Zellic

March 5, 2024

Prepared for
Erwan Or
Penumbra Labs

Prepared by
William Bowling
Ayaz Mammadov
Avraham Weinstock
Zellic

Penumbra

Smart Contract Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Penumbra Labs from January 29nd to March 1st, 2024. During this engagement, Zellic reviewed Penumbra's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Are the zero-knowledge circuits properly integrated into the shielded pool, staking component, and DEX?
- Does the value-balance mechanism prevent value from being incorrectly created/inflated?
- Can the mechanism for delegating stake be manipulated?
- Is the DEX sufficiently isolated to be unable to lose funds of users that do not interact with it?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- The general correctness of code outside of `crates/core/component/{shielded-pool,stake,dex}`
- The correctness of the zero-knowledge circuits themselves, including, but not limited to, any potential underconstraints or integer overflows in the lowering of various data types to R1CS
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

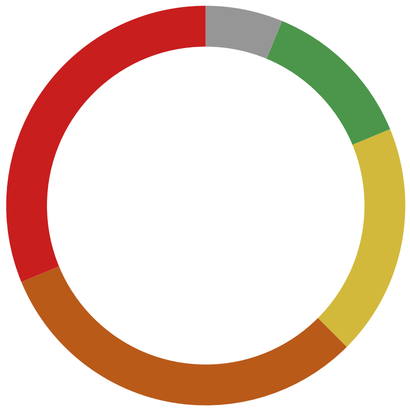
1.4. Results

During our assessment on the scoped Penumbra crates, we discovered 16 findings. Five critical issues were found. Five were of high impact, three were of medium impact, two were of low impact, and the remaining finding was informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for Penumbra Labs's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	5
🟧 High	5
🟨 Medium	3
🟩 Low	2
⬜ Informational	1



2. Introduction

2.1. About Penumbra

Penumbra is a fully private proof-of-stake network and decentralized exchange.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the crates.

Non-Determinism. Non-Determinism is a leading class of security issues on Cosmos. It can lead to consensus failure and blockchain halts. This includes, but is not limited to, vectors like wall clock times, and map iteration.

Arithmetic Issues. This includes, but is not limited to, integer overflows and underflows, floating point associativity issues, and loss of precision and unfavorable integer rounding.

Denial of Service. Denial of Service attacks are another leading issue in Cosmos projects. Issues including, but not limited to, unhandled panics, unbounded computations, and incorrect error handling can potentially lead to consensus failures.

Architecture Risk. Encompasses potential hazards originating from the blueprint of a system, which involves its core validation mechanism and other architecturally significant constituents influencing the system fundamental security attributes, presumptions, trust mode and design.

Implementation Risk. Encompasses risks linked to translating a system's specification into practical code. Constructing a custom system involves developing intricate on-chain and off-chain elements while accommodating the idiosyncrasies and challenges presented by distinct programming languages, frameworks, and execution environments.

Cross-chain functionality. Cross-chain functionality is much more complex than code limited to a single chain. This is because a smart contract on any given chain has limited visibility outside of that single chain. Cross-chain functionality opens the door to problems like race conditions and validator network attacks. In the past, several major DeFi hacks have occurred due to erroneous cross-chain code. Thus, Zellic enumerates and evaluates the presence of cross-chain functionality in a project as it often contributes to the project's risk profile.

Test suite and code coverage. We review the comprehensiveness of the project's test suite and code coverage. Untested code is error-prone and typically presents a security risk in general. On the other hand, certain testing practices like generative tests (like fuzzing), or property-based tests greatly improve the quality of a test suite and help mitigate security risks.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped crates itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Penumbra Crates

Repository	https://github.com/penumbra-zone/penumbra ↗
Version	penumbra: `efe8c79112416771db6f53c24ddeb68f97f77591`
Programs	• crates/core/component/shielded-pool/src/* • crates/core/component/stake/src/* • crates/core/component/dex/src/*
Type	Rust
Platform	Cosmos-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of 11.25 person-weeks. The assessment was conducted over the course of five calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

William Bowling
Engineer
vakzz@zellic.io

Ayaz Mammadov
Engineer
ayaz@zellic.io

Avraham Weinstock
Engineer
avi@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 29, 2024	Start of primary review period
January 31, 2024	Kick-off call
March 1, 2024	End of primary review period

3. Detailed Findings

3.1. Incorrect ICS-20 balance on time-out

Target	shielded-pool/src/component/transfer.rs		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When an ICS-20 transfer is attempted but not received by the destination chain in time, a time-out packet is sent back to the source chain so that the funds can be returned:

```
async fn timeout_packet_inner<S: StateWrite>(mut state: S, msg: &MsgTimeout)
    -> Result<()> {
     // ... snip ...
    if is_source(&msg.packet.port_on_a, &msg.packet.chan_on_a, &denom, true) {
        // sender was source chain, unescrow tokens back to sender
        // ... snip ...
        state
            .mint_note(
                value,
                &receiver,
                CommitmentSource::Ics20Transfer {
                    packet_seq: msg.packet.sequence.0,
                    channel_id: msg.packet.chan_on_a.0.clone(),
                    sender: packet_data.sender.clone(),
                },
            )
            .await
            .context("couldn't mint note in timeout_packet_inner")?;
        // ... snip ...
        let new_value_balance =
                value_balance
                    .checked_sub(&withdrawal.amount)
                    .ok_or_else(|| {
                        anyhow::anyhow!("underflow subtracting value balance
    in ics20 withdrawal")
                    })?;
        state.put(
            state_key::ics20_value_balance(&msg.packet.chan_on_a,
    &denom.id()),
            new_value_balance,
        );
    } else {
        state
            .mint_note(
                value,
                &receiver,
                // NOTE: should this be Ics20TransferTimeout?
                CommitmentSource::Ics20Transfer {
                    packet_seq: msg.packet.sequence.0,
                    channel_id: msg.packet.chan_on_a.0.clone(),
                    sender: packet_data.sender.clone(),
                },
            )
            .await
            .context("failed to mint return voucher in ics20 transfer
    timeout")?;
    }
    Ok(())
}
```

In the first branch, the funds are minted back to the sender and the ICS-20 balance for the denom is reduced. In the else block, where the denom was being returned to the source chain, the funds are minted back to the sender but the ICS-20 balance is not updated, causing it to be lower than it should be.

Impact

A malicious user could exploit this by continually transferring out a small amount of a popular denom from an external chain with a low time-out height, causing the ICS-20 balance to be reduced to zero. This would prevent anyone holding that denom in Penumbra from being able to transfer it back to the original chain.

Recommendations

The ICS-20 balance for the denom should be increased by `withdrawal.amount` after the funds are minted back to the sender.

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in commit b1b1051c ↗.

3.2. Arbitrary balance via dummy spend

Target	shielded-pool/src/spend/proof.rs		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When creating a spend proof for a note with zero value, a dummy flag is used so that certain checks in the circuit are skipped:

```
// Public inputs
let anchor_var = FqVar::new_input(cs.clone(),
    || Ok(Fq::from(self.public.anchor)))?;
let claimed_balance_commitment_var =
    BalanceCommitmentVar::new_input(cs.clone(),
    || Ok(self.public.balance_commitment))?;
let claimed_nullifier_var =
    NullifierVar::new_input(cs.clone(), || Ok(self.public.nullifier))?;
let rk_var = RandomizedVerificationKey::new_input(cs.clone(),
    || Ok(self.public.rk))?;

// We short circuit to true if value released is 0. That means this is a
    _dummy_ spend.
let is_dummy = note_var.amount().is_eq(&FqVar::zero())?;
// We use a Boolean constraint to enforce the below constraints only if this is
    not a
// dummy spend.
let is_not_dummy = is_dummy.not();

// Note commitment integrity.
let note_commitment_var = note_var.commit()?;
note_commitment_var.conditional_enforce_equal(&claimed_note_commitment,
    &is_not_dummy)?;

// Nullifier integrity.
let nullifier_var = NullifierVar::derive(&nk_var, &position_var,
    &claimed_note_commitment)?;
nullifier_var.conditional_enforce_equal(&claimed_nullifier_var,
    &is_not_dummy)?;

// ... snip ...
// Check integrity of balance commitment.
let balance_commitment = note_var.value().commit(v_blinding_vars)?;
balance_commitment
    .conditional_enforce_equal(&claimed_balance_commitment_var,
    &is_not_dummy)?;
```

The issue is that there is no way for the action handler to know if the note is a dummy spend or not, so if the proof passed, then the supplied balance commitment and nullifier are assumed to be valid.

Impact

Arbitrary generation of funds

As a dummy spend can always have a valid proof generated and validated, an arbitrary balance commitment can be used to generate any amount of funds.

We created the following simple POC to demonstrate the issue,

```
TxCmd::Pwn => {
    let mut planner = Planner::new(OsRng);
    let view: &mut dyn ViewClient = app
        .view
        .as_mut()
        .context("view service must be initialized")?;
    let self_address = view.address_by_index(AddressIndex::new(0)).await?;
    let rseed = Rseed::generate(&mut OsRng);
    let note = Note::from_parts(
        self_address,
        Value {
            amount: 0u64.into(),
            asset_id: *STAKING_TOKEN_ASSET_ID,
        },
        rseed,
    )?;

    let plan = planner.spend(
        note,
        0.into(),
    ).plan(
            app.view
                .as_mut()
                .context("view service must be initialized")?,
            AddressIndex::new(0),
        )
```

```
            .await
            .context("can't build send transaction")?;
        app.build_and_submit_transaction(plan).await?;
    }

 // ... snip ...
impl SpendPlan {
    // ... snip ...
    pub fn balance(&self) -> Balance {
        Value {
            amount: 1000000000000u64.into(),
            asset_id: self.note.value().asset_id,
        }
        .into()
    }
```

which, when submitted, will increase the account's balance by `1000000000000upenumbra`:

```
Balance before:
1001000100.069071penumbra

broadcasting transaction and awaiting confirmation...
transaction broadcast successfully: 8aef5150720af7c11dfc3d06e258684808874b80a6
c770e08ece5e756bfab54d

Balance after:
1002000100.069071penumbra

pcli view tx 8aef5150720af7c11dfc3d06e258684808874b80a6c770e08ece5e756bfab54d
Fee: 0
Expiration Height: 0
Memo Sender: penumbra15xugeart3zu820r2cxjx5fly43zzdkhzgapfmwgcdfs9wn6temxrfzla
v88cczw3cp34q4pzydlfffeqjtyx24peys53cgpln103pe2aacuwe0pg8qgnuchjfysmdze35awq5g
Memo Text:


 Tx Action Description
 Spend
 Output 1000000penumbra -> [account 0]
```

Nullifier griefing

Additionally, since a dummy spend will always pass the proof verification, it is possible for an attacker or malicious validator with access to the mempool to see a pending action, create a dummy spend using the legitimate nullifier, and try to submit it before the original action is included. If they win the race the nullifier will be marked as spent and the user's funds will be lost.

```
impl ActionHandler for Spend {
    type CheckStatelessContext = TransactionContext;
    async fn check_stateless(&self, context: TransactionContext) -> Result<()>
    {
        // ... snip ...
        // 3. Check that the proof verifies.
        let public = SpendProofPublic {
            anchor: context.anchor,
            balance_commitment: spend.body.balance_commitment,
            nullifier: spend.body.nullifier,
            rk: spend.body.rk,
        };
        spend
            .proof
            .verify(&SPEND_PROOF_VERIFICATION_KEY, public)
            .context("a spend proof did not verify")?;
        Ok(())
    }

    async fn check_stateful<S: StateRead + 'static>(&self, state: Arc<S>) ->
    Result<()> {
        // Check that the `Nullifier` has not been spent before.
        let spent_nullifier = self.body.nullifier;
        state.check_nullifier_unspent(spent_nullifier).await
    }

    async fn execute<S: StateWrite>(&self, mut state: S) -> Result<()> {
        let source = state.get_current_source().expect("source should be
    set");
        state.nullify(self.body.nullifier, source).await;
        // ... snip ...
```

Recommendations

When dealing with dummy spends, only the Merkle path validity constraint should be skipped (see sections 4.8.2, "Dummy Notes (Sapling)", and 4.17.2, "Spend Statement (Sapling)", in the Zcash protocol specification at https://zips.z.cash/protocol/protocol.pdf ↗).

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in commit `abbe262f` ↗.

3.3. Asset total supply can be inflated

Target	shielded-pool/src/component/transfer.rs		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When local funds are transferred to an external chain, the ICS-20 balance for that denom is updated, but the total token supply does not change.

```
pub trait Ics20TransferWriteExt: StateWrite {
    async fn withdrawal_execute(&mut self, withdrawal: &Ics20Withdrawal) ->
    Result<()> {
        // create packet, assume it's already checked since the component
    caller contract calls `check` before `execute`
        let checked_packet =
    IBCPacket::<Unchecked>::from(withdrawal.clone()).assume_checked();

        let prefix = format!("transfer/{}/", &withdrawal.source_channel);
        if !withdrawal.denom.starts_with(&prefix) {
            // we are the source. add the value balance to the escrow channel.
            let existing_value_balance: Amount = self
                .get(&state_key::ics20_value_balance(
                    &withdrawal.source_channel,
                    &withdrawal.denom.id(),
                ))
                .await
                .expect("able to retrieve value balance in ics20 withdrawal!
    (execute)")
                .unwrap_or_else(Amount::zero);

            let new_value_balance = existing_value_balance +
    withdrawal.amount;
            self.put(
                state_key::ics20_value_balance(&withdrawal.source_channel,
    &withdrawal.denom.id()),
                new_value_balance,
            );
```

When the tokens are returned to Penumbra, the ICS-20 balance is checked, the funds are minted, and the ICS-20 balance is reduced.

```
// 2. check if we are the source chain for the denom.
if is_source(&msg.packet.port_on_a, &msg.packet.chan_on_a, &denom, false) {
    // mint tokens to receiver in the amount of packet_data.amount in the denom
    of denom (with
    // the source removed, since we're the source)
    // ... snip ...

    // check if we have enough balance to unescrow tokens to receiver
    let value_balance: Amount = state
        .get(&state_key::ics20_value_balance(
            &msg.packet.chan_on_b,
            &unprefixed_denom.id(),
        ))
        .await?
        .unwrap_or_else(Amount::zero);

    if value_balance < receiver_amount {
        // error text here is from the ics20 spec
        anyhow::bail!("transfer coins failed");
    }

    state
        .mint_note(
            value,
            &receiver_address,
            CommitmentSource::Ics20Transfer {
                packet_seq: msg.packet.sequence.0,
                // We are chain A
                channel_id: msg.packet.chan_on_a.0.clone(),
                sender: packet_data.sender.clone(),
            },
        )
        .await
        .context("unable to mint note when receiving ics20 transfer packet")?;
```

The issue is that `mint_note` will call `increase_token_supply`, which will update the total supply for the asset, even though it was never decreased when the tokens were transferred out, causing the total supply to be higher than it should be.

Impact

A malicious user could exploit this by transferring out delegation tokens for a validator and returning them to increase the total supply, which in turn would increase the voting power of the validator as it is based on the total number of delegation tokens. Repeatedly performing this exploit could allow the validator to gain enough voting power to perform governance actions.

Recommendations

When transferring assets out of Penumbra, `decrease_token_supply` should be called so that it correctly represents the current number of tokens in the system. Alternatively, `increase_token_supply` should not be called when the source tokens are returned.

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in pull request 4020 ↗.

3.4. Delegation tokens can be forged

Target	stake/src/delegation_token.rs		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When a delegate is voting on a proposal, their voting power is determined by the amount of delegation tokens they hold for the validator. The tokens have a denom in the format of `udelegation_(?P<data>penumbravalid1[a-zA-HJ-NP-Z0-9]+)` where the final part represents the validator identity.

The validator they are voting for as well as the voting power is determined by the asset used when voting:

```
async fn check_unbonded_amount_correct_exchange_for_proposal(
    &self,
    proposal_id: u64,
    value: &Value,
    unbonded_amount: &Amount,
) -> Result<()> {
    let validator_identity =
    self.validator_by_delegation_asset(value.asset_id).await?;

 // ... snip ...

/// Look up the validator for a given asset ID, if it is a delegation token.
async fn validator_by_delegation_asset(&self, asset_id: asset::Id) ->
    Result<IdentityKey> {
    // Attempt to find the denom for the asset ID of the specified value
    let Some(denom) = self.denom_by_asset(&asset_id).await? else {
        anyhow::bail!("asset ID {} does not correspond to a known denom",
    asset_id);
    };

    // Attempt to find the validator identity for the specified denom, failing
    if it is not a
    // delegation token
    let validator_identity = DelegationToken::try_from(denom)?.validator();
    Ok(validator_identity)
}
```

```
 // ... snip ...

impl TryFrom<asset::DenomMetadata> for DelegationToken {
type Error = anyhow::Error;
fn try_from(base_denom: asset::DenomMetadata) -> Result<Self, Self::Error> {
    // Note: this regex must be in sync with both asset::REGISTRY
    // and VALIDATOR_IDENTITY_BECH32_PREFIX
    let validator_identity =
        Regex::new("udelegation_(?P<data>penumbravalid1[a-zA-HJ-NP-Z0-9]+)")
            .expect("regex is valid")
            .captures(&base_denom.to_string())
```

The issue is that there is no caret (`^`) at the start of the regex when determining extracting the validator's identity, causing the regex to match anywhere in the denom instead of only the start.

Impact

A malicious user could transfer funds into Penumbra from an external chain with a denom such as `udelegation_penumbravalid17geftyyhx73w4hns03gwyfrgfdwlde04083udj57qj6s277ndqpskd`. This would end up being prefixed in Penumbra with `transfer/channel-X/` but would still match the regex.

This fake token could then be used to vote on proposals with any amount of voting power and for any validator.

Recommendations

The regex should be updated to include a caret (`^`) at the start to ensure that it only matches the start of the denom.

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in commit `0d173cf1` ↗.

3.5. Multiple positions with the same ID

Target	dex/src/component/action_handler/position/open.rs		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When opening a new position, the `check_stateful` handler ensures that there is not an existing position with the same ID:

```
async fn check_stateful<S: StateRead + 'static>(&self, state: Arc<S>) ->
    Result<()> {
    // Validate that the position ID doesn't collide
    state.check_position_id_unused(&self.position.id()).await?;

    Ok(())
}
```

As all the `check_stateful` checks are run in parallel (see 4.1. ↗), if two positions are opened in the same transaction, then it is possible to open the same position multiple times. This allows one to receive two or more NFTs for the same position ID, even though they may have opened them with different reserves.

Impact

A malicious user could exploit this bug to withdraw more than the initial reserves through the following:

1. In the same transaction, open two identical positions but with the first having a reserve of `1upenumbra` and the other `100penumbra`.
2. They now have two NFTs for the position, but the reserve for the position has been set to 100penumbra.
3. Close both positions, and they will now have two NFTs for the closed position.
4. Withdraw the position, due to the bug in `handle_limit_order` (see 3.11. ↗); the position will be set to `Closed` instead of `Withdrawn`.
5. Withdraw the position again, and they will now have withdrawn 200penumbra.

Recommendations

When adding a new position to the state, there should be a check to ensure that the position ID has not already been used.

Remediation

This issue has been acknowledged by Penumbra Labs, and fixes were implemented in the following commits:

- `1f084185` ↗
- `2949f8f0` ↗

See also 4.1. ↗.

3.6. IBC time-out packet is fallible

Target	shielded-pool/src/component/transfer.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

If a user tries to transfer funds to an external chain via IBC, a time-out packet can be triggered if it is not received by the destination chain in time:

```
async fn timeout_packet_execute<S: StateWrite>(mut state: S, msg: &MsgTimeout)
    {
    // timeouts should never fail
    timeout_packet_inner(&mut state, msg)
        .await
        .expect("able to timeout packet");
}
```

The `timeout_packet_inner` will end up calling `mint_note->increase_token_supply`:

```
async fn increase_token_supply(
    &mut self,
    asset_id: &asset::Id,
    amount_to_add: Amount,
) -> Result<()> {
    let key = state_key::token_supply(asset_id);
    let current_supply:
    Amount = self.get(&key).await?.unwrap_or(0u128.into());

    tracing::debug!(
        ?current_supply,
        ?amount_to_add,
        ?asset_id,
        "increasing token supply"
    );
    let new_supply = current_supply.checked_add(&amount_to_add).ok_or_else(||
    {
        anyhow::anyhow!(
            "overflow updating token {} supply {} with delta {}",
            asset_id,
            current_supply,
            amount_to_add
        )
    })?;

    self.put(key, new_supply);
    Ok(())
}
```

The issue is that in between when the tokens are transferred and when the time-out packet occurs, it is possible for the token supply to change. A malicious user could transfer in enough tokens that would cause the total supply to overflow when the time-out handler attempts to mint the tokens, triggering a panic and crashing the node.

Impact

A malicious user could cause the node to crash by performing the following steps:

1. A user on an IBC-connected Chain B sends `2^128-1` coins to Penumbra; `mint_note` increases the total supply to `2^128-1`.
2. The user then returns the `2^128-1` coins from Penumbra the original chain with a time-out height that is about to expire / already expired, decreasing the total supply to zero.
3. A user on an IBC-connected Chain B sends one more coin to Penumbra; `mint_note` increases the total supply to 1.
4. The time-out handler for Step 2 is called, and `mint_note` fails as the total supply is now > `u128`.

Recommendations

An error should be returned instead of triggering a panic.

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in commit 15d81099 ↗.

3.7. Division by zero in `SwapExecution::max_price`

Target	crates/core/component/dex/src/swap_execution.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

The function `Dex::end_block` panics as a result of the call to `arbitrage`, returning an error when given a set of six positions involving five assets that trigger a zero division in `SwapExecution::max_price`.

The relevant positions are constructed in the following test case:

```
#[tokio::test]
async fn zero_division_with_six_positions() {
    use cnidarium_component::Component;
    use penumbra_sct::{component::clock::EpochManager, epoch::Epoch};
    use std::str::FromStr;
    use tendermint::abci;
    let asset_a = asset::Id::from_str(
        "passet1mv4dg744vmefu3azks3cljxzpnkux9nt778pu9n9njdql94r6u8qe90s6q"
    ).unwrap();
    let asset_b = asset::Id::from_str(
        "passet1984fctenw8m2fpl8a9wzguzp7j34d7vravryuhft808nyt9fdggqxmanqm"
    ).unwrap();
    let asset_c = asset::Id::from_str(
        "passet1z9kse4k7jdyudph4sqhhexqhnecdfzv28rsuc00exuzujn55r5xqjlyc3p"
    ).unwrap();
    let asset_d = asset::Id::from_str(
        "passet1r4kcf2m4r92jqmdks5c9yt7v2tgnht4aj3fml4qln56x72nm8qrsm9d598"
    ).unwrap();
    let asset_e = asset::Id::from_str(
        "passet15e489q49rlsr9lk0ec76cclfh8d962lls5p072jrf42dsthhjqxq6xmlfn"
    ).unwrap();
    let mut rng = rand_chacha::ChaChaRng::seed_from_u64(1312);
    let positions = vec![
        Position::new(&mut rng, DirectedTradingPair::new(asset_a, asset_b),
            0,
            Amount::from(149389448249173150936496u128),
            Amount::from(208450392905823001306740u128),
            Reserves {
                r1: Amount::from(0u64),
                r2: Amount::from(1674038047566237470u64), }
        ),
        Position::new(&mut rng, DirectedTradingPair::new(asset_c, asset_b),
            0,
            Amount::from(1148249165388158158lu128),
            Amount::from(2296498330776316316 3u128),
            Reserves {
                r1: Amount::from(6428821468204u64),
                r2: Amount::from(3997249009584709369u64), }
        ),
        Position::new(&mut rng, DirectedTradingPair::new(asset_c, asset_a),
            593,
            Amount::from(369914451723480732836066u128),
            Amount::from(253592161842809128314442u128),
            Reserves {
                r1: Amount::from(0u64),
                r2: Amount::from(3346911693977u64), }
        ),
        Position::new(&mut rng, DirectedTradingPair::new(asset_d, asset_a),
            0,
            Amount::from(461882542585266790278356u128),
            Amount::from(639568646294864146743u128),
            Reserves {
                r1: Amount::from(0u64),
                r2: Amount::from(233587891225428463u64), }
        ),
        Position::new(&mut rng, DirectedTradingPair::new(asset_d, asset_a),
            4848,
            Amount::from(1445730096413720996991u128),
            Amount::from(864162800629363471837189u128),
            Reserves {
                r1: Amount::from(323449097135647u64),
                r2: Amount::from(0u64), }
        ),
        Position::new(&mut rng, DirectedTradingPair::new(asset_d, asset_e),
            0,
            Amount::from(1u128),
            Amount::from(1u128),
            Reserves {
                r1: Amount::from(0u64),
                r2: Amount::from(1u64), }
        ),
    ];

    let storage = TempStorage::new().await.unwrap();
    let mut state = Arc::new(StateDelta::new(storage.latest_snapshot()));
    let height = 1;

    {
        let mut state_tx = state.try_begin_transaction().unwrap();
        state_tx.put_epoch_by_height(
            height,
            Epoch {
                index: 0,
                start_height: 0,
            },
        );
        state_tx.put_block_height(height);
        state_tx.apply();
    }
    {
        let mut state_tx = state.try_begin_transaction().unwrap();
        for pos in positions {
            state_tx.put_position(pos).await.unwrap();
        }
        state_tx.apply();
    }
    let end_block = abci::request::EndBlock {
        height: height.try_into().unwrap(),
    };
    crate::component::Dex::end_block(&mut state, &end_block).await;
}
```

The relevant excerpt of the stack trace is

```
5: penumbra_dex::swap_execution::SwapExecution::max_price
  at ./src/swap_execution.rs:27:21
6: penumbra_dex::component::router::route_and_fill::RouteAndFill::rout
    e_and_fill::{{closure}}::{{closure}}
  at ./src/component/router/route_and_fill.rs:277:44
7: penumbra_dex::component::router::route_and_fill::RouteAndFill::rout
    e_and_fill::{{closure}}
  at ./src/component/router/route_and_fill.rs:156:5
8: <core::pin::Pin<P> as core::future::future::Future>::poll
  at /rustc/79e9716c980570bfd1f666e3b16ac583f0168962/library/core/src/
    future/future.rs:125:9
9: penumbra_dex::component::arb::Arbitrage::arbitrage::{{closure}}::{{
    closure}}
  at ./src/component/arb.rs:63:14
10: penumbra_dex::component::arb::Arbitrage::arbitrage::{{closure}}
  at ./src/component/arb.rs:22:5
```

```
11:  <core::pin::Pin<P> as core::future::future::Future>::poll
  at /rustc/79e9716c980570bfd1f666e3b16ac583f0168962/library/core/src/
    future/future.rs:125:9
12:  <penumbra_dex::component::dex::Dex as cnidarium_component::compone
    nt::Component>::end_block::{{closure}}::{{closure}}
  at ./src/component/dex.rs:102:14
```

Impact

Causing a panic in `end_block` will halt the chain, causing a denial of service. While this specific set of positions requires a high quantity of specific tokens that are whitelisted for arbitrage to trigger the panic, it is possible that the root cause is triggerable with a smaller amount of tokens, as this set of positions was extracted from proptest with a reduction strategy that reduced the values in the positions independently after reducing the number of positions, which is not guaranteed to find a global minimum value market that triggers the same panic, only to find a local minimum.

Recommendations

If the successful completion of `arbitrage` in `Dex::end_block` is not required for security, only as an optimization, `Dex::end_block` should log its result on error instead of panicking. Additionally, consider whether `route_and_fill` can handle `execution.max_price()` returning `Err` more locally (e.g., whether it makes sense to close the erroring position and backtrack here).

Remediation

This issue has been acknowledged by Penumbra Labs, and fixes were implemented in the following commits:

- 6db483a4 ↗
- d79b7016 ↗

3.8. Duplicate `validator::Definitions` within a transaction

Target	staking/src/component/action_handler/validator_definition.rs		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

The action-handler system in Penumbra works by executing the checks in a sequential manner. The below demonstrates the execution of one transfer in N actions.

```
Action 1 -> check_stateless
Action 2 -> check_stateless
...
Action N -> check_stateless


Action 1 -> check_stateful
Action 2 -> check_stateful
...
Action N -> check_stateful


Action 1 -> execute
Action 2 -> execute
...
Action N -> execute
```

(See more about this in 4.1. ↗)

The issue arises when the side effects of the execution of an action invalidates an invariant that was checked for earlier.

Specifically, the `validator::Definition` action asserts that two validators with the same consensus key should not be uploaded, otherwise a tendermint could hang. This is done with the `check_stateful` implementation, which verifies that an existing validator with the same consensus key has not already been uploaded.

```
async fn check_stateful<S: StateRead + 'static>(&self, state: Arc<S>) ->
    Result<()> {
    ...
    // Check whether the consensus key has already been used by another
    validator.
    if let Some(existing_v) = state
        .get_validator_by_consensus_key(&v.validator.consensus_key)
        .await?
    {
        if v.validator.identity_key != existing_v.identity_key {
            ...
            // 2. If we submit a validator update to Tendermint that
            // includes duplicate consensus keys, Tendermint gets confused
            // and hangs.
            anyhow::bail!(
                "consensus key {:?} is already in use by validator {}",
                v.validator.consensus_key,
                existing_v.identity_key,
            );
        }
    }
}
```

A malicious attacker could submit a transaction with two `validator::Definition` actions; the `check_stateful` check will pass for both as no validator with that respective consensus key still exists. However, when the actions are executed, they will both upload validators with two of the same consensus key.

This is a time-of-check time-of-use bug (4.1. ↗).

Impact

Two validators uploaded with the same consensus key could cause a tendermint hang; this eventually could halt the chain.

Recommendations

An error should be returned when multiple `validator::Definition` actions in a transaction are submitted.

Remediation

This issue has been acknowledged by Penumbra Labs, and fixes were implemented in the following commits:

- 1f084185 ↗
- 2949f8f0 ↗

See also 4.1. ↗.

3.9. Swap claim proof panic

Target	./core/component/dex/src/swap_claim/proof.rs		
Category	Coding Mistakes	Severity	High
Likelihood	High	Impact	High

Description

The `verify` method of the `SwapClaim` proof ensures certain conditions with `expect` statements instead of returning an error. Many of these conditions are checks on user-provided input.

These `expects` are user-reachable; as such, a user could cause a panic and crash a node.

```
pub fn verify(
    &self,
    vk: &PreparedVerifyingKey<Bls12_377>,
    public: SwapClaimProofPublic,
) -> anyhow::Result<()> {
    let proof =
        Proof::deserialize_compressed_unchecked(&self.0[..]).map_err(|e|
    anyhow::anyhow!(e))?;

    let mut public_inputs = Vec::new();
    public_inputs.extend(
        Fq::from(public.anchor.0)
            .to_field_elements()
            .expect("Fq types are Bls12-377 field members"),
    );
    ...
}
```

Impact

Malicious attackers could supply inputs that could crash the node and cause a chain halt.

Recommendations

Change the `expects` to return `Results` instead to avoid these panics.

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in commit `fc9fbec7`↗.

3.10. Panic in `handle_batch_swaps` involving `ValueCircuitBreaker`

Target	crates/core/component/dex/src/component/router/route_and_fill.rs		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The call to `handle_batch_swaps` in `Dex::end_block` panics when given an open position and two swaps (one in each direction) for that position that seem to exercise some rounding-related behavior. The trading coefficients of the position imply an exchange rate of 1 of A for 1.2 of B, and a swap of 1 A for the current market price of B and a swap of 2 B for the current market price of A is given. This is minimized from an example where the implied exchange rate is approximately 22.9 (`{ p: 9726, q: 425 }`), and the swaps provide 1 A and 23 Bs.

```
TxCmd::SwapTest {} => {
    use penumbra_dex::{DirectedTradingPair, lp::{Reserves,
    position::Position}};
    let source = 0;
    let fvk = app.config.full_viewing_key.clone();
    let (claim_address, _dtk_d) =
        fvk.incoming().payment_address(AddressIndex::new(source));
    let cube = Value::from_str("1cube")?;
    let pizza = Value::from_str("1pizza")?;
    let pos = Position::new(OsRng, DirectedTradingPair::new(cube.asset_id,
    pizza.asset_id),
        0, Amount::from(6u64), Amount::from(5u64),
        Reserves { r1: Amount::from(0u64), r2: Amount::from(1u64), }
    );
    let plan = Planner::new(OsRng)
        .set_gas_prices(gas_prices)
        .set_fee_tier(FeeTier::Low.into())
        .position_open(pos)
        .swap(Value::from_str("1cube")?, pizza.asset_id, Fee::default(),
    claim_address)?
        .swap(Value::from_str("2pizza")?, cube.asset_id, Fee::default(),
    claim_address)?
        .plan(
            app.view
                .as_mut()
                .context("view service must be initialized")?,
            AddressIndex::new(source),
        )
        .await?;
    app.build_and_submit_transaction(plan).await?;
}
```

The relevant excerpt of the stack trace is

```
thread 'tokio-runtime-worker' panicked at /src/crates/core/component/d
    ex/src/component/router/route_and_fill.rs:128:9:
asset 1 outflow exceeds available balance
stack backtrace:
0: rust_begin_unwind
  at /rustc/82e1608dfa6e0b5569232559e3d385fea5a93112/library/std/src/p
    anicking.rs:645:5
1: core::panicking::panic_fmt
  at /rustc/82e1608dfa6e0b5569232559e3d385fea5a93112/library/core/src/
    panicking.rs:72:14
  2: penumbra_dex::component::router::route_and_fill::HandleBatchSwaps
    ::handle_batch_swaps::{{closure}}::{{closure}}
  at ./crates/core/component/dex/src/component/router/route_and_fill.r
    s:128:9
3: <tracing::instrument::Instrumented<T> as core::future::future::Futu
    re>::poll
  at /usr/local/cargo/registry/src/index.crates.io-6f17d22bba15001f/tr
    acing-0.1.40/src/instrument.rs:321:9
  4: penumbra_dex::component::router::route_and_fill::HandleBatchSwaps
    ::handle_batch_swaps::{{closure}}
  at ./crates/core/component/dex/src/component/router/route_and_fill.r
    s:27:5
5: <core::pin::Pin<P> as core::future::future::Future>::poll
  at /rustc/82e1608dfa6e0b5569232559e3d385fea5a93112/library/core/src/
    future/future.rs:125:9
6: <penumbra_dex::component::dex::Dex as cnidarium_component::componen
    t::Component>::end_block::{{closure}}::{{closure}}
  at ./crates/core/component/dex/src/component/dex.rs:64:18
7: <tracing::instrument::Instrumented<T> as core::future::future::Futu
    re>::poll
  at /usr/local/cargo/registry/src/index.crates.io-6f17d22bba15001f/tr
    acing-0.1.40/src/instrument.rs:321:9
8: <penumbra_dex::component::dex::Dex as cnidarium_component::componen
    t::Component>::end_block::{{closure}}
  at ./crates/core/component/dex/src/component/dex.rs:39:5
```

The patch that adds the property tests that found this and Finding 3.7. ↗ were provided to the client.

Impact

Causing a panic in `end_block` will halt the chain, causing a denial of service. This panic is easy to trigger with a modified PCLI command, requiring one token of any kind, and two tokens of any other kind distinct from the first. Additionally, since the panic is downstream of the root cause, it might be possible that if other positions are open, the value circuit breaker may not catch similar rounding issues, resulting in swaps happening at incorrect prices or value being extracted from the DEX.

Recommendations

The root cause of why either more value is getting swapped than is available or why the value circuit breaker is underestimating balance should be fixed.

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in commit `46a82a5f` ↗.

3.11. Limit orders can be erroneously closed

Target	dex/src/component/position_manager.rs		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

One option when opening a new position is to make it a limit order, causing it to be closed as soon as one of the reserves goes to zero:

```
fn handle_limit_order(
    &self,
    prev_position: &Option<position::Position>,
    position: Position,
) -> Position {
    let id = position.id();
    match prev_position {
        Some(_) if position.close_on_fill => {
            // It's technically possible for a limit order to be partially
    filled,
            // and unfilled on the other side. In this case, we would close it
    prematurely.
            // However, because of the arbitrage dynamics we expect that in
    practice an order
            // gets completely filled or not at all.
            if position.reserves.r1 == Amount::zero() || position.reserves.r2
    == Amount::zero()
            {
                tracing::debug!(?id, "limit order filled, setting state to
    closed");
                Position {
                    state: position::State::Closed,
                    ..position
                }
            } else {
                tracing::debug!(?id, "limit order partially filled, keeping
    open");
                position
            }
        }
        None if position.close_on_fill => {
            tracing::debug!(?id, "detected a newly opened limit order");
            position
        }
        _ => position,
    }
}
```

The issue is that this happens every time `put_position` is called, which also happens when trying to withdraw a position:

```
async fn execute<S: StateWrite>(&self, mut state: S) -> Result<()> {
     // ... snip ...
    metadata.state = position::State::Withdrawn;
    state.put_position(metadata).await?;
    Ok(())
}
```

This means that if one tries to withdraw a limit order, the state will be set to `Closed` instead of `Withdrawn`.

Impact

Even though the state of the position is incorrect, withdrawing a position still requires a `closed` NFT to be spent. Normally this would not be possible, but due to another bug, it was possible to end up with multiple NFTs and withdraw multiple times (see Finding 3.5. ↗).

Recommendations

The `handle_limit_order` function should only be called when a position is in the `Opened` state.

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in commit `1c9452d2` ↗.

3.12. Gas fees can be paid in any asset

Target	app/src/action_handler/transaction/stateful.rs		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

This issue was not part of the audit scope but was found during the audit while getting a better understanding of the codebase.

When a transaction is submitted, it must include a fee that is greater than the current base fee set by the chain.

```
pub(super) async fn fee_greater_than_base_fee<S: StateRead>(
    state: S,
    transaction: &Transaction,
) -> Result<()> {
    let current_gas_prices = state
        .get_gas_prices()
        .await
        .expect("gas prices must be present in state");

    let transaction_base_price =
    current_gas_prices.fee(&transaction.gas_cost());

    if transaction
        .transaction_body()
        .transaction_parameters
        .fee
        .amount()
        >= transaction_base_price
    {
        Ok(())
    } else {
        Err(anyhow::anyhow!(
            "consensus rule violated: paid transaction fee must be greater
    than or equal to transaction's base fee"
        ))
    }
}
```

The issue is that only the fee amount is checked and not the asset ID, allowing a user to pay the fee in any asset they control.

Impact

A malicious user could transfer in a large amount of a worthless asset via IBC and use it to pay for the transaction fees. This would allow them to spam the chain with transactions without having to pay any real cost. Although no real fee would need to be paid, a spend proof would still need to be generated by the malicious user.

Recommendations

The asset ID of the fee should be checked to ensure that it is the staking token.

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in commit `70f66af2` ↗.

3.13. Incorrect denom prefix replacement

Target	shielded-pool/src/component/transfer.rs		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Medium

Description

When an incoming IBC transfer is received, the denom is checked to see if Penumbra was the original source chain, and if so, the prefix is removed to get the original denom:

```
// 2. check if we are the source chain for the denom.
if is_source(&msg.packet.port_on_a, &msg.packet.chan_on_a, &denom, false) {
    // mint tokens to receiver in the amount of packet_data.amount in the denom
    of denom (with
    // the source removed, since we're the source)
    let prefix = format!(
        "{source_port}/{source_chan}/",
        source_port = msg.packet.port_on_a,
        source_chan = msg.packet.chan_on_a
    );

    let unprefixed_denom: asset::DenomMetadata = packet_data
        .denom
        .replace(&prefix, "")
        .as_str()
        .try_into()
        .context("couldnt decode denom in ICS20 transfer")?;

    let value: Value = Value {
        amount: receiver_amount,
        asset_id: unprefixed_denom.id(),
    };
```

The issue is that the `replace` function will remove all occurrences of the prefix, not only the first one.

Impact

In the normal flow, a transfer between multiple chains would look like the following:

1. Chain A transfers `CoinA` to Penumbra; it ends up with the denom `transfer/channel-0/CoinA`.
2. Penumbra transfers `transfer/channel-0/CoinA` to Chain B; it ends up with the denom `transfer/channel-1/transfer/channel-0/CoinA`.
3. Chain B transfers `transfer/channel-1/transfer/channel-0/CoinA` back to Penumbra, the prefix `transfer/channel-1` is stripped, and the denom is now `transfer/channel-0/CoinA`.

If Chain A then transfers `Cointransfer/channel-1/A` to Penumbra and then to Chain B, the denom would be `transfer/channel-1/transfer/channel-0/Cointransfer/channel-1/A`. When this is transferred back, both instances are replaced and the denom becomes `transfer/channel-0/CoinA`.

Chains such as Osmosis allow users to create custom denoms in the form of `factory/{creator address}/{subdenom}`, so a malicious user could create two denoms that could be reduced to the same denom on /Penumbra and withdrawn.

Recommendations

Only the first occurrence of the prefix should be removed.

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in commit 2b03045a ↗.

3.14. Malicious validator can trigger epoch

Target	stake/src/component/validator_handler/validator_manager.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

When a validator transitions out of the `Active` state, a flag is set to indicate that the current epoch should be ended after the current block has been processed:

```
async fn set_validator_state_inner(
    &mut self,
    identity_key: &IdentityKey,
    old_state: validator::State,
    new_state: validator::State,
) -> Result<()> {
    use validator::State::*;
    let validator_state_path =
    state_key::validators::state::by_id(identity_key);

    // Validator state transitions are usually triggered by an epoch
    transition. The exception
    // to this rule is when a validator exits the active set. In this case, we
    want to end the
    // current epoch early in order to hold that validator transitions happen
    at epoch boundaries.
    if let (Active, Defined | Disabled | Jailed | Tombstoned) = (old_state,
    new_state) {
        self.set_end_epoch_flag();
    }
```

The issue is that it is possible for a currently bonded validator to disable and enable themselves in the same transaction, which will end up triggering the new epoch and keep the validator in the active set.

The first action will change the validator from `Active` to `Disabled`, and the second action will change the validator from `Disabled` to `Inactive`. The `end_epoch` handler will then be run at the end of the block, and `set_active_and_inactive_validators` will set the inactive validator back to `Active`:

```
for (v, _) in active {
    self.set_validator_state(v, validator::State::Active)
        .await?;
}
for (v, _) in inactive {
    self.set_validator_state(v, validator::State::Inactive)
        .await?;
}

Ok(())
```

Impact

A malicious bonded validator could cause a new epoch to happen every block. During our testing, we noticed around a 20% slowdown in block production when this was happening. The epoch number is also stored as a `u16` in the `tct::Position`, so it could be possible for this to overflow.

Recommendations

A validator should not be able to disable and enable themselves in the same transaction, or the current bonding state should be checked to ensure that it is not currently unbonding.

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in commit `349c0baf` ↗.

3.15. Unchecked addition in ICS-20 transfer

Target	shielded-pool/src/component/transfer.rs		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When a denom is transferred to an external chain via IBC, the ICS-20 balance for it is updated to keep track of the total amount that has been transferred out:

```
async fn withdrawal_execute(&mut self, withdrawal: &Ics20Withdrawal) ->
    Result<()> {
    // create packet, assume it's already checked since the component caller
    contract calls `check` before `execute`
    let checked_packet =
    IBCPacket::<Unchecked>::from(withdrawal.clone()).assume_checked();

    let prefix = format!("transfer/{}/", &withdrawal.source_channel);
    if !withdrawal.denom.starts_with(&prefix) {
        // we are the source. add the value balance to the escrow channel.
        let existing_value_balance: Amount = self
            .get(&state_key::ics20_value_balance(
                &withdrawal.source_channel,
                &withdrawal.denom.id(),
            ))
            .await
            .expect("able to retrieve value balance in ics20 withdrawal!
    (execute)")
            .unwrap_or_else(Amount::zero);

        let new_value_balance = existing_value_balance + withdrawal.amount;
        self.put(
            state_key::ics20_value_balance(&withdrawal.source_channel,
    &withdrawal.denom.id()),
            new_value_balance,
        );
    }
```

This balance is then checked when a denom is transferred back to ensure that it is not possible to transfer in more than was transferred out.

The issue is that the withdrawal amount is added to the existing balance without using `checked_add`, so it is possible for the new balance to overflow the `u128` back to zero.

Impact

If the overflow occurs, it would prevent the majority of the denom from being transferred back to Penumbra as the ICS-20 balance would be too small to allow it.

Generally, it should not be possible to reach this condition as the total supply for a denom should always be under `u128`; however, it could be combined with another bug, allowing it to be exploited.

Recommendations

The new balance should be calculated using `checked_add` to ensure that it does not overflow and to bring it in line with the rest of the balance calculations in the codebase.

Remediation

This issue has been acknowledged by Penumbra Labs, and a fix was implemented in commit `53d1280e` ↗.

3.16. Timing side channel in Groth16 proof generation

Target	Groth16::prove		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

According to the benchmarks (`cargo bench -p penumbra-bench --features=parallel -- 'spend proving'`), SpendProofs are approximately 6ms faster (341ms versus 334ms) when proofs are generated for them with nonzero versus zero amount.

Unmodified SpendProof benchmark:

```
     Running benches/spend.rs (target/release/deps/spend-44899f51b8c07546)
Gnuplot not found, using plotters backend
Benchmarking spend proving: Warming up for 3.0000 s
Warning: Unable to complete 100 samples in 5.0s. You may wish to increase
    target time to 34.9s, or reduce sample count to 10.
spend proving time: [340.02 ms 341.16 ms 342.66 ms]
Found 5 outliers among 100 measurements (5.00%)
  3 (3.00%) high mild
  2 (2.00%) high severe

Number of constraints: 34630
```

When modifying the SpendProof benchmark to use zero amount,

```
diff --git a/crates/bench/benches/spend.rs b/crates/bench/benches/spend.rs
index 474f1101e..551537689 100644
--- a/crates/bench/benches/spend.rs
+++ b/crates/bench/benches/spend.rs
@@ -23,7 +23,7 @@ fn prove(r: Fq, s: Fq, public: SpendProofPublic, private:
    SpendProofPrivate) {
 }

 fn spend_proving_time(c: &mut Criterion) {
- let value_to_send = Value::from_str("1upenumbra").expect("valid value");
+ let value_to_send = Value::from_str("0upenumbra").expect("valid value");

     let seed_phrase = SeedPhrase::generate(OsRng);
     let sk_sender = SpendKey::from_seed_phrase_bip44(seed_phrase,
    &Bip44Path::new(0));
```

```
     Running benches/spend.rs (target/release/deps/spend-44899f51b8c07546)
Gnuplot not found, using plotters backend
Benchmarking spend proving: Warming up for 3.0000 s
Warning: Unable to complete 100 samples in 5.0s. You may wish to increase
    target time to 33.9s, or reduce sample count to 10.
spend proving time: [334.04 ms 334.80 ms 335.67 ms]
                        change: [-2.3289% -1.8650% -1.4490%] (p = 0.00 < 0.05)
                        Performance has improved.
Found 5 outliers among 100 measurements (5.00%)
  3 (3.00%) high mild
  2 (2.00%) high severe

Number of constraints: 34630
```

Impact

While the variance may be particularly high for SpendProofs on account of the underconstraints for dummy spends, Arkworks' implementation of Groth16 proof generation may be nonconstant time in general. This may leak witness values if there are situations where a software agent will predictably initiate proving in response to network traffic (such as liquidity providers updating prices or wallet software automatically redeeming swap claims).

Recommendations

Evaluate whether it is possible for upstream Arkworks to guarantee that proof-generation time does not depend on witness values.

Remediation

This issue has been acknowledged by Penumbra Labs, and is considered outside of the threat model.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. TOCTOU bugs in `ActionHandler`

The action-handler system in Penumbra works by running the checks of actions in parallel such that side effects of the execution of an action do not affect the checks of another action in the same transaction.

A more concise example of this would be the below call trace:

```
Action 1 -> check_stateless
Action 2 -> check_stateless
...
Action N -> check_stateless


Action 1 -> check_stateful
Action 2 -> check_stateful
...
Action N -> check_stateful


Action 1 -> execute
Action 2 -> execute
...
Action N -> execute
```

This causes issues where developers could expect these checks to act like the transaction handling in Cosmos, which is instead dispatched like this:

```
Action 1 -> check_stateless
Action 1 -> check_stateful
Action 1 -> execute


Action 2 -> check_stateless
Action 2 -> check_stateful
Action 2 -> execute


Action N -> check_stateless
Action N -> check_stateful
Action N -> execute
```

This mismatch between the intuitive mental model of sequential execution of each action's methods with the actual interleaved order leads to time-of-check time-of-use (TOCTOU) bugs, where action 1's `execute` modifies state that was already checked by action 2's `check_stateful`, and then action 2's `execute` proceeds as if its `check_stateful`'s checks still held. Mitigating these bugs efficiently can often be done by ensuring that there is a corresponding check in `Transaction::check_stateless` that ensures that multiple actions within the same transaction do not modify the same state.

As a concrete example, `Spend::check_stateful` checks that the nullifier for the note is not present in the nullifier set, and `Spend::execute` adds the nullifier to the set. This would be fine under a sequential order, but under the interleaved order, this would result in a note being able to be spent multiple times within one transaction — if not for `Transaction::check_stateless`'s `no_duplicate_spends` check, which ensures that each nullifier occurs at most once within a transaction, which ensures that `Spend::execute` (and `SwapClaim::execute`) check and modify disjoint parts of the state.

We recommend that:

- If it does not sacrifice too much performance (or possibly only under a `debug_assertions`-style feature flag), `Transaction::execute` should rerun each `Action`'s `check_stateful` method immediately before the corresponding `execute`, eliminating this class of bugs.
- If the sequential execution of `check_stateful` is only done under a feature flag, differential fuzzing or property testing of transaction validation (with a mutator or strategy that provides valid proofs and signatures) with and without the flag may be able to detect instances of this bug class.
- Any implementation of `ActionHandler` with a nontrival `check_stateful` method should have a documentation comment on `execute` explaining why possible interleavings of other `Action`'s `execute` methods still meet the preconditions established as if `check_stateful` had been run sequentially, referencing the corresponding check in `Transaction::check_stateless` if applicable.

Remediation

This issue has been acknowledged by Penumbra Labs, and fixes were implemented in the following commits:

- 1f084185 ↗
- 2949f8f0 ↗

In PR 3962 ↗, `check_stateful` and `execute` were renamed to `check_historical` and `check_and_execute`, with `check_historical`'s documentation explaining this risk.

Many former `check_stateful` checks were moved to `check_and_execute` pending profiling, and `SAFETY:` comments describing invariants were added to the remaining `check_historical` checks.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the crates and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. The value-balance mechanism

Penumbra uses additively homomorphic commitments in the style of Zcash Sapling to ensure that the balances contributed by actions within a transaction sum to zero, with the capability to hide intermediate amounts. The `IsAction` trait, which all actions implement, contains a function `fn balance_commitment(&self) -> balance::Commitment;`, which determines the value (potentially of multiple assets) that each action credits or debits from the transaction balance and whether that value is transparent or hidden. Balance commitments are group elements in the Decaf377 elliptic curve, whose order is slightly less than 2^{256}, and are of the form $C_j = b_j H + \Sigma_i v_{ij} G_i$, where

- b_j is a random blinding value (which is zero for transparent commitments)
- H is a generator that is the Elligator encoding of the Blake2b hash of the string `decaf377-rdsa-binding`
- v_{ij} is the (possibly negative) value associated with asset `i` in commitment `j`
- G_i is a generator depending on asset `i`'s ID (the Elligator encoding of the Poseidon hash of the Blake2b hash of the string `penumbra.value.generator` and the asset's numeric ID)

These commitments have several important properties:

- Summing them as curve points produces a commitment to the sum of the balances.
- If b_j is random, knowledge of C_j does not reveal anything about v_{ij}.
- If all of v_{ij} are zero, C_j can be used as an ECDSA public key to check signatures signed with knowledge of b_j
- If some of v_{ij} are nonzero, a malicious signer can only produce a signature for C_j by finding some k such that $kH = \Sigma_i(-v_{ij})G_i$ and signing with $b'_j = b_j + k$, but finding such a k requires solving the discrete logarithm problem.
- Statements involving their components can be efficiently proved in zero knowledge by providing b_j and v_{ij} as witnesses to the circuits.

The first property is used by `Transaction::binding_verification_key` to produce a commitment to the sum of the transaction's actions' balances. The third property is used by `Transaction::check_stateless`'s `valid_binding_signature` check to ensure that the transaction's balance sums to zero by checking the signature produced in `TransactionPlanner::apply_auth_data`, where the `synthetic_blinding_factor` used to construct the `binding_signing_key` corresponds to $\Sigma_j b_j$.

In order for this to correctly represent value, each action's implementation must ensure that no unconstrained attacker-controlled values are included in the determination of the balance commitment and that overflow modulo the order of the group is not possible. Each action must ensure that the amounts contributed to the value balance are accurately tracked, whether that be through external state (like the nullifier set/state commitment tree) or by having a mix of negative and positive of different assets within the action's balance commitment.

For a more detailed treatment of the single-asset case, see section 4.13, "Balance and Binding Signature (Sapling)", of the Zcash protocol specification (https://zips.z.cash/protocol/protocol.pdf ↗).

5.2. Crate: shielded-pool

Handler: `Ics20Transfer`

The `Ics20Transfer` handler is responsible for receiving incoming assets from an external chain over IBC. The controllable fields for the packet are:

- `denom` — the token denomination of the asset being transferred
- `amount` — the amount of the asset being transferred
- `sender` — the address of the sender on the source chain
- `receiver` — the address of the receiver on Penumbra
- `memo` — an optional memo

`Ics20Transfer::recv_packet_check` always returns `Ok(())`.

`Ics20Transfer::recv_packet_execute` performs two different actions depending on whether the received `denom` was originally sent by Penumbra or if it is an external asset. This check is done by seeing if the `denom` starts with `{source_port}/{source_channel}/` where the source port and channel are the current IBC port and channel for the connection. If Penumbra was the original source,

- The prefix of `denom` is removed.
- The ICS-20 balance is checked to ensure at least `amount` tokens were sent to the external chain.
- The tokens are minted to the receiver.
- The ICS-20 balance is then reduced by `amount`.

If the asset is from an external chain,

- The `denom` is prefixed with `{source_port}/{source_channel}/` based on the current connection.
- The prefixed `denom` is registered if it does not exist.
- The tokens are minted to the receiver.
- The ICS-20 balance for the prefixed `denom` is increased by `amount`.

Action: `Ics20Withdrawal`

The `Ics20Withdrawal` action allows a user to transfer funds from Penumbra to another chain using over an existing IBC connection. The action contains the following fields.

- `amount` — a transparent value consisting of an amount (`u128`)
- `denom` — the asset metadata of the asset being transferred, only the `base_denom` is used
- `destination_chain_address` — the address on the destination chain to send the transfer to
- `return_address` — an ephemeral address where funds are returned; this will be the `sender` of the fungible token packet data
- `timeout_height` — a chain height at which the transfer expires
- `timeout_time` — a timestamp at which the transfer expires
- `source_channel` — the IBC channel used for the withdrawal

`Ics20Withdrawal::check_stateless` verifies that the `timeout_time` is not zero.

`Ics20Withdrawal::check_stateful` verifies that

- the source channel exists and is not closed.
- there is a connection and nonfrozen client for the channel.
- the latest height of the receiving chain is less than the `timeout_height`.

`Ics20Withdrawal::execute` does two different things depending on whether the funds being transferred are local to Penumbra or if they were originally transferred in from an external chain. In the case where Penumbra is the source, the total ICS-20 balance for the specified asset is increased by the transfer amount and the total supply is unchanged (see Finding 3.3. ↗). In the external case, the total ICS-20 balance for the specified asset is decreased by the transfer amount and the total token supply for the asset is decreased by the same. Then the packet is sent to the destination chain.

Action: `Output`

The `Output` action allows a user to create a new note, subtracting from the transaction's value balance. The action contains the following fields.

- `body` — the body of the output, containing the note payload, balance commitment, and wrapped keys
- `proof` — a zero-knowledge proof to verify that the balance and note commitments are valid

`Output::check_stateless` verifies that the supplied `proof` matches the supplied public inputs.

`Output::check_stateful` is a no-op and always succeeds.

`Output::execute` will add the supplied note payload into the state commitment tree and emit an event.

Action: `Spend`

The `Spend` action allows a user to spend a note, adding to the transactions-value balance. The action contains the following fields.

- `body` — the body of the spend, containing the balance commitment, nullifier, and randomized verification key
- `auth_sig` — the signature to be verified by the randomized verification key
- `proof` — a zero-knowledge proof to verify that the supplied balance commitment, nullifier, verification key, and transactions anchor are valid

`Spend::check_stateless` verifies the spend auth signature and that the supplied `proof` matches the public inputs.

`Spend::check_stateful` checks that the nullifier has not been used before (also see 4.1. ↗).

`Spend::execute` will add the supplied note payload into the state commitment tree and emit an event.

5.3. Crate: stake

Action: `Delegate`

The controllable parameters of the `Delegate` action are

- the `validator_identity` — this is how the validator to be delegated to is identified
- the `epoch_index` is when the delegation is/will be performed — the delegation starts on the next epoch
- the `unbonded_amount` — the amount to be delegated
- the `delegation_amount` — the amount after the delegation exchange rate, used for verification of the delegation

For this action handler, there is no `check_stateless` check.

The `check_stateful` check ensures that the validator exists, that the epoch is correct by comparing it to the validator's next rate data, and that the parameters match. It then ensures that the validator is enabled and that it is not jailed or tombstoned. It then verifies that the returned `delegation_amount` matches the expected `delegation_amount`.

The stake is then delegated to the validator, and if enough stake has been reached, the validator state is transitioned from `Defined` to `Inactive`.

Action: `Undelegate`

The controllable parameters of the `Undelegate` action are

- the `validator_identity` — this is how the validator to undelegate from is identified
- the `start_epoch_index` — is the epoch the delegation started, it is used for verification
- the `unbonded_amount` — the expected unbonded amount
- the `delegation_amount` — the amount to undelegate

For this action handler, there is no `check_stateless` check.

The `check_stateful` check ensures that the validator exists, that the epoch is correct by comparing it to the validator's next rate data, and that the parameters match. It then verifies that the returned `delegation_amount` matches the expected `unbonded_amount`.

The `execute` registers the undelegation for execution at the end of the block and registers an undelegation denominator.

Action: `UndelegateClaim`

The controllable parameters of the `UndelegateClaim` action are

- the `body->validator_identity` — this is how the validator to undelegate from is identified
- the `body->start_epoch_index` — is the epoch the delegation started; this is used for verification
- the `body->penalty` — the expected penalty used for verification
- the `body->balance_commitment` — the commitment for the proof
- the `proof` — the ZK proof for the undelegation claim

The `check_stateless` check verifies the zero-knowledge proof, which ensures that the balance commitment debits an amount of bonded stake for the specified validator and credits the corresponding amount of unbonded stake at an exchange rate based on the penalty.

The `check_stateful` check verifies that the `unbonding_epoch` cannot be greater than the current epoch and that the penalty supplied matches the expected penalty.

The `execute` is not implemented as the value change of this commitment is doing the undelegation claim.

Action: `validator::Definition`

The `definition` action allows a user to define/upload a validator to the blockchain, such that they can be delegated too and eventually become an active validator.

The controllable parameters of the `Delegate` action are

- the `validator` parameter, which contains all the information about the validator, such as the name, website, and so on — it also contains information about the various keys (identity, governance, consensus), the destination funding stream, and a sequence number for updates

- the `auth_sig` — used to verify the validator against the identity key

`check_stateless` ensures limits of the descriptionary parts of the validator and ensures the auth sig validates against the identity key and that the funding streams remain below 10,000 BPS.

`check_stateful` ensures that the sequence number is increasing and that another validator with the same consensus key does not already exist; however, these checks are erroneous (see Finding

3.8. ↗).

`execute` adds the validator and registers it to tendermint.

5.4. Crate: dex

Action: `Swap`

A `Swap` action specifies two assets to convert between at the current market rate and contains a ciphertext that allows subsequent proof (in `SwapClaim`) that a corresponding amount of each asset was burnt.

A `Swap` action contains a trading pair, two transparent amounts Δ_1 and Δ_2, a fee commitment, a swap ciphertext, a commitment to the hash of the fields of the corresponding swap plaintext, and a zero-knowledge proof relating these values. The swap ciphertext is authenticated and encrypted with ChaCha20Poly1305, and the corresponding swap plaintext contains another trading pair, amounts, a trans parent fee, an address (to redeem the swap at), and a nonce (rseed). It is expected in normal use that one of Δ_1 or Δ_2 is zero (since the amount of whichever is smaller could be deducted from both amounts to produce a `Swap` with the same outcome), but this is not enforced.

Since Δ_1 and Δ_2 are transparent, swaps do not currently hide the values being swapped; this is intended to be solved by flow encryption in future work. Since there are no bounds on what range of exchange rates to accept, slippage may occur if liquidity positions are modified in the same block that the swap is included in.

`Swap::check_stateless` verifies the zero-knowledge proof, which enforces that

- the plaintext swap provided as a witness has the same hash as the swap commitment.
- the transparent fee inside the plaintext swap matches the fee commitment.
- the amounts in the plaintext swap are the same as the amounts in the action.
- the balance commitment deducts the fee and values corresponding to the amounts of the specified assets in the trading pair.

`Swap::check_stateful` unconditionally succeeds.

`Swap::execute` records state relating to the swap:

- Under "`/dex/swap_flows`", it adds the amounts to be swapped to the trading pair to be processed at the end of the block.
- It adds swap commitment to the state commitment tree.
- It adds the payload containing the swap ciphertext and swap commitment to "`dex/pending_payloads`", to be stored in the compact block.

Action: `SwapClaim`

A `SwapClaim` action redeems the assets that were swapped by a `Swap` action in an earlier block as notes (which can be spent with a `Spend` action in a subsequent block).

A `SwapClaim` action contains a nullifier, a transparent fee, two commitments to note ciphertexts, a `BatchSwapOutputData` struct, a zero-knowledge proof, and an epoch duration. A `BatchSwapOutputData` struct contains a trading pair that mentions which assets were swapped, input amounts (Δ_1, Δ_2), aggregated amounts swapped from the corresponding `Swap`'s block (Λ_1, Λ_2), and aggregated unfilled amounts from the corresponding `Swap`'s block $(\texttt{unfilled}_1, \texttt{unfilled}_2)$, a block height, and an epoch height.

`SwapClaim::check_stateless` verifies the zero-knowledge proof, which enforces that

- the swap plaintext provided as a witness matches the swap commitment provided as another witness.
- the swap commitment is present in the state commitment tree.
- the nullifier was computed correctly.
- the transparent fee matches the one in the swap plaintext.
- the output data's block height is equal to the sum of the output data's epoch height and the epoch-relative block height (from the SCT proof).
- the output data's trading pair matches the one in the swap plaintext.
- the amounts attempting to be claimed are equal to the fraction of the aggregated swaps corresponding to this swap plaintext's amounts.
- the note commitments are correctly computed for notes with the current claimed values, with blinding factors provided as witnesses.

`SwapClaim::check_stateful` checks that

- the epoch duration provided in the `SwapClaim` matches the state's epoch duration parameter (although the `SwapClaim`'s epoch duration appears otherwise unused).
- the provided `BatchSwapOutputData` matches the state's records for the corresponding block height.
- the nullifier is unspent.

`SwapClaim::execute` records

- the note commitments to the SCT
- the note positions and commitments to the compact block
- the nullifier to the spent nullifier set

To avoid the potential TOCTOU issue (4.1. ↗) regarding duplicate nullifier spends in the same transaction, `Transaction::check_stateless`'s `no_duplicate_spends` checks that all nullifiers are unique within a transaction.

Action: `PositionOpen`

A `PositionOpen` action opens a trading position that provides liquidity for swaps and contains just a position.

A position contains a state, which is one of `Opened`, `Closed`, `Withdrawn`, or `Claimed`; two reserve amounts; a `TradingFunction`, a nonce, and a flag indicating whether it is a limit order (that should be automatically closed once one of its reserves reach zero).

A `TradingFunction` specifies which pair of assets liquidity is being provided for, what fee should be paid for trades that go through this position, and two trading coefficients that specify what ratio this position permits swaps at.

The balance commitment associated with a `PositionOpen` is transparent, debits the position's reserves from the balance, and credits the LPNFT for the new position to the transaction's balance.

An LPNFT is an asset whose generator is derived from a hash of the position ID and current state. A position ID is a hash of the position's nonce, asset types, fee, and trading coefficients (importantly, not of its reserves or state, which are mutable).

The transaction planner will by default store the LPNFT for the position in an `Output` note as change, which can be used in a subsequent transaction with a `Spend` + `PositionClose`, but other arrangements are possible (e.g., using a `PositionOpen` and `PositionClose` within the same transaction to only provide liquidity at a certain exchange rate for one block and then storing the closed position's LPNFT in an `Output` note).

`PositionOpen::check_stateless` invokes `Position::check_stateless`, which checks that

- both reserves are less than or equal to `MAX_RESERVE_AMOUNT` $= 2^{80} - 1$.
- at least one of the reserves is nonzero.
- both trading coefficients are nonzero (either being zero would imply an infinite price for the opposite asset).
- both trading coefficients are less than or equal to `MAX_RESERVE_AMOUNT` $= 2^{80} - 1$.
- the trading function's assets are distinct (to avoid creating self-edges in the position graph).
- the fee is less than 50%.

`PositionOpen::check_stateful` checks that the position ID did not occur in any previous transaction by checking the state under "`dex/position/{id}`".

`PositionOpen::execute` calls `PositionManager::put_position`, which stores the position in "`dex/position/{id}`" and updates various indexes relating to the position.

There is a TOCTOU bug (4.1. ↗) that allows a position to be opened multiple times with the same ID within one transaction (see Finding 3.5. ↗), which should be prevented by adding a check to `Transaction::check_stateless` that ensures that all transaction IDs opened are unique.

Action: `PositionClose`

A `PositionClose` action closes a currently open position, specified by its ID.

The balance commitment associated with a `PositionClose` is transparent, debits the LPNFT for the open position with the specified ID, and credits the LPNFT for the closed position with the specified ID to the transaction's balance.

`PositionClose::check_stateless` and `PositionClose::check_stateful` both unconditionally succeed.

`PositionClose::execute` queues the position ID in the state under “`dex/pending_position_closures`” to be closed at the end of the block.

Action: `PositionWithdraw`

A `PositionWithdraw` action withdraws reserves from a closed position, specified by its ID, and specifies a transparent commitment to the reserves.

The balance commitment associated with a `PositionWithdraw` is transparent, debits the LPNFT for the closed position with the specified ID, and credits both the LPNFT for the withdrawn position with the specified ID as well as the reserves to the transaction’s balance.

`PositionWithdraw::check_stateless` unconditionally succeeds.

`PositionWithdraw::check_stateful` checks that the reserves currently associated with the position in the state match the specified reserves to withdraw.

`PositionWithdraw::execute` retrieves the position from the state, checks that it is closed (returning an error if it is not), and stores that it is withdrawn in the state.

While there is a potential TOCTOU issue (4.1. ↗) if the reserves are modified (such as being traded against while open, then closed, then withdrawn, potentially resulting in the assets being swapped twice — once in the batch swap and again in the withdraw-with-stale-reserves), this seems to be unreachable in practice since `PositionWithdraw` checks that the position is closed immediately, while `PositionClose` defers the close to the end of the block.

The comments in `PositionWithdraw::check_stateful` imply that submitting `PositionClose` + `PositionWithdraw` for the same position in a single transaction is intended to be supported. This seems like it would require deferring withdrawals to the end of block to happen after the queue of position closures is processed and require checking that the reserves have not shifted there as well. But that would also require changing how the reserves are withdrawn, since while the state transition could be cancelled (and the position left closed instead of withdrawn) at the end of the block, there would be no straightforward way to cancel the withdrawn value at that point (since the value is not necessarily stored in a pair of `Output` notes, there is not necessarily even a pair of nullifiers to burn).

Per discussion with Penumbra Labs, it’s not intended that a position should be able to be closed + withdrawn in the same transaction, and the comment in `PositionWithdraw::check_stateful` will be revised.

Action: `PositionRewardClaim`

The `PositionRewardClaim` actions were intended, in the future, to allow providing retroactive liquidity incentives. They are currently an unimplemented placeholder, and `PositionRewardClaim::{check_stateless,check_stateful,execute}` all unconditionally return an error.

`PositionRewardClaim` and `State::Claimed` were removed in `f10d6a44` ↗ in favour of a different mechanism involving adding sequence numbers to `State::Withdrawn`.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the mainnet.

During our assessment on the scoped Penumbra crates, we discovered 16 findings. Five critical issues were found. Five were of high impact, three were of medium impact, two were of low impact, and the remaining finding was informational in nature. Penumbra Labs acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.