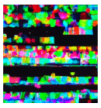

ZKSECURITY

# Audit of Penumbra's Circuits

**July 28th, 2023**

# Introduction

On July 10th, 2023, zkSecurity was commissioned to perform a security audit of Penumbra's circuits. For a total of five person-weeks, three consultants reviewed all the zero-knowledge circuits of the Penumbra protocol, looking for security as well as privacy issues.

The code was found to be thoroughly documented, rigorously tested, and well specified.

A number of issues were found, which are detailed in the following sections. One consultant performed three days of review following the audit, in order to ensure that the issues were fixed correctly and without introducing new bugs. The result of the review is included in this report as well (under each finding).

In addition, zkSecurity reported some overconstraints and optimization opportunities, as well as light issues with the specifications. For the sake of brevity, most of these comments were not included in this report.

## Scope

zkSecurity reviewed the release `0.56.0` of the Penumbra repository.

Included in scope were any of the circuits written with the **arkworks r1cs-std** library. At a high level this included:

- **Penumbra's multi-asset shielded pool**, which is used to store the balances of all users in a private way.
- **Staking and delegation**, which allows users to delegate their balances to a validator, and receive rewards for doing so.
- **Decentralized exchange**, which is used to trade assets in a private way.
- **Governance**, which is used to vote on proposals in the protocol.

Some or more of this logic made use of lower-level gadgets that were included in the scope as well:

- **decaf377**, the group used in circuits to perform cryptographic operations.
- **poseidon377**, the hash function used in circuits.
- **Tiered Commitment Tree**, the Merkle tree used in the protocol.
- **Fixed-point arithmetic**, low-level operations to handle precision in non-integer operations.

## Recommendations

We make the following recommendations to the Penumbra team:

- Fix the important findings found in this report.
- Perform an additional audit if circuits need to be changed due to the **Flow Encryption** feature.
- Consider conducting an audit of state transitions between transaction actions.

# Background on the Penumbra protocol

Penumbra is a **Cosmos zone** (an application-specific blockchain in the Cosmos ecosystem) where the main token is used to delegate power to validators and vote on proposals. The protocol allows the Penumbra token itself, as well as external tokens (connected to Penumbra via the **IBC protocol**), to live in a shielded pool a la Zcash where transactions within the same token provide full privacy.

The protocol's main feature is a decentralized exchange (also called DEX) in which users can trade tokens in the clear (by moving their shielded tokens to open positions), or in a private way if they choose to trade at market price.

To enable the privacy features of Penumbra, **zero-knowledge proofs** are used. Specifically, the **Groth16** proof system is used on top of the **BLS12-377** elliptic curve.

In order to perform group operations within the circuit, the **Edwards curve introduced in the ZEXE paper** was chosen as the "inner curve". In addition, the curve is never used directly, but rather through the **decaf377** abstraction/encoding.

In the next sections we introduce different aspects of the protocol, when relevant to the zero-knowledge proofs feature of the protocol.

## Statements in code

The zero-knowledge proofs in Penumbra can be observed in users' transactions, proving that specific statements are correct to the validators, and allowing the validators to safely perform state transitions in the consensus protocol.

Each transaction can contain a list of actions, defined in the **specification**. Not all actions contain zero-knowledge proofs, as some of them will be performed in the clear. Interaction between actions that modify balances, specifically between private actions themselves, or between private and public actions, happen hidden within commitments.

For example, as part of a single transaction some actions might create positive balances for some tokens hidden in commitments, and some other actions will create negative balances for some tokens hidden in commitments. The result will be verified to be a commitment to a 0 balance, either in the clear or via a proof from the user (as commitments can be hiding). (The proof is also embedded in a user's signature via the use of **binding signatures**.)

In the codebase, each component is cleanly separated from the rest and self-contained within a crate (Rust library). Actions are encoded in `action.rs` files under the different crates, and circuits are encoded in `proof.rs` files under the same crate. Some gadgets are implemented under `r1cs.rs` files.

Circuits are implemented using of the **arkworks r1cs-std** library, and as such can easily be found as an implementation of a `ConstraintSynthesizer` trait on circuit structures:

```
pub trait ConstraintSynthesizer<F: Field> {
    fn generate_constraints(self, cs: ConstraintSystemRef<F>) -> crate::r1cs::Result<()>;
}
```

To form a transaction, users follow a "transaction plan". The implementation of a transaction always leads the logic to `plan.rs` files which contain the creation of the action, including the creation of a proof for private actions.

On the other hand, an `action_holder/` folder can always be found for each component, which will determine how validators need to handle actions. For private actions, this will lead to the validators verifying the proofs contained in a transaction's actions. Verifications are split into two parts, a stateful one that performs checks which need to read the state, and a stateless one that performs any other checks:

```
pub trait ActionHandler {
    type CheckStatelessContext: Clone + Send + Sync + 'static;
    async fn check_stateless(&self, context: Self::CheckStatelessContext) -> Result<()>;
    async fn check_stateful<S: StateRead + 'static>(&self, state: Arc<S>) -> Result<()>;
    async fn execute<S: StateWrite>(&self, state: S) -> Result<()>;
}
```

An `ActionHandler` is also implemented for a transaction, which will call the relevant action handlers for each of the actions contained in the transaction. Among others, it also sums up each action's *balance commitment* in order to ensure (using the signature binding scheme previously mentioned) that they all sum up to 0.

In the rest of the sections we review the different actions that are related to zero-knowledge proofs, while at the same time giving high-level pseudocode description of the circuits (not-including gadgets and building blocks).

## Multi-asset shielded pool

Similar to Zcash, values in penumbra are stored as commitments in a Merkle tree. A particularity of Penumbra's Merkle tree is that each node links to four children. It is also technically a hyper tree (a tree of tree) where a leaf tree contains all of the commitments that were created in a block, the middle tree contains all of the blocks created in an epoch, and the top tree routes to the different epochs.

In order to spend a commitment, a spend action (and its spend proof) must be present in a transaction. In effect, the spend proof verifies that the note exists, and that it has never been spent. Revealing the commitment itself could allow validators to check if a commitment has been spent before, it leads to poor privacy. For this reason, another value strongly tied to the commitment is derived (provably) called a *nullifier*. In a sense, if Bitcoin tracks all of the unspent outputs, Penumbra tracks all of the spent ones instead.

In pseudocode, the logic of the spend circuit is as follows:

```
def spend(private, public):
    # private inputs
    note = NoteVar(private.note)
    claimed_note_commitment = StateCommitmentVar(private.state_commitment_proof.commitment)
```

```
position = PositionVar(private.state_commitment_proof.position)
merkle_path = MerkleAuthPathVar(private.state_commitment_proof)

v_blinding = uint8vec(private.v_blinding)

spend_auth_randomizer = SpendAuthRandomizer(private.spend_auth_randomizer)
ak_element = AuthorizationKeyVar(private.ak)
nk = NullifierKeyVar(private.nk)

# public inputs
anchor = public.anchor
claimed_balance_commitment = BalanceCommitmentVar(public.balance_commitment)
claimed_nullifier = NullifierVar(public.nullifier)
rk = RandomizedVerificationKey(public.rk)

# dummy spends have amounts set to 0
is_dummy = (note.amount == 0)
is_not_dummy = not is_dummy

# note commitment integrity
note_commitment = note.commit()
if is_not_dummy:
    assert(note_commitment == claimed_note_commitment)

# nullifier integrity
nullifier = NullifierVar.derive(nk, position, claimed_note_commitment)
if is_not_dummy:
    assert(nullifier == claimed_nullifier)

# merkle auth path verification against the provided anchor
if is_not_dummy:
    merkle_path.verify(position, anchor, claimed_note_commitment)

# check integrity of randomized verification key
computed_rk = ak_element.randomize(spend_auth_randomizer)
if is_not_dummy:
    assert(computed_rk == rk)

# check integrity of diversified address
ivk = IncomingViewingKey.derive(nk, ak_element)
computed_transmission_key = ivk.diversified_public(note.diversified_generator)
if is_not_dummy:
    assert(computed_transmission_key == note.transmission_key)

# check integrity of balance commitment
balance_commitment = note.value.commit(v_blinding)
if is_not_dummy:
    assert(balance_commitment == claimed_balance_commitment)

# check the diversified base is not identity
if is_not_dummy:
    assert(decaf377.identity != note.diversified_generator)
    assert(decaf377.identity != ak_element)
```

A spend proof includes a committed balance in its public input, exposing a hidden version of the balance that was contained in the note. As notes can hold different types of assets, the commitments to notes are computed as Pedersen commitments that use different base points for different assets.

As said previously, validators will eventually check that all actions from a transaction have balance commitments that cancel out. One of the possible ways to decrease the now positive balance is to create a *spendable output* for someone else (or oneself). The action to do that is called an *output action*, which simply exposes a different kind of (committed) balance: a negative one, potentially negating the balance of a spend action, but of other actions as well. An output proof also exposes a commitment to that new note it just created, allowing validators to add it to the hyper tree.

The pseudocode for that circuit is as follows:

```
def output(private, public):
    # private inputs
    note = NoteVar(private.note)
    v_blinding = uint8vec(private.v_blinding)

    # public inputs
    claimed_note_commitment = StateCommitmentVar(public.note_commitment)
    claimed_balance_commitment = BalanceCommitmentVar(public.balance_commitment)

    # check the diversified base is not identity
    assert(decaf377.identity != note.diversified_generator)

    # check integrity of balance commitment
    balance_commitment = BalanceVar(note.value, negative).commit(v_blinding)
    assert(balance_commitment == claimed_balance_commitment)

    # note commitment integrity
    note_commitment = note.commit()
    assert(note_commitment == claimed_note_commitment)
```

## Governance

A delegator vote is a vote on a proposal from someone who has delegated some of their tokens to a validator. Their voting power is related to their share of the total tokens delegated to that validator. As such, a delegator vote proof simply shows that they have (or had, at the time the proposal got created) *X* amount of delegated tokens in the pool of validator *Y*, revealing both of these values. In addition, it also reveals the nullifier correlated with the then-delegated note, in order to prevent double voting.

In exchange for voting, an NFT is created and must be routed towards a private output in the shielded pool (using an *output action*).

The pseudocode for the delegator vote statement is as follows:

```
def delegator_vote(private, public):
    # private inputs
    note = NoteVar(private.note)
    claimed_note_commitment = StateCommitmentVar(private.state_commitment_proof.commitment)

    delegator_position = private.state_commitment_proof.delegator_position
    merkle_path = merkleAuthPathVar(private.state_commitment_proof)

    v_blinding = u8vec(private.v_blinding)

    spend_auth_randomizer = SpendAuthRandomizer(private.spend_auth_randomizer)
```

```
ak_element = AuthorizationKeyVar(private.ak)
nk = NullifierKeyVar(private.nk)

# public inputs
anchor = public.anchor
claimed_balance_commitment = BalanceCommitmentVar(
    public.balance_commitment)
claimed_nullifier = NullifierVar(public.nullifier)
rk = RandomizedVerificationKey(public.rk)
start_position = PositionVar(public.start_position)

# note commitment integrity
note_commitment = note.commit()
assert(note_commitment == claimed_note_commitment)

# nullifier integrity
nullifier = NullifierVar.derive(
    nk, delegator_position, claimed_note_commitment)
assert(nullifier == claimed_nullifier)

# merkle auth path verification against the provided anchor
merkle_path.verify(delegator_position.bits(),
                   anchor, claimed_note_commitment)

# check integrity of randomized verification key
# ak_element + [spend_auth_randomizer] * SPEND_AUTH_BASEPOINT
computed_rk = ak_element.randomize(spend_auth_randomizer)
assert(computed_rk == rk)

# check integrity of diversified address
ivk = IncomingViewingKey: : derive(nk, ak_element)
computed_transmission_key = ivk.diversified_public(
    note.address.diversified_generator)
assert(computed_transmission_key == note.transmission_key)

# check integrity of balance commitment
balance_commitment = note.value.commit(v_blinding)
assert(balance_commitment == claimed_balance_commitment)

# check elements were not identity
assert(identity != note.address.diversified_generator)
assert(identity != ak_element)

# check that the merkle path to the proposal starts at the first commit of a block
assert(start_position.commitment == 0)

# ensure that the note appeared before the proposal was created
assert(delegator_position.position < start_position.position)
```

## Decentralized exchange

The main feature of Penumbra (besides its multi-asset shielded pool) is a decentralized exchange (also called DEX). In the DEX, peers can swap different pairs of tokens in the open by creating positions.

Positions, created by *position actions*, create negative balances as non-hiding commitments, aiming at canceling out spendings created by spend actions. As positions are a *public* market-making feature of the DEX, allowing

users to use different trading strategies (or provide liquidity to pairs of tokens) in the clear, we will not talk more about that in this document.

On the other side of the DEX are *Zswaps*, which allow users to trade tokens privately at "market price". A Zswap has two steps to it: a *swap*, and a *swap claim* (or *sweep*).

The swap action and proof takes a private *swap plaintext*, which describes the pair of tokens being exchanged, and what amount of tokens are being exchanged for each asset. The intention with having two amounts is to hide which is being traded. So in practice, only one amount will be set to non-zero.

The proof verifiably releases a commitment of the swap plaintext, which will allow the user to claim their tokens once the trade has been executed. The commitment to the swap plaintext is derived as a unique asset id, so as to be stored in the same multi-asset shielded pool as a commitment note of value 1 (i.e. as a non-fungible token).

In addition, the proof also releases a hidden commitment of a negative balance, subtracting both amounts from the user's balance. Validators will eventually verify that positive balances matching these negative balances are created in other actions of the same transaction.

(Note that a prepaid fee is also subtracted from the balance, which will allow the transaction to claim the result of that trade later on without having to spend notes to cover the transaction fee.)

The pseudocode of the swap statement is as follows:

```
def swap(private, public):
    # private inputs
    swap_plaintext = SwapPlaintextVar(private.swap_plaintext)
    fee_blinding = uint8vec(private.fee_blinding_bytes)

    # public inputs
    claimed_balance_commitment = BalanceCommitmentVar(
        public.balance_commitment)
    claimed_swap_commitment = StateCommitmentVar(public.swap_commitment)
    claimed_fee_commitment = BalanceCommitmentVar(public.fee_commitment)

    # swap commitment integrity check
    swap_commitment = swap_plaintext.commit()
    assert(swap_commitment == claimed_swap_commitment)

    # fee commitment integrity check
    fee_balance = BalanceVar.from_negative_value_var(swap_plaintext.claim_fee)
    fee_commitment = fee_balance.commit(fee_blinding)
    assert(fee_commitment == claimed_fee_commitment)

    # reconstruct swap action balance commitment
    balance_1 = BalanceVar.from_negative_value_var(swap_plaintext.delta_1)
    balance_2 = BalanceVar.from_negative_value_var(swap_plaintext.delta_2)
    balance_1_commit = balance_1.commit(0) # will be blinded by fee
    balance_2_commit = balance_2.commit(0) # will be blinded by fee
    transparent_balance_commitment = balance_1_commit + balance_2_commit
    total_balance_commitment = transparent_balance_commitment + fee_commitment

    # balance commitment integrity check
    assert(claimed_balance_commitment == total_balance_commitment)
```

Once a block has successfully processed the transaction containing this trade, the user can claim the result of the trade (i.e. the swapped tokens). To do that, a swap claim proof must be provided in which the user provides the path to the committed swap plaintext in the commitment tree, and exchanges it (or converts it) into two committed balances of the two traded tokens (which can then be routed to output notes using output actions within the same transaction).

The pseudocode for this circuit is the following:

```
def swap_claim(private, public):
    # private inputs
    swap_plaintext = SwapPlaintextVar(private.swap_plaintext)
    claimed_swap_commitment = StateCommitmentVar(
        private.state_commitment_proof.commitment)
    position_var = PositionVar(private.state_commitment_proof.position)
    position_bits = position.to_bits_le()
    merkle_path = MerkleAuthVar(private.state_commitment_proof)
    nk = NullifierKeyVar(private.nk)
    lambda_1_i = AmountVar(private.lambda_1_i)
    lambda_2_i = AmountVar(private.lambda_2_i)
    note_blinding_1 = private.note_blinding_1
    note_blinding_2 = private.note_blinding_2

    # public inputs
    anchor = public.anchor
    claimed_nullifier = NullifierVar(public.nullifier)
    claimed_fee = ValueVar(public.claim_fee)
    output_data = BatchSwapOutputDataVar(public.output_data)
    claimed_note_commitment_1 = StateCommitmentVar(public.note_commitment_1)
    claimed_note_commitment_2 = StateCommitmentVar(public.note_commitment_2)

    # swap commitment integrity check
    swap_commitment = swap_plaintext.commit()
    assert(swap_commitment == claimed_swap_commitment)

    # merkle path integrity. Ensure the provided note commitment is in the TCT
    merkle_path.verify(position_bits, anchor, claimed_swap_commitment)

    # nullifier integrity
    nullifier = NullifierVar.derive(nk, position_var, claimed_swap_commitment)
    assert(nullifier == claimed_nullifier)

    # fee consistency check
    assert(claimed_fee == swap_plaintext.claim_fee)

    # validate the swap commitment's height matches the output data's height (i.e. the
clearing price height)
    block = position_var.block  # BooleanVar[16..32] as FqVar
    note_commitment_block_height = output_data.epoch_starting_height + block
    assert(output_data.height == note_commitment_block_height)

    # validate that the output data's trading pair matches the note commitment's trading
pair
    assert(output_data.trading_pair == swap_plaintext.trading_pair)

    # output amounts integrity
    computed_lambda_1_i, computed_lambda_2_i = output_data.pro_rata_outputs(
```

```
        swap_plaintext.delta_1_i, swap_plaintext.delta_2_i)
    assert(computed_lambda_1_i == lambda_1_i)
    assert(computed_lambda_2_i == lambda_2_i)

    # output note integrity
    output_1_note = NoteVar(address=swap_plaintext.claim_address, amount=lambda_1_i,
                            asset_id=swap_plaintext.trading_pair.asset_1,
note_blinding=note_blinding_1)
    output_1_commitment = output_1_note.commit()

    output_2_note = NoteVar(address=swap_plaintext.claim_address, amount=lambda_2_i,
                            asset_id=swap_plaintext.trading_pair.asset_2,
note_blinding=note_blinding_2)
    output_2_commitment = output_2_note.commit()

    assert(output_1_commitment == claimed_note_commitment_1)
    assert(output_2_commitment == claimed_note_commitment_2)
```

## Staking

Delegation and undelegation to a validator's pool are both done "half in the clear". With the undelegation part being subject to a delay.

Delegation happens by providing spend actions and proofs that spend committed Penumbra notes in the multi-asset shielded pool. A public *delegation action* is then used in conjunction to subtract the number of Penumbra tokens from the transaction's balance and add a calculated amount of the validator's token to the transaction's balance. All of these extra balances are committed in a non-hiding way so that they can interact with the exposed committed balances of the private actions. Finally, output actions and proofs can use the positive validator token balance to produce new note commitments in the multi-asset shielded pool (by exposing an equivalent amount of negative validator token balance).

Undelegation happens in two steps. The first step is public, and converts (via a *undelegation action*) the delegated tokens into *unbounding tokens*. Unbounding tokens are tokens which asset ids are derived from the validator's unique identity and the epoch in which the tokens were unbounded. As such, an undelegation action simply creates a negative balance of delegated tokens (which the transaction provides via spend actions) and a positive balance of unbounding tokens (which the transaction can store back into the multi-asset shielded pool via output actions).

Once a proper delay has been observed (for some definition of proper), a user can submit a new transaction containing an *undelegate claim action* to the network. This action finally consumes the unbounding tokens created previously, converting them back into Penumbra tokens after a penalty has been applied (in cases where the validator might have misbehaved). The proof accompanying the action ensures that the released balance commitment correctly contains both a positive balance of Penumbra tokens (after correctly applying the penalty) and a negative balance of unbounding tokens. As such, such an action is accompanied by spend actions matching the balance of unbounding tokens and output actions matching the balance of Penumbra tokens.

What follows is the pseudocode for the undelegate claim circuit:

```
def undelegate_claim(private, public):
    # private inputs
    unbonding_amount = AmountVar(private.unbounding_amount)
    balance_blinding = Uint8vec(private.balance_blinding)
```

```
# public inputs
claimed_balance_commitment = BalanceCommitment(public.balance_commitment)
unbonding_id = AssetVarId(public.unbonding_id)
penalty = PenaltyVar(public.penalty)

# verify balance commitment
expected_balance = penalty.balance_for_claim(
    unbonding_id, unbonding_amount)
expected_balance_commitment = expected_balance.commit(balance_blinding)
assert(claimed_balance_commitment == expected_balance_commitment)
```

## 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 | core/transaction | Double-voting due to lack of duplicate nullifiers check | High |
| 01 | core/keys | Double spending due to incoming viewing key (ivk) derivation not constrained | High |
| 02 | core/num | Unsound fixed-point addition | High |
| 04 | core/num | Unsound fixed-point multiplication | High |
| 05 | core/num | Unsound division of AmountVar | Medium |
| 06 | circuits | Mixing runtime and compile-time logic | Informational |
| 07 | core/balance | Unoptimized balance computation | Informational |
| 08 | circuits | Unnecessary constraining output values | Informational |
| 09 | action handlers | Avoid user-controlled inputs when they can be recomputed | Informational |
| 0a | core/num | Unnecessary constraints in Amount::quo_rem | Informational |

## # 00 - Double-voting due to lack of duplicate nullifiers check

core/transaction

High

**Description**. Nullifier sets are global pools managed by each validator in order to ensure that no note is being spent more than once. Since delegator votes reveal nullifiers, but do not intend on spending them, delegator vote nullifiers are managed in separate per-proposal pools (as per the **specification**).

In addition to keeping track of nullifiers globally and checking nullifier-based actions against these global nullifier sets, validators must also check that within a transaction itself no nullifier is being used more than once. This is because transactions can contain multiple actions.

You can see this check in the `ActionHandler::check_stateless` implementation for a `Transaction`:

```
impl ActionHandler for Transaction {
    async fn check_stateless(&self, _context: ()) -> Result<()> {
        // TRUNCATED...

        no_duplicate_nullifiers(self)?;
```

This check only checks nullifiers released by *spend* and *swap claim* actions, which makes sense as delegator vote nullifiers must be checked separately.

Unfortunately no check seems to exist for checking duplicate delegator vote nullifiers within the same transactions, allowing users to vote multiple times with the same (then-)delegated notes as long as they do it within the same transaction.

**Recommendations**. Add a similar check as the `no_duplicate_nullifiers` check but for the delegator vote nullifiers.

In addition, add a negative test to ensure that voting twice using the same delegated note within the same transaction is not possible.

**Client Response**. The issue was fixed by adding the following function to the `ActionHandler` logic of the `Transaction` type, in order to check for duplicate delegator vote nullifiers:

```
fn no_duplicate_votes(tx: &Transaction) -> Result<()> {
    // Disallow multiple `DelegatorVotes`s with the same proposal and the same `Nullifier`.
    let mut nullifiers_by_proposal_id = BTreeMap::new();
    for vote in tx.delegator_votes() {
        // Get existing entries
        let nullifiers_for_proposal = nullifiers_by_proposal_id
            .entry(vote.body.proposal)
            .or_insert(BTreeSet::default());

        // If there was a duplicate nullifier for this proposal, this call will return
`false`:
        let not_duplicate = nullifiers_for_proposal.insert(vote.body.nullifier);
        if !not_duplicate {
```

```
                return Err(anyhow::anyhow!(
                    "Duplicate nullifier in transaction: {}",
                    &vote.body.nullifier
                ));
            }
        }

        Ok(())
    }
```

Notice that the set is per-proposal, in order to allow duplicate nullifiers when voting for different proposals that are happening at the same time.

## # 01 - Double spending due to incoming viewing key (ivk) derivation not constrained

core/keys

High

**Description**. In shielded transactions, a spend action allows one to spend one of their committed notes, as long as it hasn't been spent before. Since the spending of a note is hidden via zero-knowledge proofs, it cannot be known whether a note was spent or not by just looking at the notes themselves. For this reason, Penumbra introduces the concept of "*nullifiers*", which can be released in the clear as part of spending a note, and prevent reuse of notes as they are strongly tied to the notes themselves (without leaking to which note they are tied to).

In order to produce a nullifier, users possess *nullifier keys* `nk`s, which they can use with a note commitment in order to deterministically derive a nullifier.

In order to prove that a user can spend a note, they must prove that they own an *incoming viewing key* `ivk` which is strongly tied with the note to be spent.

What ties a nullifier to a note, is that the nullifier key `nk` and the incoming viewing key `ivk` are also strongly tied: the `ivk` is derived from the `nk`.

During the derivation of the `ivk`, the logic must convert a value from the circuit field (`Fq`) to a non-native field (`Fr`). This is because the `ivk` is a scalar value that is eventually used to perform a scalar multiplication. The problem arises in the implementation, which takes the value out of circuit to convert it to the other field:

```
impl IncomingViewingKeyVar {
    pub fn derive(nk: &NullifierKeyVar, ak: &AuthorizationKeyVar) -> Result<Self,
SynthesisError> {
        // TRUNCATED...
        let inner_ivk_mod_q: Fq = ivk_mod_q.value().unwrap_or_default();
        let ivk_mod_r = Fr::from_le_bytes_mod_order(&inner_ivk_mod_q.to_bytes());
        let ivk = NonNativeFieldVar::<Fr, Fq>::new_variable(
            cs,
            || Ok(ivk_mod_r),
            AllocationMode::Witness,
        )?;
```

In the above snippet, the circuit value of `ivk_mod_q` is extracted **out of circuit**, and then **reinserted as a witness value** under `ivk`. The problem is that doing that removes any constraint on `ivk` to the previous circuit value `ivk_mod_q`.

At this point, a malicious prover can decide to set anything as the output `ivk` of the derivation function. This is not very useful, as they must use a correct `ivk` later on to prove that they can spend the note.

This is not the end of it, a malicious prover can also go backward and decide not to derive the correct `ivk_mod_q` (but to use the correct `ivk` as the output of the function still). In this case, it allows the malicious prover to use an arbitrary nullifier key `nk`, and thus produce an incorrect nullifier.

This means that they can double spend (or spend as many times as they want) notes they own, simply by choosing a different nullifier every time.

**Reproduction steps**. The issue can be reproduced by modifying the test `spend_proof_verification_success` in `crates/core/component/shielded-pool/src/spend/proof.rs` to use a different nullifier key (and thus produce a different nullifier) based on the value of an environment variable:

```
        let note = Note::generate(&mut rng, &sender, value_to_send);
        let note_commitment = note.commit();
        let rsk = sk_sender.spend_auth_key().randomize(&spend_auth_randomizer);
        let mut nk = *sk_sender.nullifier_key();
+        if let Ok(s) = std::env::var("ZKSEC") {
+            let shift = Fq::from_str(&s).unwrap();
+            nk.0 += shift;
+        }
```

In order to derive the correct `ivk`, the `crates/core/keys/src/keys/ivk.rs` file needs to be modified as well to correct the `nk` when an attack is being performed:

```
pub fn derive(nk: &NullifierKeyVar, ak: &AuthorizationKeyVar) -> Result<Self,
SynthesisError> {
        let cs = nk.inner.cs();
        let ivk_domain_sep = FqVar::new_constant(cs.clone(), *IVK_DOMAIN_SEP)?;
        let ivk_mod_q = poseidon377::r1cs::hash_2(
            cs.clone(),
            &ivk_domain_sep,
            (nk.inner.clone(), ak.inner.compress_to_field()?),
        )?;

        // Reduce `ivk_mod_q` modulo r
+        let inner_ivk_mod_q = match (std::env::var("ZKSEC"), cs.is_in_setup_mode()) {
+            (Ok(shift), false) => {
+                let shift = Fq::from_str(&shift).unwrap();
+                let real_nk = nk.inner.value().unwrap() - shift;
+                let ak: Element = ak.inner.value().unwrap();
+                let compressed_ak = ak.vartime_compress_to_field();
+                poseidon377::hash_2(
+                    &IVK_DOMAIN_SEP,
+                    (real_nk, compressed_ak),
+                )
+            }
+            _ => ivk_mod_q.value().unwrap_or_default()
+        };
+        let ivk_mod_r =
+            Fr::from_le_bytes_mod_order(&inner_ivk_mod_q.to_bytes());
```

You can then run the test using a shift of 1, for example:

```
$ ZKSEC=1 cargo test --color=always --lib spend::proof::tests::zksec_double_spend -- --
nocapture
```

**Recommendation**. Add a constraint that verifies that the `ivk` value is a valid encoding (in `Fr`) of the previous `ivk_mod_q` value.

**Client Response**. Penumbra fixed the issue by computing the reduced value `res` out of circuit and proving that it correctly satisfies `ivk_mod_q = quotient * r_modulus + res` modulo the circuit field `Fq` (with `quotient <= 4`). This works because the modulus of the scalar field `Fr` is smaller than the circuit field `Fq`.

# # 02 - Unsound fixed-point addition

core/num

High

## Fixed-point arithmetic

**Description**. Parts of Penumbra's logic relies on a `U128x128Var` type and associated subcircuits. The type implements `checked_add`, to add two 256-bit values without allowing overflow. The function is underconstrained, producing the wrong results in some cases.

Under the hood, a `U128x128Var` stores its value as 4 limbs of `UInt64`s, an Arkworks type which internally simply contains vectors of boolean circuit variables.

The `checked_add` function first converts the limbs to field values, then adds them pairwise, tracking the carry for each limb. In addition, it enforces that the most significant carry is zero, to prevent overflow. The carry is tracked since addition of two 64-bit values can produce a 65-bit value, which is then split into a 64-bit value and a carry bit (e.g. `0b11 + 0b11 = 0b110`).

The following code shows the issue at play:

```
pub struct U128x128Var {
    pub limbs: [UInt64<Fq>; 4],
}

// TRUNCATED...

impl U128x128Var {
    pub fn checked_add(self, rhs: &Self) -> Result<U128x128Var, SynthesisError> {
        // x = [x0, x1, x2, x3]
        // x = x0 + x1 * 2^64 + x2 * 2^128 + x3 * 2^192
        // y = [y0, y1, y2, y3]
        // y = y0 + y1 * 2^64 + y2 * 2^128 + y3 * 2^192
        let x0 = Boolean::<Fq>::le_bits_to_fp_var(&self.limbs[0].to_bits_le())?;
        let x1 = Boolean::<Fq>::le_bits_to_fp_var(&self.limbs[1].to_bits_le())?;
        let x2 = Boolean::<Fq>::le_bits_to_fp_var(&self.limbs[2].to_bits_le())?;
        let x3 = Boolean::<Fq>::le_bits_to_fp_var(&self.limbs[3].to_bits_le())?;

        let y0 = Boolean::<Fq>::le_bits_to_fp_var(&rhs.limbs[0].to_bits_le())?;
        let y1 = Boolean::<Fq>::le_bits_to_fp_var(&rhs.limbs[1].to_bits_le())?;
        let y2 = Boolean::<Fq>::le_bits_to_fp_var(&rhs.limbs[2].to_bits_le())?;
        let y3 = Boolean::<Fq>::le_bits_to_fp_var(&rhs.limbs[3].to_bits_le())?;

        // z = x + y
        // z = [z0, z1, z2, z3]
        let z0_raw = &x0 + &y0; // 65-bit potentially
        let z1_raw = &x1 + &y1; // 65-bit potentially
        let z2_raw = &x2 + &y2; // 65-bit potentially
        let z3_raw = &x3 + &y3; // must be 64-bit, otherwise there is an overflow

        // z0 < 2^64 + 2^64 < 2^(65) => 65 bits
```

```
        let z0_bits = bit_constrain(z0_raw, 64)?; // no carry-in
        let z0 = UInt64::from_bits_le(&z0_bits[0..64]);
        let c1 = Boolean::<Fq>::le_bits_to_fp_var(&z0_bits[64..].to_bits_le()?)?;

        // z1 < 2^64 + 2^64 + 2^64 < 2^(66) => 66 bits
        let z1_bits = bit_constrain(z1_raw + c1, 66)?; // carry-in c1
```

The problem here comes from the wrong assumption on the contract of the `bit_constrain` function: the `bit_constrain` function returns an array of the size of its second argument (64), but the code assumes that the rest of the bits will trail the output of the function (if it was the case, the `bit_constrain` function would actually "throw", as it not only produces bit constraints for `n` bits of the given input, but also recombine them to enforce that they fully describe the given input).

So in practice, `z0_bits[64..].to_bits_le()` will return the empty vec in this code. Meaning that the carry `c1` will always be zero during proving.

**Reproduction steps**. Running a test with `u64::MAX` as the two inputs will produce a `z0` of size 64, and a `c1` of size 0, and the test will fail (since the test additionally enforces the result). One can do this by modifying the tests in `crates/core/num/src/fixpoint.rs` to use the specific `a` and `b` values as follows:

```
let (lo, hi) = (u64::MAX, 0);
let a = U128x128(U256::from_words(hi, lo));
let b = U128x128(U256::from_words(hi, lo));
```

Note that the current proptests did not surface this issue because they do not generate `U128x128` values in the full range of possibilities (they always set the low 128 bits to 0, and the high 128 bits are sampled from $[0, 2^{64})$).

**Recommendations**. Fix the property tests to generate values in the full range of possibilities, and fix the current logic in order to correctly enforce the carry.

**Client Response**. Penumbra released a fix that correctly computes each limb in the circuit.

# # 04 - Unsound fixed-point multiplication

core/num

High

**Description**. A similar error as with the **Unsound fixed-point addition** finding is present in the fixed-point multiplication `U128x128Var::checked_mul`. To see why the computation is erroneous, let's first see what the computation is trying to achieve.

Fixed-point multiplications handle scaled values $\mathrm{ax}$ and $\mathrm{ay}$ for some scalar $\mathrm{a}$. Multiplying them directly gives out a result $(\mathrm{ax})(\mathrm{ay}) = \mathrm{a}(\mathrm{axy})$ that needs to be unscaled; we want $\mathrm{a}(\mathrm{xy})$.

For $\mathrm{a} = 2^{128}$, which is what Penumbra uses, we need to multiply by $2^{-128}$ and truncate any decimals.

Since `U128x128Var` handles values chunked in limbs of size 64-bit, we have the following values computed on the limbs of the operands $\mathrm{x}$ and $\mathrm{y}$:

- $\mathrm{z}_0 = \mathrm{x}_0 \cdot \mathrm{y}_0$
- $\mathrm{z}_1 = \mathrm{x}_0 \cdot \mathrm{y}_1 + \mathrm{x}_1 \cdot \mathrm{y}_0$
- $\mathrm{z}_2 = \mathrm{x}_0 \cdot \mathrm{y}_2 + \mathrm{x}_1 \cdot \mathrm{y}_1 + \mathrm{x}_2 \cdot \mathrm{y}_0$
- $\mathrm{z}_3 = \mathrm{x}_0 \cdot \mathrm{y}_3 + \mathrm{x}_1 \cdot \mathrm{y}_2 + \mathrm{x}_2 \cdot \mathrm{y}_1 + \mathrm{x}_3 \cdot \mathrm{y}_0$
- $\mathrm{z}_4 = \mathrm{x}_1 \cdot \mathrm{y}_3 + \mathrm{x}_2 \cdot \mathrm{y}_2 + \mathrm{x}_3 \cdot \mathrm{y}_1$
- $\mathrm{z}_5 = \mathrm{x}_2 \cdot \mathrm{y}_3 + \mathrm{x}_3 \cdot \mathrm{y}_2$
- $\mathrm{z}_6 = \mathrm{x}_3 \cdot \mathrm{y}_3$

where

$$\mathrm{z} = \mathrm{z}_0 + \mathrm{z}_1 \cdot 2^{64} + \mathrm{z}_2 \cdot 2^{128} + \mathrm{z}_3 \cdot 2^{192} + \mathrm{z}_4 \cdot 2^{256} + \mathrm{z}_5 \cdot 2^{320} + \mathrm{z}_6 \cdot 2^{384}$$

Since we need to divide by $2^{128}$, we need to compute $\mathrm{z} \cdot 2^{-128}$:

$$\mathrm{z} \cdot 2^{-128} = \mathrm{z}_0 \cdot 2^{-128} + \mathrm{z}_1 \cdot 2^{-64} + \mathrm{z}_2 + \mathrm{z}_3 \cdot 2^{64} + \mathrm{z}_4 \cdot 2^{128} + \mathrm{z}_5 \cdot 2^{192} + \mathrm{z}_6 \cdot 2^{256}$$

and obtain $\mathrm{w} = \mathrm{w}_0 + \mathrm{w}_1 \cdot 2^{64} + \mathrm{w}_2 \cdot 2^{128} + \mathrm{w}_3 \cdot 2^{192}$ such that:

- $\mathrm{w}_0 = \mathrm{z}_2 + \mathrm{c}_0$
- $\mathrm{w}_1 = \mathrm{z}_3 + \mathrm{c}_1$
- $\mathrm{w}_2 = \mathrm{z}_4 + \mathrm{c}_2$
- $\mathrm{w}_3 = \mathrm{z}_5 + \mathrm{c}_3$

Where the $\mathrm{c}_i$ are the carries. For example, $\mathrm{c}_0$ is essentially the bits left after truncating the first 128 bits of $\mathrm{z}_0 + \mathrm{z}_1 \cdot 2^{64}$. Note that $\mathrm{z}_0 = \mathrm{x}_0 * \mathrm{y}_0$ is going to be of 128 bits, and thus you can discard it entirely. So $\mathrm{c}_0 = \mathrm{bits}(\mathrm{z}_1)[64..]$

(In addition we want to check that $\mathrm{z}_6 = 0$ because we do not allow overflow.)

We can see that the implementation is computing a wrong $\mathrm{w}_0$ right at the start, as it does not use $\mathrm{z}_2$:

```
pub fn checked_mul(self, rhs: &Self) -> Result<U128x128Var, SynthesisError> {
        // TRUNCATED...
```

```
        let z0 = &x0 * &y0;
        let z1 = &x0 * &y1 + &x1 * &y0;
        let z2 = &x0 * &y2 + &x1 * &y1 + &x2 * &y0;

        // TRUNCATED...

        let t0 = z0 + z1 * Fq::from(1u128 << 64);
        let t0_bits = bit_constrain(t0.clone(), 193)?;
        let t1 = z2 + Boolean::<Fq>::le_bits_to_fp_var(&t0_bits[128..193].to_bits_le()?)?;
        let t1_bits = bit_constrain(t1, 129)?;

        let w0 = UInt64::from_bits_le(&t0_bits[0..64]);
```

**Reproduction steps**. One can produce an erroring test by modifying the tests in `crates/core/num/src/fixpoint.rs` to use the specific `a` and `b` values as follows

```
let (hi, lo) = (0, 1);
        let a = U128x128(U256::from_words(hi, lo));
        let b = U128x128(U256::from_words(hi, lo));
```

**Recommendations**. As with the **Unsound fixed-point addition** finding, fix the property tests to use the full domain of possible `U128x128` values. In addition, ensure that the limbs of the result are computed correctly according to the description of this finding.

**Client Response**. Penumbra fixed the issue by correctly constraining the limbs.

## # 05 - Unsound division of AmountVar

core/num

Medium

**Description**. The `AmountVar` type is used to represent amounts of tokens in all of Penumbra's circuits. Different operations can be performed on `AmountVar` types, including divisions via the `AmountVar::quo_rem` function. The gadget is currently underconstrained, allowing malicious provers to arbitrary choose the remainder part of the division. The reason why is that the quotient `quo_var` is not constrained at all:

```
pub fn quo_rem(
    &self,
    divisor_var: &AmountVar,
) -> Result<(AmountVar, AmountVar), SynthesisError> {
    let current_amount_bytes: [u8; 16] = self.amount.value().unwrap_or_default().to_bytes()
        [0..16]
        .try_into()
        .expect("amounts should fit in 16 bytes");
    let current_amount = u128::from_le_bytes(current_amount_bytes);
    let divisor_bytes: [u8; 16] = divisor_var.amount.value().unwrap_or_default().to_bytes()
        [0..16]
        .try_into()
        .expect("amounts should fit in 16 bytes");
    let divisor = u128::from_le_bytes(divisor_bytes);

    // Out of circuit
    let quo = current_amount.checked_div(divisor).unwrap_or(0);
    let rem = current_amount.checked_rem(divisor).unwrap_or(0);

    // Add corresponding in-circuit variables
    let quo_var = AmountVar {
        amount: FqVar::new_witness(self.cs(), || Ok(Fq::from(quo)))?,
    };
```

Notice that if the quotient $q$ is not constrained in a division $x = qy + r$ in the field, then you can set the remainder $r$ arbitrarily and compute the quotient as $q = (x - r)y^{-1}$.

Using **sage**, we can show that:

```
F = GF(8444461749428370424248824938781546531375899335154063827935233455917409239041)
F(7) == F(3 * 2 + 1) # True
F(7) == F(3 * 2 + 2) # False
new_q = F(7 - 2) / F(3)
F(7) == F(3 * new_q + 2) # True
```

Note that exploitation of the flawed gadget seems limited in practice, due to additional checks on the callers' sides. There's currently only one call site to `AmountVar::quo_rem` in the codebase, which is in `PenaltyVar::apply_to`, and which does not appear to make use of the resulting remainder (only the quotient).

**Recommendations**. Both $q$ and $d$ cannot be larger than 64 bits since $x$ is constrained to 128 bits. Thus, constraining that $q$ is 128-bit, and that either $q$ or $d$ is 64-bit, is enough to ensure that there are no overflows.

**Client Response**. Penumbra implemented the fix by constraining the quotient $q$ to 128-bits, and adding an extra constraint ensureing that either $q$ or $d$ has a size of 64 bits.

## # 06 - Mixing runtime and compile-time logic

circuits

Informational

The arkworks R1CS library works by mixing compile-time (setup) with runtime (proving) logic. That is, users of the library must write code that will handle both pathways.

To do this, a user must first declare a structure associated with the circuit, and then write its circuit by implementing the `ConstraintSynthesizer` trait. For example, the proof associated with proving Merkle tree membership is defined with the following struct and trait implementation:

```
struct MerkleProofCircuit {
    state_commitment_proof: tct::Proof,
    pub anchor: tct::Root,
    pub epoch: Fq,
    pub block: Fq,
    pub commitment_index: Fq,
}

impl ConstraintSynthesizer<Fq> for MerkleProofCircuit {
    fn generate_constraints(
        self,
        cs: ConstraintSystemRef<Fq>,
    ) -> ark_relations::r1cs::Result<()> {
        // TRUNCATED...
    }
}
```

The struct can mix both compile-time values as well as runtime values. It would be much clearer if runtime values were encapsulated in `Option` types. This way, you wouldn't need to give them any value when compiling the circuit, and you would be forced to give them a non-default value when proving (as runtime code would need to `unwrap` these values).

Furthermore, logic in the circuit is also mixed between compile-time and runtime. For example, in the `U128x128Var::checked_div` function some values are computed *out of circuit* and then witnessed and constrained in the circuit:

```
let xbar_ooc = self.value().unwrap_or_default();
let ybar_ooc = rhs.value().unwrap_or(U128x128::from(1u64));
let Ok((quo_ooc, rem_ooc)) = stub_div_rem_u384_by_u256(xbar_ooc.0, ybar_ooc.0) else {
    return Err(SynthesisError::Unsatisfiable);
};

// TRUNCATED...

let q = U128x128Var::new_witness(cs.clone(), || Ok(U128x128(quo_ooc)))?;
```

Mixing out-of-circuit code with in-circuit code is error-prone, as was shown in **Double spending due to incoming viewing key (ivk) derivation not constrained**. While this pattern is inevitable, due to the nature of the library, it

could be made safer by using `Option` types and by making the code more explicit.

That is, no default values should be used, and instead out-of-circuit code should happen in a monadic way as much as possible (using `Option`s).

So, for example, the previous code could be rewritten as:

```
let (quo_val, rem_val) = (!cs.in_setup_mode())
    .then(|| {
        let x_bar = self.value().unwrap();
        let y_bar = rhs.value().unwrap();
        stub_div_rem_u384_by_u256(x_bar.0, y_bar.0)
            .map_err(|_| SynthesisError::Unsatisfiable)?
    )
    .unzip();
};

// TRUNCATED...

let q = U128x128Var::new_witness(cs.clone(), || Ok(U128x128(quo_val.unwrap())))?;
```

or even more concisely by using the closure passed to `new_witness`:

```
let q = U128x128Var::new_witness(cs.clone(), || {
    let x_bar = self.value().unwrap();
    let y_bar = rhs.value().unwrap();
    let (q_bar, r_bar) = stub_div_rem_u384_by_u256(x_bar.0, y_bar.0)
        .map_err(|_| SynthesisError::Unsatisfiable)?;
    Ok(q_bar)
})?;
```

In addition, new circuit values are allocated using the arkworks function `AllocVar::new_variable`. As such, custom user types must implement this function themselves. Its signature takes a closure `f`, which is supposed to contain a recipe on how to compute the witness value associated with the circuit variable at runtime.

```
pub trait AllocVar<V, F: Field>
where
    Self: Sized,
    V: ?Sized,
{
    fn new_variable<T: Borrow<V>>(
        cs: impl Into<Namespace<F>>,
        f: impl FnOnce() -> Result<T, SynthesisError>,
        mode: AllocationMode,
    ) -> Result<Self, SynthesisError>;
```

In many custom implementations of `AllocVar::new_variable`, it was observed that the closure `f` was called/executed even when running the setup phase (not just during witness generation/proving). This is an ill pattern as the closure can contain logic that is not supposed to be executed in setup mode. See the previous refactoring recommendation for an example.

For example, in the following code, the closure `f` is executed even in setup mode:

```rust
impl AllocVar<Note, Fq> for NoteVar {
    fn new_variable<T: std::borrow::Borrow<Note>>(
        cs: impl Into<ark_relations::r1cs::Namespace<Fq>>,
        f: impl FnOnce() -> Result<T, SynthesisError>,
        mode: ark_r1cs_std::prelude::AllocationMode,
    ) -> Result<Self, SynthesisError> {
        let ns = cs.into();
        let cs = ns.cs();
        let note1 = f()?; // <---- executed even in setup mode!
        let note: &Note = note1.borrow();
        let note_blinding =
            FqVar::new_variable(cs.clone(), || Ok(note.note_blinding().clone()), mode)?;
```

## # 07 - Unoptimized balance computation

core/balance

Informational

**Description**. In `crates/core/asset/src/balance.rs`, the `BalanceVar::commit` gadget conditionally computes a commitment to the balance, depending on the sign of the amount. The code is as follows:

```
// We scalar mul first with value (small), then negate [v]G_v if needed
let commitment_plus_contribution =
    commitment.clone() + G_v.scalar_mul_le(value_amount.to_bits_le()?.iter())?;
let commitment_minus_contribution =
    commitment - G_v.scalar_mul_le(value_amount.to_bits_le()?.iter())?;
commitment = ElementVar::conditionally_select(
    sign,
    &commitment_plus_contribution,
    &commitment_minus_contribution,
)?;
```

Here `to_bits_le` is called twice, which is going to increase the size of the witness twice instead of once. Not only calling it once should improve the number of constraints, but the second scalar multiplication should be avoidable altogether by computing $\mathrm{P} = [\mathrm{v}]\mathrm{G}$ and then the negated point directly as $-\mathrm{P}$.

## # 08 - Unnecessary constraining output values

circuits

Informational

**Description**. Gadgets are used in different places, sort of like subcircuits, to abstract parts of the logic. When gadgets return an output value, the value is automatically being allocated in the witness. It is thus redundant to reallocate it on the caller side and enforce that the two values match.

This pattern has been seen several times in the codebase, and increase the size of the circuit for no clear benefits. For example, in `crates/core/component/stake/src/penalty.rs` the result of the call to `AmountVar::quo_rem` is constrained to be equal to a free witness variable `penalized_amount_var`.

```
impl PenaltyVar {
    pub fn apply_to(&self, amount: AmountVar) -> Result<AmountVar, SynthesisError> {
        // TRUNCATED...

        // Out of circuit penalized amount computation:
        let amount_bytes = &amount.value().unwrap_or(Amount::from(0u64)).to_le_bytes()
[0..16];
        let amount_128 =
            u128::from_le_bytes(amount_bytes.try_into().expect("should fit in 16 bytes"));
        let penalized_amount = amount_128 * (1_0000_0000 - penalty.0 as u128) / 1_0000_0000;

        // Witness the result in the circuit.
        let penalized_amount_var = AmountVar::new_witness(self.cs(), || {
            Ok(Amount::from(
                u64::try_from(penalized_amount).expect("can fit in u64"),
            ))
        })?;

        // TRUNCATED...

        let (penalized_amount_quo, _) = numerator.quo_rem(&hundred_mil)?;
        penalized_amount_quo.enforce_equal(&penalized_amount_var)?;
```

Note that this particular example is not necessarily a good one, as the final `enforce_equal` is actually not benign since the call to `AmountVar::new_witness` will constrain `penalized_amount_var` to be 128 bits, which will constrain the quotient computed by `AmountVar::quo_rem` to be 128-bit as well.

So in practice, while this specific check seems redundant, it helps mitigating the issue described in **Unsound Division of AmountVar**. But since the unsound division issue should be fixed, this check could be removed as well.

## # 09 - Avoid user-controlled inputs when they can be recomputed

### action handlers

Informational

**Description**. In order to handle actions in user transactions, action handlers are implemented for each action. These action handlers are in charge of doing stateless checks as well as stateful checks, where the former checks usually verify zero-knowledge proofs when present, and the latter require reading the state of the blockchain.

One pattern was noticed during the audit: some values in the body of the action (provided by the user) can sometimes appear redundant.

For example, the body for a delegator vote action contains an *unbounded amount* value, which is provided by the user as the amount their delegated note would have unbounded to in the Penumbra token (if they had unbounded the note at the time). Importantly, the stateful check of the delegator vote action handler will recompute this value (based on the value of the note and historical information about the validator delegation rate at the time the note was delegated).

```
impl ActionHandler for DelegatorVote {
    // TRUNCATED...

    async fn check_stateful<S: StateRead + 'static>(&self, state: Arc<S>) -> Result<()> {
        let DelegatorVote {
            // TRUNCATED...
            body:
                DelegatorVoteBody {
                    // TRUNCATED...
                    value,
                    unbonded_amount,
                },
        } = self;

        // TRUNCATED...

        state
            .check_unbonded_amount_correct_exchange_for_proposal(*proposal, value,
unbonded_amount)
            .await?;
```

In general this is a dangerous and error-prone pattern, as any user-controlled value is potentially malicious. Bugs can arise easily, when forgetting to verify such redundant values. When values can be computed by the validator themselves, it is better to do so.

## # 0a - Unnecessary constraints in Amount::quo_rem

core/num

Informational

**Description**. In the `Amount::quo_rem` function, the following checks are being enforced in the gadget:

```
zero_var
    .amount
    .enforce_cmp(&rem_var.amount, core::cmp::Ordering::Less, true)?;
rem_var
    .amount
    .enforce_cmp(&divisor_var.amount, core::cmp::Ordering::Less, false)?;
divisor_var.enforce_not_equal(&zero_var)?;
```

The first check is redundant, as it is merely checking that `rem_var` is not negative (that is is strictly less than $(\mathrm{p}-1)/2$), but we're already doing this in the second line as `enforce_cmp` will check that both operand are non-negative.

In addition, the last check that $\mathrm{d} \neq 0$ is also unnecessary, as it is already implied by the previous check that $0 \leq \mathrm{r} < \mathrm{d}$.