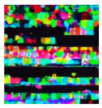


ZKSECURITY

Audit of Renegade's Atomic Settlement Feature

October 21, 2024

Introduction

On October 21, 2024, zkSecurity was tasked with auditing Renegade's atomic settlement feature. The specific code to review was shared via public GitHub repositories. The audit lasted five days with one consultant.

Written specifications detailing the circuits, the smart contracts of the atomic settlement feature were also shared.

The code was well commented and was found to be of great quality. The Renegade team was very responsive and helpful in providing additional context.

Scope

The scope of the audit included the circuits and contracts of the atomic settlement feature. zkSecurity used the `zksecurity-audit-10-21-24` branch for `renegade` (commit `1934f69c6cbb290f7536809ff63bb6068ba1028b`) and `renegade-contracts` (commit `f7b7dced3fae359185690dff7e63c356b7d6effb`).

Renegade circuit:

1. The circuit constraints for the new NP statement: `VALID MATCH SETTLE ATOMIC` (`zk_circuits/valid_match_settle_atomic.rs`).
2. The logic to link `VALID MATCH SETTLE ATOMIC` proof to the other proofs of `VALID COMMITMENTS` (`zk_circuits/proof_linking.rs#L274-L363`).

Renedage contracts:

1. Darkpool Entrypoint (`contracts/darkpool.rs#L542-L571`) for `process_atomic_match_settle`.
2. The core settlement contract (`contracts/core/core_settlement.rs`): responsible for state updates and routing to other smart contracts for settlement specific calls.
3. The new transfer logic (`contracts/transfer_executor.rs#L159`): responsible for executing direct ERC20 `transfer` and `transferFrom` calls.
4. The settlement verifier contract (`contracts/verifier/verifier_settlement.rs#L28`): responsible for setting up batch verification and proof-linking elements before passing onto the verifier core.

Overview of Renegade's Atomic Settlement Feature

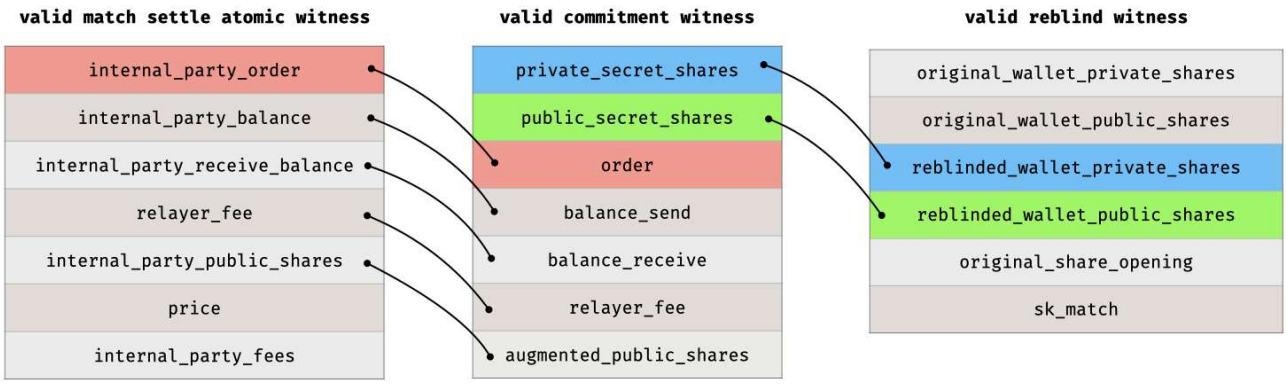
Renegade is a dark pool trading platform that allows users to trade ERC-20 assets without revealing any information about their trades and balances. Previously, a user needs to deposit, place an order, match, and withdraw to perform a trade, which was non-atomic. The atomic settlement feature enables an external party to swap with the dark pool in one transaction, without the need to deposit in advance. This can be helpful for certain integrators, such as CoW Swap solvers.

This is the high level flow of atomic settlement:

1. The external party queries the `/request-external-match` endpoint on the relayer with their intended trade intention.
2. The relayer queries its internal pool to find any matches. If a match is found, the relayer returns a ZKP proof for `VALID MATCH SETTLE ATOMIC`, which includes the matching result (e.g., match amount and price).
3. The external party receives the proof and statement. If it meets the requirements, the external party calls the contract to perform the swap. The contract then verifies the proof, transfers in the sell token, and transfers out the buy token in a single transaction.

The atomic match circuit functions similarly to the previous internal match circuit. The difference is that the atomic match circuit only verifies the state of one internal party, leaving the external party's token transfer for the contract to execute.

The circuit ensures that the `ExternalMatchResult` in the statement is valid. To confirm a valid wallet state transition, the proof is linked with a valid commitment witness, which in turn is linked to a valid reblind witness to ensure the wallet is reblinded from an existing wallet.



In the contract, the proof is first verified to ensure a valid internal transition. Then, the sell token is transferred from the external party into the pool, and the buy token is transferred to the external party.

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
00	renegade-contracts/core_settlement.rs	Transfer Amount Underflow Can Lead to Loss of Funds	High
01	renegade-contracts/darkpool.rs	A Wallet Can Be Temporarily Blocked by Duplicated Blinder	Medium
02	circuits/zk_gadgets/fixed_point.rs	Decimal Ceiling Can Lead to Micro-Amount Theft	Medium
03	renegade-contracts/*	No Restriction on ERC-20 Token Contracts Allows for Reentrant Execution	Low

00 - Transfer Amount Underflow Can Lead to Loss of Funds

● renegade-contracts/core_settlement.rs

High

Description. In Renegade's atomic settlement, the external party deposits tokens, exchanges, and withdraws tokens in one transaction. The withdrawal involves three ERC20 transfers: withdrawing the tokens bought by the receiver, paying the protocol fee, and paying the relayer fee. The protocol fee and relayer fee are deducted from the tokens bought by the receiver. However, due to an oversight, the relayer fee is not properly restricted in the circuit and can be larger than the amount bought by the receiver. As a result, an underflow may occur when deducting the fees.

```
        // The amount received by the external party after deducting the fees
        let trader_take = receive_amount - fees.total();
        transfers_batch.push(SimpleErc20Transfer::new_withdraw(
            tx_sender,
            receive_mint,
            trader_take,
        ));
```

In the code above, **`receive_amount`** and the fees are restricted to a 100-bit number. If an underflow occurs, it will result in **`trader_take`** being a very large value, close to **`U256::MAX`**. An attack is only possible when the total deposit of the protocol is close to **`U256::MAX`**.

At first glance, such an attack seems impossible: The protocol always withdraws the total amount of **`trader_take + protocol_fee + relayer_fee`**. If an underflow occurs, the total withdrawal amount would exceed **`U256::MAX`**, which is the maximum possible amount of ERC20 tokens. Thus, the attack seems doomed to fail.

However, the above statement is not true. Such an attack can occur. The attacker can set their relayer fee address as the protocol contract address. In this case, the **`relayer_fee`** will be transferred to the protocol itself. The total withdrawal amount will be **`trader_take + protocol_fee`**, which will not exceed **`U256::MAX`**.

Recommendations. It is recommended to use safe math in the contract when dealing with token arithmetic. For Rust, we can use **`checked_sub`** and **`checked_add`**:

```
        // The amount received by the external party after deducting the fees
        let trader_take = receive_amount.checked_sub(fees.total());
        transfers_batch.push(SimpleErc20Transfer::new_withdraw(
            tx_sender,
            receive_mint,
            trader_take,
        ));
```

If there is an upper bound ratio (e.g., 1%) for the relayer fee for the external party, it's recommended to restrict it directly in the circuit. This also reduces the risk to the external party in case they forget to check the relayer fee.

01 - A Wallet Can Be Temporarily Blocked by Duplicated Blinder

● renegade-contracts/darkpool.rs

Medium

Description. Renegade's off-chain clients use the public blinder to index the wallet. Thus, Renegade disallows duplicated public blinder to avoid seeing conflicting wallets. Every time in wallet create, update and reblind operation, the contract checks the new wallet's public blinder doesn't exist, and then insert it into the **`public_blinder_set`**.

```
/// Marks the given public blinder as used
pub fn mark_public_blinder_used<C: CoreContractStorage, S: TopLevelStorage + BorrowMut<C>>(
    s: &mut S,
    blinder: ScalarField,
) -> Result<(), Vec<u8>> {
    // First check that the blinder hasn't been used
    let this = s.borrow_mut();
    let blinder = scalar_to_u256(blinder);
    assert_result!(
        !this.public_blinder_set().get(blinder),
        PUBLIC_BLINDER_USED_ERROR_MESSAGE
    )?;

    // Mark the blinder as used
    this.public_blinder_set_mut().insert(blinder, true);
    Ok(())
}
```

In wallet reblind operation, the blinder of new wallet is deterministically derived from the old wallet: **`[new_blinder, new_blinder_private] = CSPRNG(prev_private_wallet.blinder)`**. However, there is no restriction on blinder when creating or updating wallet. Thus, a wallet will be blocked if someone creates the same public blinder (in wallet create or wallet update operation) in advance and inserts it into **`public_blinder_set`**. In this case the wallet cannot directly perform the wallet reblind operation again, which is the key step of order matching.

The relayer knows the private blinder of the wallet so it can perform the attack. An external attacker can also perform the attack: it can listen to the transaction mempool, get the new public blinder from unconfirmed transactions, and then frontrun the transaction to create a wallet with the same public blinder. With the atomic settlement feature, the **`ValidOfflineFeeSettlementStatement`** (which contains the new public blinder) will be sent to the external party before the transaction. Then it will be easier for the external party to perform the attack.

Note that if the attack happens, the wallet can change the blinder without restriction with a wallet update operation. This ensures that the wallet will not be blocked permanently.

Recommendations. It is recommended to add restriction on the blinder in wallet create and wallet update operation. For example, witness the `blinder_seed` and ensure `[blinder, blinder_private] = hash("new_blinder_padding", blinder_seed)` when creating or updating wallet. This ensures that the new created blinder will never conflict with the existing blinder.

02 - Decimal Ceiling Can Lead to Micro-Amount Theft

● circuits/zk_gadgets/fixed_point.rs

Medium

Description. Renegade uses the **`constrain_equal_integer_ignore_fraction`** gadget to perform decimal rounding. According to the specifications, the rounding should be toward the floor. However, in the code, the gadget is loosely constrained and allows ceiling rounding, which can lead to micro-amount theft.

The rouding gadget is implemented as:

```
    /// Constrain a fixed point variable to be equal to the given integer
    /// when ignoring the fractional part
    pub fn constrain_equal_integer_ignore_fraction(
        lhs: FixedPointVar, // the precise decimal number
        rhs: Variable, // the rounding result
        cs: &mut PlonkCircuit,
    ) -> Result<(), CircuitError> {
        // Shift the integer and take the difference
        let zero_var = cs.zero();
        let one_var = cs.one();
        let one = ScalarField::one();

        let lhs_minus_rhs =
            cs.lc(&[lhs.repr, rhs, zero_var, zero_var], &[one, -*TWO_TO_M_SCALAR, one, one])?;

        // Constrain the difference to be less than the precision on the fixed point,
        // This is effectively the same as constraining the difference to have an
        // integral component of zero
        // (2^m - 1) - (lhs - rhs) > 0
        let diff = cs.lc(
            &[one_var, lhs_minus_rhs, zero_var, zero_var],
            &[(*TWO_TO_M_SCALAR - ScalarField::one()), -one, one, one],
        )?;
        GreaterThanEqZeroGadget::<{ DEFAULT_FP_PRECISION + 1 }>::constrain_greater_than_eq_zero(
            diff, cs,
        )
    }
```

This is actually checking **`0 <= (2^m - 1) - (lhs - rhs) <= 2^{m+1} - 1`**, which translates to **`-2^m <= (lhs - rhs) <= (2^m - 1)`**. This allows for rounding to either floor or ceiling, which can introduce vulnerabilities under certain circumstances.

Renegade uses decimal rounding in two places: calculating the fee and the matching amount.

For the fee calculation, this issue enables the relayer to obtain a larger `relayer_fee` by simply rounding up the `relayer_fee`. It may also result in `relayer_fee + protocol_fee` exceeding `received_amount`. Take the following code as an example: suppose `received_amount` is `1`, then `expected_relayer_fee` and `expected_protocol_fee` will be small fractions. However, due to ceiling rounding, the `relayer_fee` and `protocol_fee` can become 1. This means the total fee exceeds the `received_amount`, and the internal party receives a negative token amount. This may cause an unintentional underflow and introduce unexpected behavior.

```
    /// Validate a fee take for a given relayer fee and protocol fee
    fn validate_fee_take_singleprover(
        received_amount: Variable,
        relayer_fee: FixedPointVar,
        protocol_fee: FixedPointVar,
        fee_take: &FeeTakeVar,
        cs: &mut PlonkCircuit,
    ) -> Result<(), CircuitError> {
        let expected_relayer_fee = relayer_fee.mul_integer(received_amount, cs)?;
        let expected_protocol_fee = protocol_fee.mul_integer(received_amount, cs)?;

        FixedPointGadget::constrain_equal_integer_ignore_fraction(
            expected_relayer_fee,
            fee_take.relayer_fee,
            cs,
        )?;

        FixedPointGadget::constrain_equal_integer_ignore_fraction(
            expected_protocol_fee,
            fee_take.protocol_fee,
            cs,
        )
    }
```

For the matching amount, this issue allows the counterparty to steal `1` unit of the token in every transaction. For example, an attacker could propose an order to buy WBTC with `0` units of USDC. If there is a match, according to the rules, the attacker should receive `0` WBTC. However, due to ceiling rounding, the attacker could receive `1` unit of WBTC. This means the attacker can swap `0` units of USDC for `1` unit of WBTC (about $0.0006 at the time of writing) for free.

Recommendations. It is recommended to modify the rounding gadget to enforce strict floor rounding.

It is also recommended to prohibit zero-swaps (`buy_amount` = 0 or `sell_amount` = 0), as these are less intuitive and may introduce unexpected behavior in the future.

03 - No Restriction on ERC-20 Token Contracts Allows for Reentrant Execution

● renegade-contracts/*

Low

Description. Renegade supports arbitrary ERC-20 tokens on the platform. Currently, there are no restrictions on the token contract, allowing users to create custom tokens and add them to the pool. The functions in a token contract may contain arbitrary logic and introduce unexpected behavior.

For example, at the last step of an atomic settlement call, the dark pool will call `` `ERC20::transfer` `` to transfer the purchased token to the external party. An ERC-20 token contract might embed another atomic settlement call within the `` `ERC20::transfer` `` function, leading to nested atomic settlement calls. Below is the resulting call stack:

```
--> Darkpool::process_atomic_match_settle
    --> Darkpool::verify_proof
    --> ERC20::transfer
        --> Darkpool::process_atomic_match_settle
            --> Darkpool::verify_proof
            --> ERC20::transfer
            ...
```

This results in reentrant execution.

Recommendations. It's recommended to prevent reentrant execution in the contract as it may introduce unexpected behavior. Additionally, consider implementing an allowlist to restrict the platform to specific ERC-20 tokens.