = coins(1000, "tokens");
    let info = mock_info(&user, &creator_balance);
    let msg = ExecuteMsg::Create(create.clone());
    execute(deps.as_mut(), mock_env(), info, msg).unwrap();

    // arbiter approves the escrow even though receipient never deposited
    let info_arbiter = mock_info(&arbiter, &[]);
    let msg = ExecuteMsg::Approve { id: 1 };
    let res = execute(deps.as_mut(), mock_env(), info_arbiter.clone(),
msg.clone()).unwrap();

    // creator's funds will be sent to recipient
    assert_eq!(
        res.messages,
        vec![
            SubMsg {
                id: 0,
                msg: CosmosMsg::Bank(BankMsg::Send {
```

```
                    to_address: recipient,
                    amount: vec![Coin {
                        denom: "tokens".to_string(),
                        amount: Uint128::new(990),
                    }],
                }),
                gas_limit: None,
                reply_on: cosmwasm_std::ReplyOn::Never,
            },
            SubMsg {
                id: 0,
                msg: CosmosMsg::Bank(BankMsg::Send {
                    to_address: "collector1".to_string(),
                    amount: vec![Coin {
                        denom: "tokens".to_string(),
                        amount: Uint128::new(5),
                    }],
                }),
                gas_limit: None,
                reply_on: cosmwasm_std::ReplyOn::Never,
            },
            SubMsg {
                id: 0,
                msg: CosmosMsg::Bank(BankMsg::Send {
                    to_address: "collector2".to_string(),
                    amount: vec![Coin {
                        denom: "tokens".to_string(),
                        amount: Uint128::new(5),
                    }],
                }),
                gas_limit: None,
                reply_on: cosmwasm_std::ReplyOn::Never,
            }
        ]
    )

}
```

5. **Test case for “OTC_ACTIVE_PAIRS_COUNT is not decrement when public OTC is modified into private”**

```
#[test]
fn public_to_private_does_not_decrease_active_otc() {
    let mut deps = mock_dependencies();

    let creator = String::from("creator");
    // let arbiter = String::from("arbiter");
    let recipient = String::from("recipient");

    // init contract
    let instantiate_msg = default_instantiate_msg();
    let info = mock_info(&creator, &[]);
    instantiate(deps.as_mut(), mock_env(), info, instantiate_msg).unwrap();

    let asking_price = coin(1_000, "earth");

    let create = CreateMsg {
        arbiter: None,
        recipient: None,
        title: None,
        end_time: None,
        description: None,
        asking_price: asking_price.clone(),
        vesting_account: None,
        vesting_end_time: None,
        vesting_delayed: None,
        partial_fills_allowed: None,
        partial_fill_minimum: None,
        partial_fill_maximum: None,
    };

    // create an escrow
    let creator_balance = coins(1_000, "tokens");
    let info = mock_info(&creator, &creator_balance);
    let msg = ExecuteMsg::Create(create.clone());
    execute(deps.as_mut(), mock_env(), info.clone(), msg).unwrap();

    let res = query_escrow(deps.as_ref(), &mock_env(), 1).unwrap();
    assert_eq!(res.arbiter, None);
    assert_eq!(res.recipient, None);

    let otc_active_pair_count =
query_market_otc_active_pairs_count(deps.as_ref()).unwrap();
    assert_eq!(otc_active_pair_count.pairs.len(), 1);

    // creator sets recipient
    let msg = ExecuteMsg::SetRecipient { id: 1, recipient: recipient.clone() };
    execute(deps.as_mut(), mock_env(), info.clone(), msg).unwrap();
```

```
    // recipient is updated
    let res = query_escrow(deps.as_ref(), &mock_env(), 1).unwrap();
    assert_eq!(res.arbiter, None);
    assert_eq!(res.recipient, Some(Addr::unchecked(recipient.clone())));

    // public otc becomes private, however query still shows active
    let otc_active_pair_count =
query_market_otc_active_pairs_count(deps.as_ref()).unwrap();
    assert_ne!(otc_active_pair_count.pairs.len(), 0);

    // complete the escrow
    let info = mock_info(&recipient, &[asking_price.clone()]);
    let msg = ExecuteMsg::ReceiverDeposit { id: 1, vesting_account: None };
    execute(deps.as_mut(), mock_env(), info.clone(), msg.clone()).unwrap();

    // verify escrow completed
    let res = query_escrow(deps.as_ref(), &mock_env(), 1).unwrap();
    assert_eq!(res.status, EscrowStatus::Completed);

    // otc query still shows 1
    let otc_active_pair_count =
query_market_otc_active_pairs_count(deps.as_ref()).unwrap();
    assert_eq!(otc_active_pair_count.pairs.len(), 1);

}
```