ZKSECURITY

Audit of Renegade's Smart Contracts

February 26, 2024

Introduction

On February 26, 2024, Renegade tasked zkSecurity with auditing its smart contracts. The specific code to review was shared via GitHub as public repositories. The audit lasted 3 weeks with 3 consultants.

Written specifications detailing the circuits, the smart contracts, and additional protocols (like the proof linking scheme) were also shared.

The documentation provided, and the code that zkSecurity looked at, was found to be of prime quality. The Renegade team was very responsive and helpful in answering questions and providing additional context.

Scope

The scope of the audit included the following components:

Whitepaper: Circuit Specification. The circuit specifications for all proofs, including checks of "linked" wires across proofs. zkSecurity checked that nothing was under-constrained in the specification itself. The most up-to-date specification is Updated Circuit Specification.

Whitepaper: Linkable Proofs. The mathematical specification for "linkable proofs". zkSecurity ensured that auxiliary proofs were sound and faithfully implemented cross-proof wire constraints. Location: Proof Linking PlonK

Circuit Spec Implementation. Implementation of the circuit specification. zkSecurity audited the gadgets and high-level circuits to ensure correct implementations of the specification. Specific Directories included:

1. `` `renegade/circuits/src/zk_circuits` `` — All circuit definitions.
2. `` `renegade/circuits/src/zk_gadgets` `` — ZK gadgetry used in the circuits (e.g. ElGamal, Poseidon, etc).
3. `` `renegade/circuit-types/src` `` — Type definitions for circuits, including how we convert runtime types into witness types in the PlonK wire assignment.
4. `` `renegade/circuit-macros/src/circuit_type` `` — Macro definitions that derive the traits in `` `traits.rs` ``.

SRS logic. SRS logic to handle verification keys. Specific Directories:

1. `` `renegade/circuit-types/src/srs.rs` `` — SRS-related types.
2. `` `renegade-contracts/scripts/src/commands.rs` `` — Script to handle verification keys.
3. `` `renegade-contracts/contracts-stylus/vkeys/prod` `` — Verification key outputs.

Renegade smart contracts. Renegade-specific logic inside the Stylus smart contract. Includes Merkle tree implementation, nullifier set logic, storing of secret shares, and access control. Specific directories:

1. `` `renegade-contracts/contracts-stylus/src` `` — Smart contracts.

2. `renegade-contracts/contracts-common/src` — Types and serialization logic used both in the contracts and in surrounding testing / utilities.
3. `renegade-contracts/contracts-core/src/crypto` — Implementation of ECDSA verification (using the EVM's `ecRecover` precompile) and Poseidon 2 hashing (imported from the relayer codebase).

PlonK verifier. The core PlonK verifier itself inside Stylus smart contract, including "linkable" proof checks. Specific Directories:

1. `renegade-contracts/contracts-core/src` — Core transcript and PlonK verification logic.

Recommendations

In addition to the findings discussed later in the report, we have the following strategic recommendations:

Audit integration. Ensure that integration with external systems is secure. While this audit specifically focused on the parts of Renegade described above, security of the entire system is only as strong as its weakest link. This includes the P2P network protocol between relayer nodes and price discovery system (since optimal execution price seems to be an important property of the system).

Fix high-severity issues. Ensure that the findings with high severity are properly fixed, as these findings can lead to critical issues. See Deposits Can Be Stolen By Frontrunners. Consider re-auditing the affected components.

Specify more parts of the system. Consider writing a specification of the P2P network protocol for discovery of matching orders across different relayers and finding (and, perhaps, proving) the theoretically-optimal midpoint price (as per https://docs.renegade.fi).

Overview of Renegade

Renegade is a dark pool trading platform that allows users to trade ERC-20 assets without revealing any information about their trades and balances (except when depositing to or withdrawing from Renegade). Renegade is a decentralized system built on Arbitrum Stylus, an Ethereum Layer 2.

Renegade's network node software and smart contracts are written in Rust: Arbitrum Stylus enables smart contracts compiled to Wasm which is a great compile target for Rust.

Architecturally, Renegade is built on top of a Zcash-like shielded pool built within a smart contract. Users can create and "update" their wallets to deposit or withdraw balances, as well as create and cancel orders.

The matching engine in charge of matching orders is a multi-party computation (MPC) involving two wallets, and producing a zero-knowledge proof if and only if the match is successful.

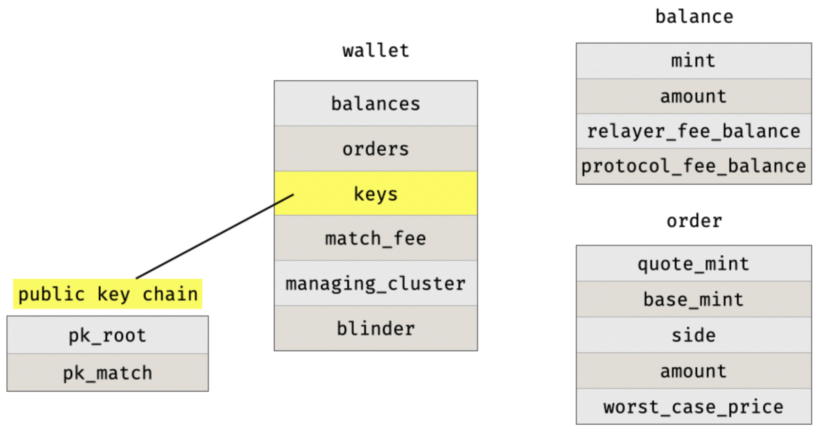
As the matching engine of Renegade requires a wallet owner to be online to run synchronous protocols with other nodes, agents called *relayers* can be used to delegate the matching and settlement of orders. In exchange for this service, relayers collect fees for each successful match.

Next, we introduce the lowest primitive used to build Renegade: a *wallet*.

Wallets

A *wallet* is a secret-shared data structure hosting a number of fields:

- 5 asset `balances`, which all have to be different with regard to the asset held
- 5 active `orders`, which can be matched against other wallets' orders
- 2 public `keys`: `pk_root`, allows the user to authenticate themselves, and `pk_match`, allows the wallet's associated relayer to authenticate themselves
- the `match_fee` rate that the wallet owner is willing to pay for the relayer's service
- the `managing_cluster` encryption key — involved in relayer fee settlement
- a `blinder` used to blind shares of the wallet (more on that later)



Such wallets are interpreted as a number of field elements, and these are passed as *blinded additive secret shares* to the Renegade smart contract.

By *secret shares* we mean that each value w_i that make up a wallet are split (shared) into two values $w_{i,1}, w_{i,2}$ such that $w_i = w_{i,1} + w_{i,2}$. Of these two secret shares, one is public and one is private. The public share is *blinded* (see below) for it to be safe to be publicly passed to the smart contract. It's referred to as the *blinded public share* (or simply *public share*). The *private share* is only known to the wallet owner (and its relayer).

Splitting the values of a wallet between public and private shares allows for several interesting properties:

1. The MPCs can be done much more efficiently as some of the inputs are already secret-shared.
2. The values of a wallet can be updated by updating the public shares in the same way: additions and subtractions transfer to the underlying values.

Blinding and Reconstructing a Wallet

Note that knowing the public share of a value along with the plaintext value would allow someone to recompute its private share. As such, wallet values are actually split into three shares, the third share being the *blinder*. The blinder value is itself shared into two additive shares, which is OK as it is supposedly random.

As an implementation detail, note that the blinder is part of the wallet and is thus secret-shared in the same way that the wallet is (but obviously, without being itself blinded). As such, the unblinding and blinding operations often have to treat the blinder field as an edge case:

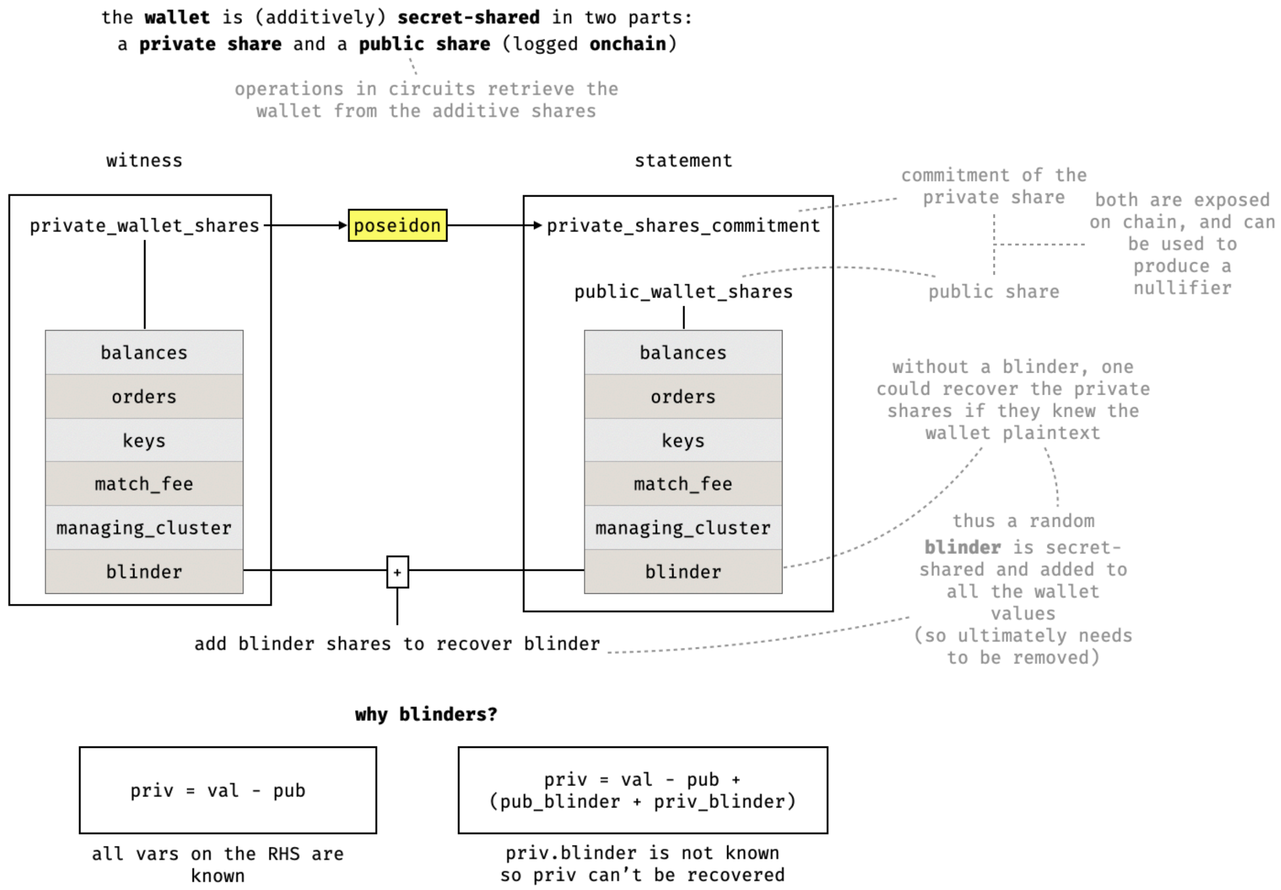
```
/// Unblinds the wallet, but does not unblind the blinder itself
pub fn unblind_shares<C: Circuit<ScalarField>>(
    self,
    blinder: Variable,
    circuit: &mut C,
) -> WalletShareVar<MAX_BALANCES, MAX_ORDERS> {
    let prev_blinder = self.blinder;
    let mut unblinded = self.unblind(blinder, circuit);
    unblinded.blinder = prev_blinder;

    unblinded
}
```

To recover the whole wallet from `` `private_share` `` and `` `blinded_public_share` ``, the following algorithm is used:

1. Recover the blinder: `` `blinder = blinded_public_share.blinder + private_share.blinder` ``.
2. Unblind the public share: `` `public_share = blinded_public_share.unblind_shares(blinder)` ``.
3. Recover the wallet: `` `wallet = private_share + public_share` ``.

We illustrate the shares and the blinding process in the following diagram:



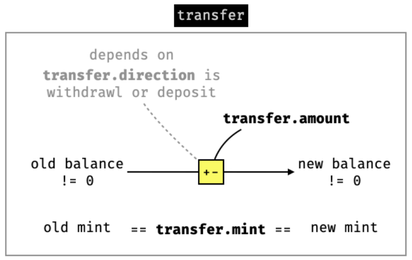
the **wallet** is (additively) **secret-shared** in two parts:
a **private share** and a **public share** (logged **onchain**)
operations in circuits retrieve the wallet from the additive shares
witness
statement
private_wallet_shares
poseidon
private_shares_commitment
commitment of the private share
both are exposed on chain, and can be used to produce a nullifier
public_wallet_shares
public share
balances
orders
keys
match_fee
managing_cluster
blinder
balances
orders
keys
match_fee
managing_cluster
blinder
+
without a blinder, one could recover the private shares if they knew the wallet plaintext
thus a random **blinder** is secret-shared and added to all the wallet values (so ultimately needs to be removed)
add blinder shares to recover blinder
why blinders?
priv = val - pub
all vars on the RHS are known
priv = val - pub + (pub_blinder + priv_blinder)
priv.blinder is not known so priv can't be recovered

Wallet Operations

There are two user-initiated wallet updates:

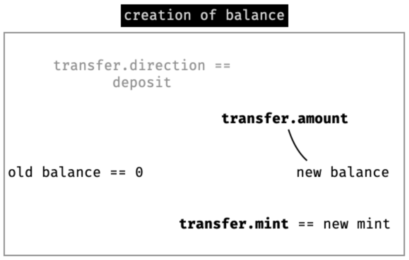
- **Wallet Create**. Create a wallet.
- **Wallet Update**. Update a wallet, which includes updating keys, balances, and orders.

with the latter one letting you do most of the user updates. This includes change of keys, balance, and orders. We illustrate, as an example, the balance updates enforced in the different flows:



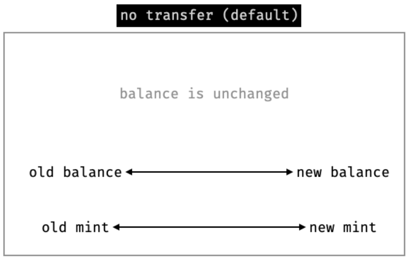
transfer idx == balance idx

and old balance != 0 && old mint != 0

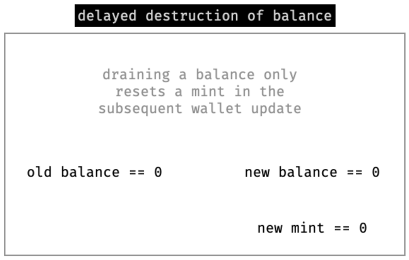


transfer idx == balance idx

and old balance == 0



transfer idx != balance idx



transfer idx != balance idx

and old balance == 0

Other wallet operations are performed by relayers. The main one is the **process match settle** which allows a relayer to finalize a match between two orders.

In addition, there are a number of fee functions:

- **ValidRelayerFeeSettlement** is used when a wallet that has an unpaid fee balance is managed by the relayer getting the match fee.
- **ValidOfflineFeeSettlement** is used when the match fee recipient relayer may be offline or unresponsive. An *encrypted note* is created with the fee amount and the unpaid fee balance is set to `` `0` ``.
- **ValidFeeRedemption** is the other side of **ValidOfflineFeeSettlement**: the fee from the note is redeemed and added to the balance of the recipient wallet.

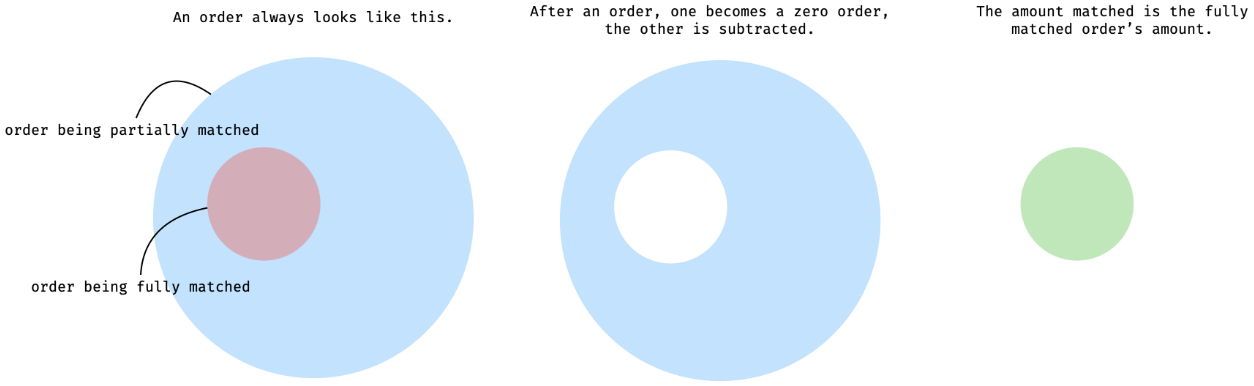
Note that each operation is often split into two parts:

- an off-chain part implemented as a circuit, producing a ZK proof of some statement
- an on-chain part implemented as a smart contract, verifying the ZK proof and working with on-chain persistent state

The most complex operation is the match settlement, which we explain in further detail in the next section.

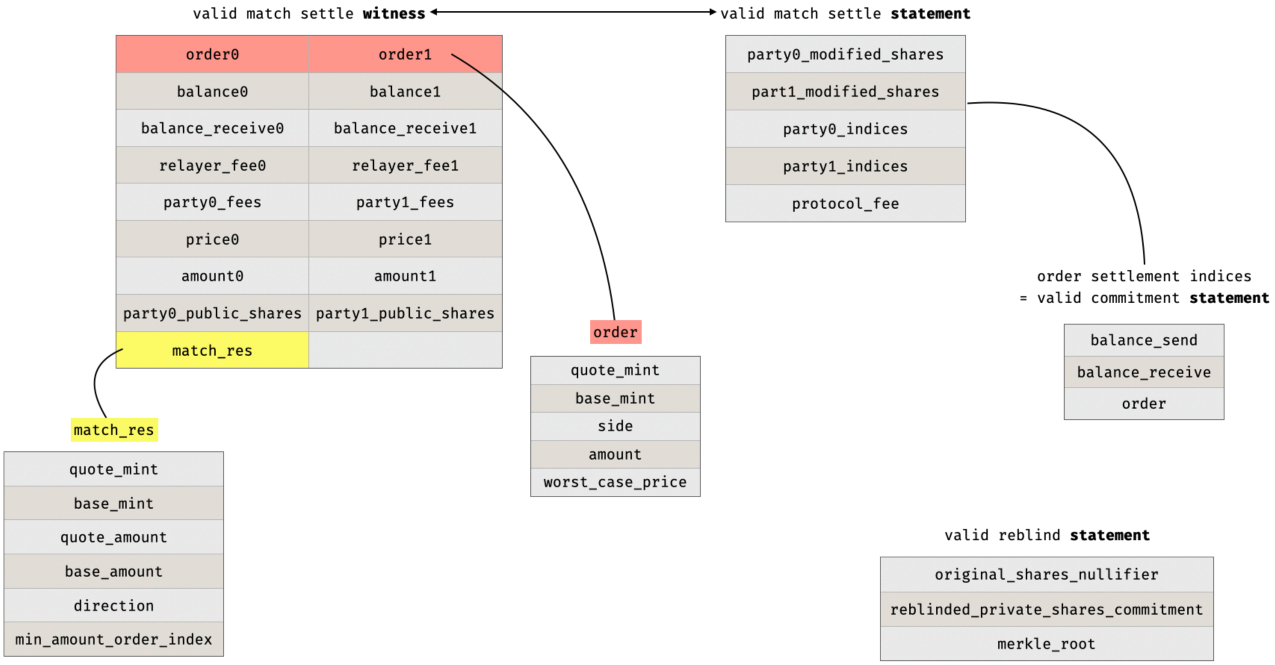
Order Matching

Order matching is the core of the system and, not surprisingly, the most complicated operation. Final order matching and settlement ZK statement comes as a bundle of linked proofs of statements about orders from two parties (**ValidReblind** and **ValidCommitments** — from each party, **ValidMatchSettle** — for the whole trade).


An order always looks like this.
order being partially matched
order being fully matched
After an order, one becomes a zero order,
the other is subtracted.
The amount matched is the fully
matched order's amount.

A **ValidMatchSettle** proof is either produced by a single relayer (in case when both parties' wallets are managed by the same relayer) or collaboratively by two relayers using MPC.

ValidReblind and ValidCommitments do not interact with smart contracts individually — only as a linked proof bundle with ValidMatchSettle.


valid match settle **witness**
valid match settle **statement**
order settlement indices
= valid commitment **statement**
valid reblind **statement**

order0	order1
balance0	balance1
balance_receive0	balance_receive1
relayer_fee0	relayer_fee1
party0_fees	party1_fees
price0	price1
amount0	amount1
party0_public_shares	party1_public_shares
match_res	

party0_modified_shares
part1_modified_shares
party0_indices
party1_indices
protocol_fee

match_res
quote_mint
base_mint
quote_amount
base_amount
direction
min_amount_order_index

order
quote_mint
base_mint
side
amount
worst_case_price

balance_send
balance_receive
order

original_shares_nullifier
reblinded_private_shares_commitment
merkle_root

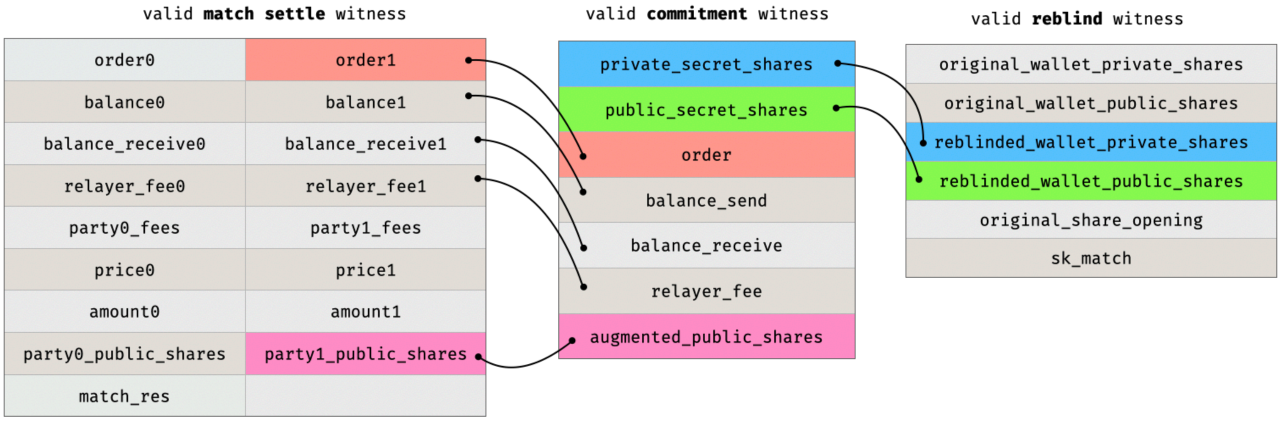
A (greatly) simplified view of how this works is:

1. First, **ValidReblind** proof is produced for a wallet ready for trading — populated with orders to be matched and has the corresponding assets in its balances.
2. Second, **ValidCommitments** proof is produced **for each order** in the wallet, just before looking for matches. As a result, we now have a re-blinded wallet and a set of commitments, ready to engage in order matching.
3. **ValidReblind** and **ValidCommitments** proofs are **linked** by values of some witness fields (namely, ``reblinded_wallet_private_shares``, ``reblinded_wallet_public_shares``).
4. The match is made by relayer nodes that, at this point, have a bundle of **ValidReblind** and **ValidCommitments** proofs (with their respective public inputs) for each order.

5. The MPC is performed and, if successful, results in **ValidMatchSettle** proof, linked with **ValidCommitments** from the two parties with updated balances from executed matching orders.
6. The proof bundle is communicated to the smart contract and verified as a whole, and the on-chain state is updated for both participating wallets (as described below, in *On-Chain Persistent State*).

Proof Linking

Some of the proofs in the proof bundle mentioned above are "linked" in order to ensure that they reuse some of their private inputs. The following diagram illustrates what is being linked between the different witnesses:



valid **match settle** witness

order0	order1
balance0	balance1
balance_receive0	balance_receive1
relayer_fee0	relayer_fee1
party0_fees	party1_fees
price0	price1
amount0	amount1
party0_public_shares	party1_public_shares
match_res	

valid **commitment** witness

private_secret_shares
public_secret_shares
order
balance_send
balance_receive
relayer_fee
augmented_public_shares

valid **reblind** witness

original_wallet_private_shares
original_wallet_public_shares
reblinded_wallet_private_shares
reblinded_wallet_public_shares
original_share_opening
sk_match

In order to prove that some of the same variables were used between two different witness polynomials a and b, the following identity is proven for some quotient polynomial q:

$$a(x) - b(x) = Z_S(x) \cdot q(x)$$

and $Z_S(x) = \prod_{i \in S}(x - g^i)$ — vanishing polynomial for the set S of the indices of the variables that are being linked.

This is equivalent to proving that polynomial $q(x)$ satisfying the above identity exists.

Note that as different circuits might be on different domains, values of a larger domain have to be aligned to match the values of the smaller domain (as only some of the rows of a larger domain align with the rows of a smaller domain, due to the 2-adicity of the field used).

The verifier has access to the commitments $[a]$, $[b]$, and then $[q]$, but can't produce a commitment to $[Z_S]$ as it involves multiple multiplications with $[x]$.

To work around that, the verifier demands to check the equation at some random value η:

$$a(\eta) - b(\eta) = Z_S(\eta) \cdot q(\eta)$$

which allows us to "linearize" the equation and compute $Z_S(\eta)$ first.

So once the verifier gets $[a]$, $[b]$, and $[q]$ they sample η. Then they produce $Z_S(\eta)$ and compute the commitment of

$$f(x) = a(x) - b(x) - Z_S(\eta) \cdot q(x)$$

they can do that by computing

$$[f] = [a] - [b] - Z_S(\eta) \cdot [q]$$

The only thing left to for the prover is to show, using KZG, that $f(\eta) = 0$, this implies showing that there exists a a polynomial q', such that

$$f(x) - 0 = (x - \eta) \cdot q'(x)$$

or, in terms of commitments,

$$[f] = (x - \eta) \cdot [q']$$

The last identity is the same as $x \cdot [q'] = [f] + \eta \cdot [q']$, which we can check in a pairing

$$e([q']_1, [x]_2) = e([f]_1 + \eta \cdot [q']_1, [1]_2)$$

which ultimately looks like

$$e([q']_1, [x]_2) \cdot e(-[f]_1 - \eta \cdot [q']_1, [1]_2) = e([1]_1, [1]_2)$$

On-Chain Persistent State

The most important parts of the on-chain persistent state are:

- the nullifier set — stored as part of **`DarkpoolCore`** contract
- the current Merkle root — a part of **`Merkle`** contract
- the Merkle root history — a part of **`Merkle`** contract

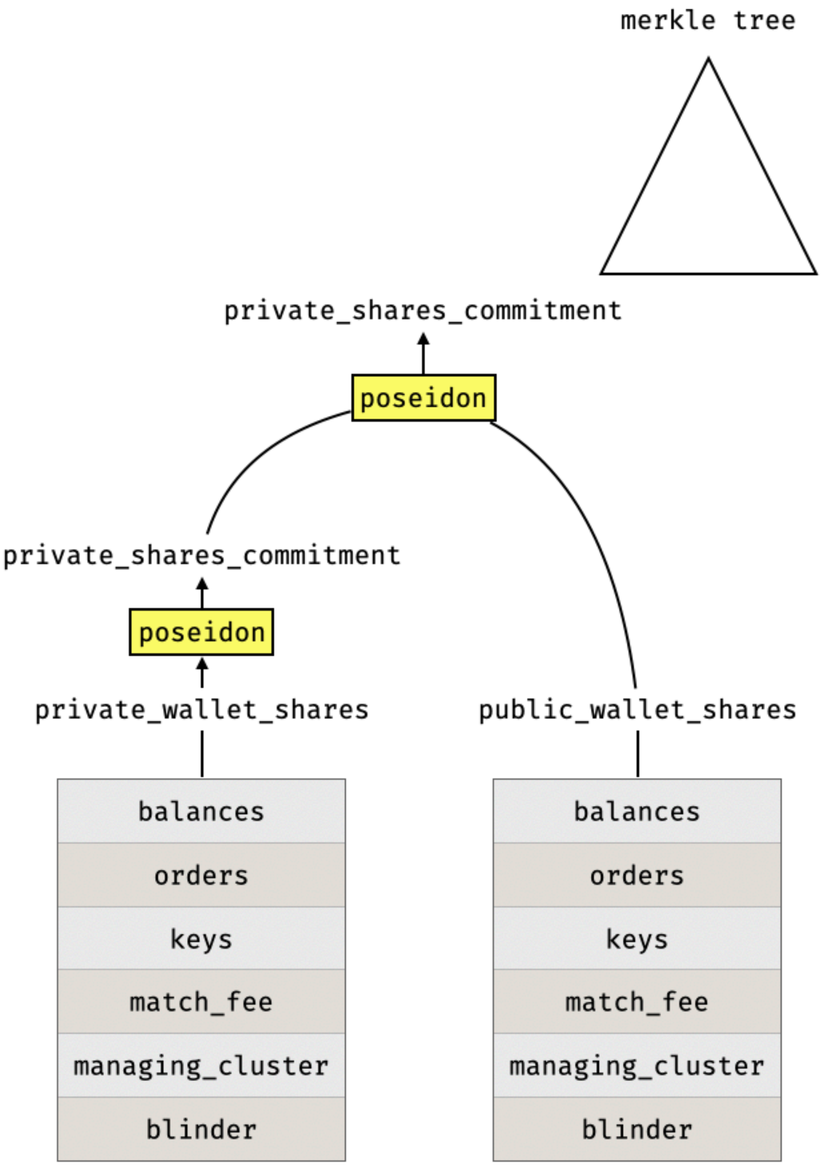
The snippet below contains the discussed fields:

```
/// The set of wallet nullifiers, representing a mapping from a nullifier
/// (which is a Bn254 scalar field element serialized into 32 bytes) to a
/// boolean indicating whether or not the nullifier is spent
nullifier_set: StorageMap<U256, StorageBool>,
// TRUNCATED...
/// The current root of the Merkle tree
pub root: StorageU256,
/// The set of historic roots of the Merkle tree
pub root_history: StorageMap<U256, StorageBool>,
```

Any operation, resulting in new or updated wallets, involves these interactions with the on-chain persistent states:

- Old wallets' Merkle roots are checked (that they exist in the Merkle root history).
- Old wallet nullifiers are also checked (that they **don't** yet exist) and inserted into the nullifier set.
- New / updated wallet commitments are computed and inserted into the Merkle tree.

We illustrate below the Merkle tree used to store wallets (as well as encrypted notes for fees!):



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-circuits/transfer_executor	Deposits Can Be Stolen By Frontrunners	High
02	renegade/zk_gadgets	Unsafe Euclidian Division	Medium
03	renegade/zk_gadgets	Poseidon Always Squeeze The Same Value	Medium
04	renegade/{zk_gadgets,renegade-crypto}	Random-Number Generators Are Not Backward Secure	Low
05	renegade-circuits/merkle	Duplicate Commitments Could Lead To Loss Of Funds	Low
06	renegade-contracts/contracts-core	VKey Not Absorbed When Generating The Challenge In Fiat-Shamir	Low
08	*	No Field Left Behind	Informational

# 00 - Deposits Can Be Stolen By Frontrunners

- **renegade-circuits/transfer_executor**

High

Description. A wallet update is an operation that encompasses a number of flows: deposit, withdraw, create or update orders, etc. A user performs a wallet update by:

1. consuming an existing wallet
2. updating their wallet
3. signing the new wallet

The first two steps are done in a private way via the use of zero-knowledge proofs. The last step is done out of circuit in the smart contract's logic. If the wallet update triggers a transfer of tokens (a withdrawal or deposit) then the smart contract handles some separate logic to process the transfer.

More specifically, the permit2 contract and its `permitTransferFrom` function are used to perform the transfer of on-chain funds in a user-friendly way.

The transfer logic of the Renegade smart contract is described here in pseudo-code:

```
def execute_external_transfer(old_pk_root, transfer, transfer_aux_data):
    if transfer.is_withdrawal:
        # either the
        assert(valid_sig(
            pub=old_pk_root,
            msg=transfer,
            sig=transfer_aux_data.transfer_signature)
            )
        transfer.mint.transfer_call(transfer.account_addr, transfer.amount)
    else: # deposit flow
       permit = {
          "permitted": [transfer.amount, transfer.mint],
          "nonce": transfer_aux_data.permit_nonce,
          "deadline": transfer_aux_data.permit_deadline,
       }
       signature_transfer_details = {
          "to": THIS_CONTRACT_ADDRESS,
          "requested_amount": transfer.amount
       }
       CONTRACT_PERMIT2_ADDRESS.permit_transfer_from_call(
            permit,
            signature_transfer_details,
            transfer.account_addr,
            transfer_aux_data.signature
        )
```

Note that `transfer_aux_data` is not authenticated by the proof. This means that two things can happen:

1. the auxiliary data can be reused in a different context by a frontrunning transaction or
2. someone can reuse the rest of a transaction with a different `transfer_aux_data`

The second attack does not work because the `transfer_aux_data` is basically a signature, and changing the data being signed (or the signer) will render the signature invalid. This signature passed as auxiliary data covers the token being transferred and its amount, the address spending it (the Renegade contract), as well as a nonce and a deadline.

As such, the auxiliary data is strongly tied to the transfer authenticated (passed as public input) by the proof, but not to any other user intent or the proof itself.

This makes the first attack possible. To perform such an attack, a malicious actor Bob could follow these steps:

1. Bob observes Alice trying to deposit some amount of token via an `update_wallet` transaction (via their mempool).
2. Bob creates a similar `update_wallet` transaction to deposit the same amount of tokens but for their own wallet, using Alice's `transfer_aux_data` (so that Alice funds Bob's wallet)
3. Now Bob is attempting to frontrun Alice's transaction and get theirs processed first (for example, by using a higher gas price).

Recommendation. The Renegade contract should use the permit2's `permitWitnessTransferFrom` function in place of `permitTransferFrom`, as it allows authenticating additional data as part of approving a transfer.

We recommend adding the proof's bytes to the witness field of the `permitWitnessTransferFrom` function, to strongly tie the transfer authorization to the user proof (which encapsulates the whole intent).

Alternatively, the `pk_root` of the recipient could be used instead, as it would strongly tie both ends of the transfer.

# 02 - Unsafe Euclidian Division

● **renegade/zk_gadgets**

Medium

Description. Renegade implements a number of circuit gadgets to perform fixed-point arithmetic operations. One of these internal gadgets (used in `` `FixedPointGadget::floor` ``) implements the euclidian division in the field arithmetic of the circuit field. That is, it calculates the values r and q such that:

$$\mathrm{a} = \mathrm{b} \cdot \mathrm{q} + \mathrm{r}$$

As of now, the implementation is not sound as it does not properly constrain the quotient variable (`` `q_var` ``) which could wrap around when multiplied with the divisor.

For example, an attacker might want to find q' and r' such that $\mathrm{a} = \mathrm{b} \cdot \mathrm{q}' + \mathrm{r}'$ and $\mathrm{q}' \neq \mathrm{q}$ and/or $\mathrm{r}' \neq \mathrm{r}$. This can be done by solving the following equation (obtained by subtracting the two equations):

$$0 = \mathrm{b}(\mathrm{q} - \mathrm{q}') + (\mathrm{r} - \mathrm{r}')$$

If we fix r', for example, we can solve for q'. That is, as long as r' is smaller than the divisor b, as it is constrained in the circuit:

```
// Constraint r < b
LessThanGadget::<D>::constrain_less_than(r_var, b, cs)?;
```

Note that we could not find a way to exploit this in practice, but the code can still become an issue in the future as it is easy to misuse.

Reproduction steps. The following sagemath code solves for a small example $\mathrm{a} = 563$ and $\mathrm{b} = 25$:

```
# define BN254 scalar field
F = GF(21888242871839275222246405745257275088548364400416034343698204186575808495617)

# setup a small problem
a = 563
b = 25
q = a // b # 22
r = a % b # 13
assert(a == q * b + r) # 563 = 25 * 22 + 13

# let's set r' = 10
r2 = 10
q2 = F((b * q + r - r2)/b) #
7879767433862139080008706068292619031877411184149772363731353507167291058400
```

```
assert(q2 != q)
assert(r2 != r)
assert(r2 < b)
assert(a == b * q2 + r2)
```

One can test the attack in Rust by modifying the behavior of the `` `DivRemGadget::div_rem` `` function as follows:

```
pub struct DivRemGadget<const D: usize> {}
impl<const D: usize> DivRemGadget<D> {
    /// Return (q, r) such that a = bq + r and r < b
    pub fn div_rem(
        a: Variable,
        b: Variable,
        cs: &mut PlonkCircuit,
    ) -> Result<(Variable, Variable), CircuitError> {
        let a_bigint = scalar_to_biguint(&a.eval(cs));
        let b_bigint = scalar_to_biguint(&b.eval(cs));

        // Compute the divrem outside of the circuit
-        let (q, r) = a_bigint.div_rem(&b_bigint);
+        let attack_a: BigUint = 563u64.into();
+        let attack_b: BigUint = 25u64.into();
+        let (q, r) = if a_bigint == attack_a && b_bigint == attack_b {
+            let q_str =
+
"14008475437977136142237699676964656056670953216266261979966850679408517437217";
+            let q = q_str.parse().unwrap();
+            let r = 10u64.into();
+            (q, r)
+        } else {
+            a_bigint.div_rem(&b_bigint)
+        };

        let q_var = biguint_to_scalar(&q).create_witness(cs);
        let r_var = biguint_to_scalar(&r).create_witness(cs);

        // Constrain a == bq + r
        let one_var = cs.one();
        let one = ScalarField::one();
        cs.mul_add_gate(&[b, q_var, r_var, one_var, a], &[one, one])?;

        // Constraint r < b
        LessThanGadget::<D>::constrain_less_than(r_var, b, cs)?;

        Ok((q_var, r_var))
    }
}
```

And then run the following test:

```rust
#[test]
fn test_div_rem_attack() {
    let aa = BigUint::from(563u32);
    let bb = BigUint::from(25u32);

    let (expected_q, expected_r) = aa.div_rem(&bb);
    let expected_q = biguint_to_scalar(&expected_q);
    let expected_r = biguint_to_scalar(&expected_r);

    // Build a constraint system
    let mut cs = PlonkCircuit::new_turbo_plonk();
    let expected_q_var = expected_q.create_public_var(&mut cs);
    let expected_r_var = expected_r.create_public_var(&mut cs);

    // Allocate the inputs in the constraint system
    let dividend_var = biguint_to_scalar(&aa).create_witness(&mut cs);
    let divisor_var = biguint_to_scalar(&bb).create_witness(&mut cs);

    let (q_res, r_res) =
        DivRemGadget::<32 /* bitlength */>::div_rem(dividend_var, divisor_var, &mut cs)
            .unwrap();

    let q_eq = cs.is_equal(expected_q_var, q_res).unwrap();
    use circuit_types::traits::CircuitVarType;
    assert!(!q_eq.eval(&cs));
    let r_eq = cs.is_equal(expected_r_var, r_res).unwrap();
    assert!(!r_eq.eval(&cs));

    assert!(cs.check_circuit_satisfiability(&[expected_q.inner(), expected_r.inner()]).is_ok())
}
```

Recommendation. We recommend constraining the size of q such that the equation being constrained cannot wrap around. As a and b are both assumed to be 32-bit values, q could easily be constrained to be 32-bit as well.

# 03 - Poseidon Always Squeeze The Same Value

- **renegade/zk_gadgets**

Medium

Description. Renegade uses the Poseidon hash function in multiple places in its circuit logic. The Poseidon hash function uses the "sponge" paradigm, which is to absorb data, and then squeeze as many times as you want to output different random values. This kind of construction has been called XOF for eXtended Output Functions in the past.

Unfortunately, the current implementation produces the same value when squeezed multiple times. While the current code never squeezes more than once, which does not lead to vulnerabilities in the protocol, we highly recommend fixing the issue as it is not clear how the code will be used in the future.

The problem is in this part of the code:

```
impl PoseidonHashGadget {
    /// Squeeze an element from the sponge and return its representation in the
    /// constraint system
    pub fn squeeze<C: Circuit<ScalarField>>(
        &mut self,
        cs: &mut C,
    ) -> Result<Variable, CircuitError> {
        // Once we exit the absorb state, ensure that the digest state is permuted
        // before squeezing
        if !self.in_squeeze_state || self.next_index == RATE {
            self.permute(cs)?;
            self.next_index = 0;
            self.in_squeeze_state = true;
        }

        Ok(self.state[CAPACITY + self.next_index])
    }
```

As one can see, the `next_index` is never incremented, and so it always points to the same value to squeeze in the state.

Recommendation. Make sure that the squeeze function properly increments the `next_index` variable:

```
+ let res = self.state[CAPACITY + self.next_index];
+ self.next_index += 1;
- Ok(self.state[CAPACITY + self.next_index])
+ Ok(res)
```

# 04 - Random-Number Generators Are Not Backward Secure

● renegade/{zk_gadgets,renegade-crypto}

Low

Description. Renegade uses different pseudo-random number generators (PRNGs) in its protocol to deterministically produce randomness.

PRNGs have different security properties, including **backward security** which dictates that one should not be able to predict future outputs based on the state of a PRNG. This property is useful in case the state of the PRNG state is leaked.

We found examples of non-backward secure PRNGs, which not only leak their state when producing random values, but also leak all future states. That being said, we could not find a way to exploit this particular issue in the protocol.

The problem resides in `renegade/circuits/src/zk_gadgets/poseidon.rs`:

```rust
pub struct PoseidonCSPRNGGadget;
impl PoseidonCSPRNGGadget {
    /// Samples values from a chained Poseidon hash CSPRNG, seeded with the
    /// given input
    pub fn sample<C: Circuit<ScalarField>>(
        mut seed: Variable,
        num_vals: usize,
        cs: &mut C,
    ) -> Result<Vec<Variable>, CircuitError> {
        let mut values = Vec::with_capacity(num_vals);

        // Chained hash of the seed value
        let mut hasher = PoseidonHashGadget::new(cs.zero() /* zero_var */);
        for _ in 0..num_vals {
            // Absorb the seed and then squeeze the next element
            hasher.absorb(seed, cs)?;
            seed = hasher.squeeze(cs)?;

            values.push(seed);

            // Reset the hasher state; we want the CSPRNG chain to be stateless, this
            // includes the internal state of the Poseidon sponge
            hasher.reset_state(cs);
        }

        Ok(values)
    }
}
```

As one can see, the squeezed value is used as both the random output and the seed for the next iteration of the PRNG. This means that learning one of the random values leaks the state, as well as future states.

This problem is mirrored in `renegade/renegade-crypto/src/hash.rs`:

```
/// Compute a chained Poseidon hash of the given length from the given seed
    pub fn evaluate_hash_chain(seed: Scalar, length: usize) -> Vec<Scalar> {
        let mut seed = seed.inner();
        let mut res = Vec::with_capacity(length);

        for _ in 0..length {
            // Create a new hasher to reset the internal state
            let mut hasher = Poseidon2Sponge::new();
            seed = hasher.hash(&[seed]);

            res.push(Scalar::new(seed));
        }

        res
    }
```

# 05 - Duplicate Commitments Could Lead To Loss Of Funds

● renegade-circuits/merkle

Low

Description. At the lower layer of Renegade lies a Zcash-like shielded pool on which the protocol is built. This Zcash-like shielded pool is a Merkle tree that stores commitments to the different user wallets and allows private retrieval and use of these wallets.

The way the shielded pool works is that before any wallet operations, users must prove in zero-knowledge proofs that they know the path to a wallet in a historical snapshot of that Merkle tree, and that the wallet hasn't been consumed before by revealing a unique value verifiably derived from the wallet itself, but hiding which wallet it was derived from.

Each time a wallet is used, it is consumed in this way and a new wallet along with a new derived commitment is produced and inserted in the tree.

Currently, the Merkle tree contract does not check if a commitment has already been added before adding it to the Merkle tree.

There's a few annoying edge cases that might arise from that. First, a user might mistakenly produce two wallets that converge to the same active commitments. This would be problematic as consuming one of the wallets would then void the other, which would cause a loss of funds.

A relayer could also produce a wallet with the same content as one of their user's wallets (and thus the same commitment), but doing that seems to only be a problem for the relayer (as they would need to match the `` `pk_root` `` field to obtain the same commitment, which would effectively transfer their funds to the targeted user).

While these two scenarios seem unlikely or benign, they appear as strange enough edge cases to warrant preventing this behavior.

Recommendation. Keep track of all commitments added previously, for example, by mirroring the way the Renegade smart contracts already keep track of past Merkle roots. In addition, ensure that a new commitment was not previously added before adding it.

# 06 - VKey Not Absorbed When Generating The Challenge In Fiat-Shamir

● **renegade-contracts/contracts-core**

Low

Description. Renegade uses *proof linking* to bind different (but related) Plonk proofs together into a single proof bundle. This means, there's a need to prove that some witness variables have exactly the same values between two linked proofs (as described in Proof Linking).

The verifier challenge η is generated using Poseidon hash from the proof linking statement, which consists of commitments to wire polynomials $[a]_1$ and $[b]_1$ and commitment to the linking quotient polynomial $[q]_1$.

The issue is: to verify the linking proof, the verifier computes the value of $Z_S(\eta)$, and the input data for this computation (the `` `LinkingVerificationKey` ``) is not absorbed into the hash to compute η.

```
let LinkingVerificationKey {
    link_group_generator,
    link_group_offset,
    link_group_size,
} = linking_vkey;

// TRUNCATED

let mut transcript = Transcript::<H>::new();
let eta = transcript
    .compute_linking_proof_challenge(
        wire_poly_comms.0,
        wire_poly_comms.1,
        linking_quotient_poly_comm,
    )
    .map_err(|_| VerifierError::ScalarConversion) ?;
```

With **fixed** LinkingVerificationKey, this should not be a problem: the variable parts (the "statement") are all bound when computing η.

Generally speaking, verification key, as well as the full statement, should always be absorbed when computing Fiat-Shamir challenges. Not doing so is known to have lead to vulnerabilities in the past (see https://eprint.iacr.org/2023/691).

Was LinkingVerificationKey not fixed (and controlled by the attacker), an attack would be easy to construct: make `` `link_group_size` `` == `` `0` ``. Proof and verification would succeed without any witness elements being necessarily equal.

Recommendation. Absorb the entire `LinkingVerificationKey`, as well as `wire_poly_comms.0`, `wire_poly_comms.1`, `linking_quotient_poly_comm`, to compute the challenge `eta`.

# 08 - No Field Left Behind

● *

Informational

Description. In several places in the codebase, all fields contained in a struct must be used or checked. For example, when verifying a state transition between a wallet and its updated counterpart, all untouched fields must remain the same.

```
fn verify_sender_state_transition(
    send_index: Variable,
    is_protocol_fee: BoolVar,
    old_wallet: &WalletVar<MAX_BALANCES, MAX_ORDERS>,
    new_wallet: &WalletVar<MAX_BALANCES, MAX_ORDERS>,
    cs: &mut PlonkCircuit,
) -> Result<(), CircuitError> {
    // All orders should be the same
    EqGadget::constrain_eq(&old_wallet.orders, &new_wallet.orders, cs)?;

    // The keys, match fee, and managing cluster should be the same
    EqGadget::constrain_eq(&old_wallet.keys, &new_wallet.keys, cs)?;
    EqGadget::constrain_eq(&old_wallet.match_fee, &new_wallet.match_fee, cs)?;
    EqGadget::constrain_eq(&old_wallet.managing_cluster, &new_wallet.managing_cluster, cs)?;

    // TRUNCATED...

    for (old_bal, new_bal) in old_wallet.balances.iter().zip(new_wallet.balances.iter()) {
        // TRUNCATED...
        EqGadget::constrain_eq(&old_bal.mint, &new_bal.mint, cs)?;
        EqGadget::constrain_eq(&old_bal.amount, &new_bal.amount, cs)?;
```

During the audit, we had to manually check this to make sure that every field was properly constrained. This is a tedious process and could lead to mistakes.

Fortunately, Rust provides a simple way to help by using the destructuring syntax. Destructing an object allows all the fields to enter the scope, and if a field is unused the compiler will throw a warning visible to the developer.

For example, the above code could be rewritten as:

```
fn verify_sender_state_transition(
    send_index: Variable,
    is_protocol_fee: BoolVar,
    old_wallet: &WalletVar<MAX_BALANCES, MAX_ORDERS>,
    new_wallet: &WalletVar<MAX_BALANCES, MAX_ORDERS>,
    cs: &mut PlonkCircuit,
) -> Result<(), CircuitError> {
    let WalletVar {
        balances: old_balances,
        orders: old_orders,
        keys: old_keys,
        match_fee: old_match_fee,
        managing_cluster: old_managing_cluster,
        blinder: _,
    } = old_wallet;
    let WalletVar {
        balances: new_balances,
        orders: new_orders,
        keys: new_keys,
        match_fee: new_match_fee,
        managing_cluster: new_managing_cluster,
        blinder: _,
    } = new_wallet;
    // All orders should be the same
    EqGadget::constrain_eq(old_orders, new_orders, cs)?;

    // The keys, match fee, and managing cluster should be the same
    EqGadget::constrain_eq(old_keys, new_keys, cs)?;
    EqGadget::constrain_eq(old_match_fee, new_match_fee, cs)?;
    EqGadget::constrain_eq(old_managing_cluster, new_managing_cluster, cs)?;

    // TRUNCATED...

    for (old_bal, new_bal) in old_balances.iter().zip(new_balances) {
        let BalanceVar {
            mint: old_mint,
            amount: old_amount,
            relayer_fee_balance: old_relayer_fee_balance,
            protocol_fee_balance: old_protocol_fee_balance,
        } = old_bal;
        let BalanceVar {
            mint: new_mint,
            amount: new_amount,
            relayer_fee_balance: new_relayer_fee_balance,
            protocol_fee_balance: new_protocol_fee_balance,
        } = new_bal;
        // TRUNCATED...
        EqGadget::constrain_eq(old_mint, new_mint, cs)?;
        EqGadget::constrain_eq(old_amount, new_amount, cs)?;
```