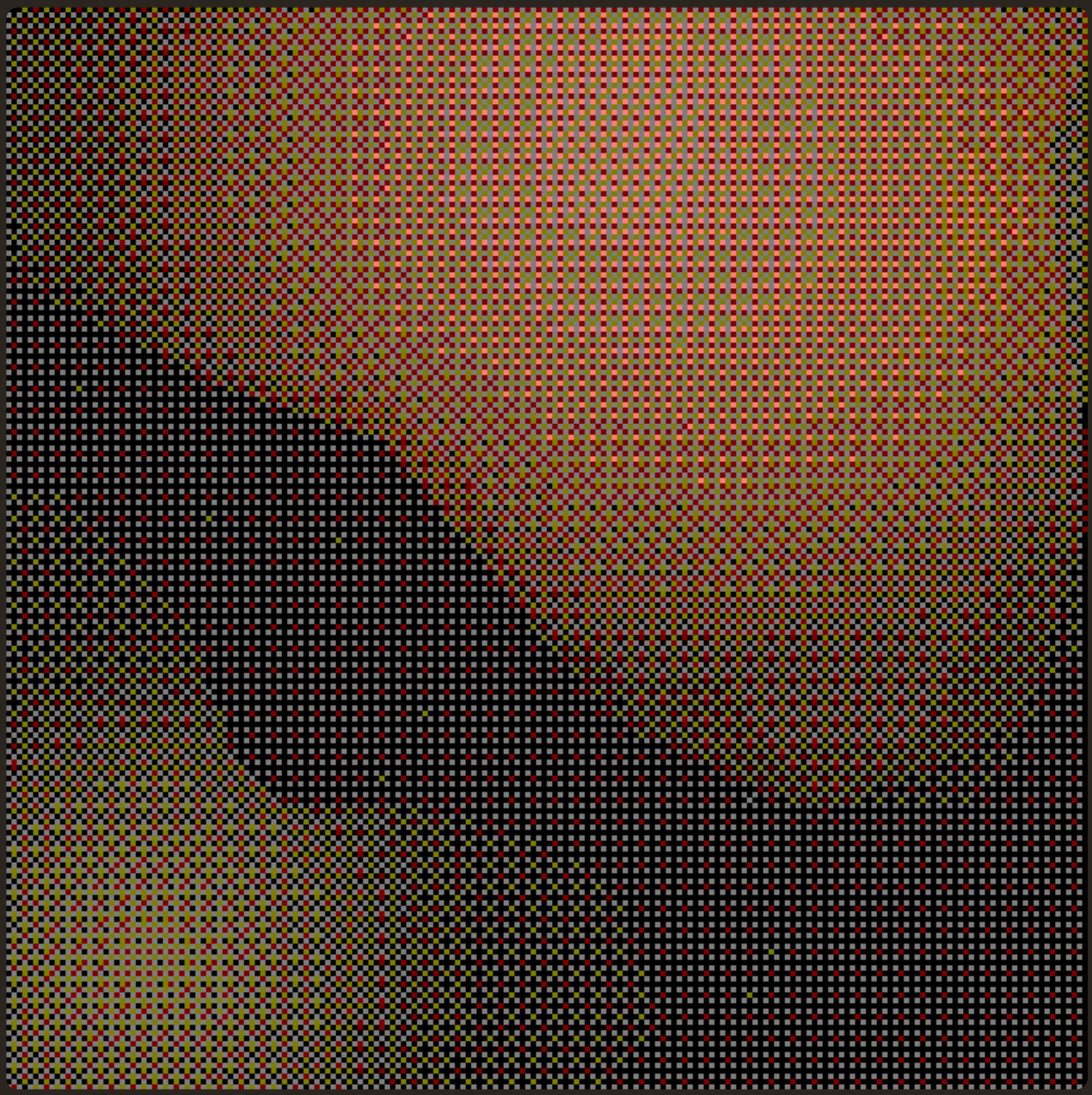
ZKSECURITY

Audit of Lighter's Exit Hatch

NOVEMBER 5TH, 2025 • TECHNICAL REPORT



Introduction

This report presents the findings of a security audit conducted on Lighter's desert exit ZK circuits. The audit was performed by zkSecurity starting November 3rd, 2025, with two consultants over a period of one week. The audit focuses on the emergency withdrawal (escape hatch) mechanism that allows users to withdraw funds when the Sequencer is unavailable, and some minor changes done to other components. This is the 5th audit in a series of audits on the Lighter protocol, so we will give a brief overview focused on the newly reviewed functionality.

Scope

The audit was conducted on commit `b99c425b8c0868501e65e78fd16aaf16bbe0ea2b` of a private `lighter-prover` repository. The primary focus was the desert exit circuit located in the `desertexit/circuits/` folder. This circuit implements the escape hatch mechanism and consists of:

- `inner_circuit.rs`: The core business logic that validates exit claims against the state root, computes total account value (TAV), and verifies withdrawal amounts.
- `outer_circuit.rs`: The proof wrapping layer that enables recursive proof verification.
- `pubdata_account.rs` / `pubdata_market.rs`: Data structures representing account and market state extracted from blob data.
- `deserializers.rs`: Logic for deserializing blob data into circuit-compatible formats.

In addition to the desert exit circuit, several minor changes to the main transaction circuits were in scope. These include: corrections to zero quote calculation in liquidation and deleverage transactions; exclusion of frozen markets from margin requirements (while still counting unrealized PnL and funding in TAV); modifications to the delta circuit's sign packing during polynomial evaluation; removal of the health check requirement for deleveragers; transfer restrictions for frozen public pools; and allowing users to set a zero public key.

Two cross-cutting changes were also reviewed: the `try_sub_biguint` refactor, which now explicitly returns and checks the borrow flag to avoid underflows in disabled constraint branches; and the migration of `quote_multiplier` from the private market hash to the public hash and blob data, making it available for desert exit TAV calculations.

Overview of Key Concepts

The *desert exit* mechanism (or *Escape Hatch* in the Lighter whitepaper) is a functionality of the Lighter L1 contract. It allows users to submit an exit claim and withdraw funds from the contract, even when the Sequencer fails to include transactions.

Circuit. Naturally, any exit claim submitted by a user must be checked for validity against the latest available state root. This verification is performed using a SNARK. The SNARK circuit takes as input a trusted state root and validium root, an untrusted exit claim, and untrusted blob data. It then performs the following verification logic:

1. assert that the validium and state roots can be reconstructed from the untrusted blob data.
2. compute the user's total account value (TAV) from the (now trusted) blob data.
3. assert that the exit claim is made for valid accounts and matches the computed TAV.

Proof wrappers. The SNARK uses the common pattern of proof wrapping (or proof recursion). The circuit is first proven client-side using a proof system that has good prover performance but larger proofs and verifier times, in this case Plonky2. This produces a first proof that then gets wrapped using a different proof system which has small proofs and cheap on-chain verification. Specifically, the Gnark implementation of Plonk over the BN254 elliptic curve.

We note that in the audited code, the SNARK uses 3 proof system layers (and therefore wraps twice):

- the first layer is a Plonky2 circuit that does arithmetic over the Goldilocks field. It enforces the business logic listed above. The Merkle trees and verifier challenges for this proof are computed using the Poseidon hash function for the Goldilocks field.
- the second layer is a Plonky2 circuit that does arithmetic over the Goldilocks field. It verifies the first layer proof against its public inputs. The Merkle trees and verifier challenges for this proof are computed using the Poseidon hash function for the scalar field of the BN254 elliptic curve. This change of hash function allows the proof to be efficiently verified by the next layer.
- the third and final layer is a Gnark circuit that does arithmetic over the scalar field of the BN254 elliptic curve. It verifies the second layer proof against its public inputs. The resulting proof is much smaller than the hash-based proofs produced by the previous two layers.

TAV calculation. The total account value (TAV) calculation is a central step in the validation of an exit claim since it dictates how much funds can be withdrawn. The desert hatch circuit is hard-coded to support an exit claim on one main account with up to 16 sub-accounts. The

circuit assumes that these sub-accounts represent shares in public pools.

The TAV is calculated from the positions of the main account; the TAV of each pool, proportioned by the amount of shares owned in that pool; and the remaining collateral. The withdrawable amount depends directly on the TAV. If the TAV is 0 or negative, then no funds can be withdrawn. If the main account is a public pool, then only the operator fee (see documentation) can be withdrawn; this fee is calculated separately as a function of the pool TAV. Finally, if the TAV is positive and the main account is not a public pool, then the withdrawable amount is exactly the TAV.

We detail the constants, variables and steps that go into the calculation of the TAV and withdrawable amount below. Note that for this exposition we fix a missing variable assignment (as described in our findings).

Constants:

- $M = \texttt{USDC\_TO\_COLLATERAL\_MULTIPLIER}$, set to 10^6.
- $n_{\text{pos}} = \texttt{POSITION\_LIST\_SIZE}$, set to 255.
- $n_{\text{accounts}} = \texttt{DESERT\_NUM\_ACCOUNTS}$, set to 17 (1 main account and 16 pools).

Variables:

- v_i^{notional} base notional value of position i.
- $\delta_i^{\text{funding}}$ base funding delta for position i.
- C_{agg} aggregated collateral.
- s_i user's share amount in pool i.
- S_i total shares in pool i.
- s_{op} operator shares.
- S_{total} total shares (for operator pools).

Step-by-step calculation (as in the circuit):

1. Calculate positions TAV:
$$\texttt{positions\_tav} = M \times \sum_{i=0}^{n_{\text{pos}}-1} \left(v_i^{\text{notional}} + \delta_i^{\text{funding}}\right)$$
2. Extend collateral:
$$\texttt{extended\_collateral} = M \times C_{\text{agg}}$$
3. Combine:
$$\text{TAV}_{\text{except\_pools}} = \texttt{positions\_tav} + \texttt{extended\_collateral}$$
4. For each pool $i \in \{1, \ldots, n_{\text{accounts}}\}$,
   (a) Compute the pool TAV (in extended form):
$$\texttt{pool\_tav}_i = M \times \sum_{j=0}^{n_{\text{pos}}-1} \left(v_{i,j}^{\text{notional}} + \delta_{i,j}^{\text{funding}}\right) + M \times C_{\text{agg},i}$$
   (b) Calculate share value using total shares in extended form ($S_i \times M$):
$$\texttt{pool\_value}_i = \frac{s_i \times \texttt{pool\_tav}_i}{S_i \times M}$$
   (c) Extend pool value:
$$\texttt{extended\_pool\_value}_i = M \times \texttt{pool\_value}_i$$
5. Sum pool contributions:
$$\text{TAV}_{\text{pools}} = \sum_{i=1}^{n_{\text{accounts}}} \texttt{extended\_pool\_value}_i$$
6. Combine components:
$$\text{TAV}_{\text{raw}} = \text{TAV}_{\text{except\_pools}} + \text{TAV}_{\text{pools}}$$
7. Apply floor:
$$\text{TAV}_{\text{positive}} = \begin{cases} 0, & \text{if } \text{TAV}_{\text{raw}} < 0 \\ \text{TAV}_{\text{raw}}, & \text{if } \text{TAV}_{\text{raw}} \geq 0 \end{cases}$$
8. Adjust for pool operator:
$$\text{TAV}_{\text{adjusted}} = \begin{cases} \frac{\text{TAV}_{\text{positive}} \times s_{\text{op}}}{S_{\text{total}}} & \text{if pool operator} \\ \text{TAV}_{\text{positive}} & \text{otherwise} \end{cases}$$
9. Final TAV:
$$\text{TAV}_{\text{final}} = \frac{\text{TAV}_{\text{adjusted}}}{M}$$

The complete formula for the TAV calculation is:

$$\boxed{\text{TAV}_{\text{final}} = \frac{1}{M} \times \max\left(0, \text{TAV}_{\text{raw}}\right) \times \alpha}$$

where:

- $\text{TAV}_{\text{raw}} = M \times \left[\sum_{i=0}^{n_{\text{pos}}-1} \left(v_i^{\text{notional}} + \delta_i^{\text{funding}}\right) + C_{\text{agg}}\right] + \sum_{i=1}^{n_{\text{accounts}}} \frac{s_i \times \texttt{pool\_tav}_i}{S_i}$.
- $\alpha = 1$ for regular users and $\alpha = s_{\text{op}} / S_{\text{total}}$ for pool operators.

> *Note: following the introduction of the spot market functionality, the exit hatch circuit has been modified to support withdrawal of assets other than USDC. We discuss these changes in the corresponding report, "Audit of Lighter's Spot Market Circuits and Multi-Asset Support".*

Computing the state root. The state root is computed as a hash of hashes: the public market info root, the account tree root and the validium root. Each root is computed as a hash (or hash tree) of some underlying data structure. The specific schema for these data structures may vary.

From a security perspective, the properties of hash functions makes the state root a binding commitment to some (set of) *strings*. In order to ensure that the state roots bins the protocols to a given *state*, the schema must be designed such that every string has a unique interpretation for the protocol.

Summary and Recommendations

We found the desert exit circuit to be well-designed and to follow sound architectural principles for emergency withdrawal functionality. Our analysis focused on adversarial scenarios where a malicious user or sequencer might attempt to exploit the escape hatch mechanism, examining both soundness (preventing invalid withdrawals) and completeness (ensuring legitimate withdrawals succeed).

The findings below identify logical issues that warrant attention, particularly the TAV computation vulnerability which could allow withdrawal of funds from accounts with negative balances. The remaining findings are relatively minor and relate to arithmetic safety and input validation.

We recommend strengthening the test coverage for the desert exit functionality. The repository includes an end-to-end test (`test_inner_desert_if_witness_exists`) and a proving pipeline (`prove.sh`), for which only one example input scenario is provided. Automated end-to-end tests with diverse scenarios, including corner cases such as zero balances, negative TAV, maximum values, and boundary conditions, would provide stronger confidence on the functional correctness of the desert exit.

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	lighter-prover/desertexit/circuits/src/inner_circuit.rs	Negative TAV bypass allows withdrawal of absolute value	**High**
#01	lighter-prover/desertexit/circuits/src/pubdata_account.rs	Unauthenticated master_account_index in desert exit commitment	**Medium**
#02	lighter-prover/desertexit/circuits/src/pubdata_market.rs, inner_circuit.rs, witness/utils.go	Arithmetic overflows at type boundaries	**Low**
#03	lighter-prover/desertexit/witness/utils.go	Signed integer conversions in collateral computation obscure unsigned semantics	**Informational**

#00 - Negative TAV bypass allows withdrawal of absolute value

Severity: High **Location:** lighter-prover/desertexit/circuits/src/inner_circuit.rs

Description. The desert exit circuit is intended to clamp negative Total Account Values (TAV) to zero before allowing withdrawals. The code includes a comment and logic suggesting this intention at lines 197-203 of `inner_circuit.rs`:

```
// Negative to zero
{
    let zero_bigint = self.builder.zero_bigint();
    let is_tav_negative = self.builder.is_sign_negative(calculated_tav.sign);
    self.builder.select_bigint(is_tav_negative, &zero_bigint, &calculated_tav);
}
```

However, the result of `select_bigint` is never assigned back to `calculated_tav`. Compare this to the correct pattern used elsewhere in the same file (line 241):

```
calculated_tav = self.builder.select_bigint(
    is_pool_account,
    &pool_tav,
    &calculated_tav,
);
```

Because the clamping is not applied, the circuit proceeds with the original (potentially negative) `calculated_tav`. Later, at lines 206-208, the circuit:

1. Takes the absolute value: `calculated_tav.abs`
2. Asserts it equals the witnessed `total_account_value`
3. Includes this value in the exit commitment

```
self.builder.connect_biguint(
    &calculated_tav.abs,
    &self.target.total_account_value,
);
```

Impact. A malicious prover with a negative TAV (underwater position with collateral smaller than liabilities) can:

1. Compute a legitimate negative TAV (e.g., -1000 USDC)
2. Witness the absolute value (1000 USDC) in `total_account_value`
3. Pass circuit verification because: - The clamping to zero is not applied - The circuit only checks `|calculated_tav| == witnessed_value` - The absolute value matches
4. Extract positive funds from the contract despite having negative account value

This allows insolvent users to drain funds from the protocol during desert mode, potentially leaving the contract undercollateralized and harming other users.

Recommendation. Assign the result of `select_bigint` back to `calculated_tav`. This ensures that negative TAV values are clamped to zero before the absolute value check, preventing underwater accounts from withdrawing funds.

Client response. Fixed in commit `f6818da` on the `spot` branch. The result of `select_bigint` is now correctly assigned back to `usdc_balance`.

#01 - Unauthenticated master_account_index in desert exit commitment

Severity: Medium **Location:** lighter-prover/desertexit/circuits/src/pubdata_account.rs

Description. The desert exit circuit includes `master_account_index` in the exit commitment without cryptographically authenticating it against the on-chain state. This creates an authentication gap between what the circuit proves and what the contract receives.

In Lighter's architecture, accounts can be either master accounts or sub-accounts. Sub-accounts inherit their L1 withdrawal address from their master account during creation (`l2_create_sub_account.rs:128`). The `master_account_index` field identifies this parent relationship.

In the main protocol, the Private Account Tree hash includes `master_account_index` in its Poseidon hash:

```
// circuit/src/types/account/account_hash.rs:135
let mut elements = vec![];
elements.extend_from_slice(&partial_hash[0].elements);
elements.push(self.master_account_index);  //  Authenticated
elements.extend_from_slice(&self.l1_address.limbs...);
...
```

This cryptographically binds sub-accounts to their owners through the Merkle tree.

In the desert exit circuit however, the Account Pub Data Tree hash omits this field:

```
// desertexit/circuits/src/pubdata_account.rs:264-276
let non_empty_pub_data_hash = {
    let mut pub_data_elements = vec![];
    pub_data_elements.extend_from_slice(&partial_pubdata_hash.elements);
    pub_data_elements.extend_from_slice(&self.l1_address.limbs...);
    pub_data_elements.push(self.account_type);
    pub_data_elements.extend_from_slice(&self.aggregated_collateral.abs.limbs...);
    pub_data_elements.push(self.aggregated_collateral.sign.target);

    builder.hash_n_to_hash_no_pad::<Poseidon2Hash>(pub_data_elements)
};
```

The pubdata hash authenticates:

- `l1_address` (the account's withdrawal address)
- `aggregated_collateral` (the balance)
- `positions`, `public_pool_shares`, `account_type`

But does not include the `master_account_index`.

The witness-supplied `master_account_index` is directly copied into the exit commitment (`inner_circuit.rs:386-390`) without verification:

```
// Master account index (6 bytes, 48-bit)
let mut master_account_index_bytes = self.builder
    .split_bytes(self.target.accounts[0].master_account_index, 6);
master_account_index_bytes.reverse();
elems.extend_from_slice(&master_account_index_bytes);
```

Impact. The circuit allows a prover to include an arbitrary `master_account_index` in the exit commitment without cryptographic verification. A user proving their legitimate account (with authenticated `accountIndex`, `l1_address`, and `TAV`) can claim any `master_account_index` value.

The actual impact depends on how the on-chain contract uses this field, which is outside the scope of this circuit audit. Based on developer feedback, the contract increases `balanceToWithdraw[masterAccountIndex]` based on the exit commitment, and users withdraw funds via a separate call that maps their L1 address to their master account index.

This means users could potentially: 1. Mistakenly redirect their withdrawal balance to an incorrect master account 2. Intentionally “donate” their balance to another user’s master account 3. Cause confusion in accounting if multiple provers claim different master accounts for the same sub-account

The authenticated `l1_address` in the pubdata hash provides some protection if the contract uses it correctly, but relying on an unauthenticated field for critical logic creates fragility at the circuit/contract boundary.

Recommendation. The authentication gap should be closed either at the circuit level or contract level:

- A the circuit level, one solution is to include `master_account_index` in the pubdata account hash alongside the other authenticated fields like `l1_address`, `account_type`, and `aggregated_collateral`. This would ensure that `master_account_index` is cryptographically bound to the account state through the Merkle tree. This approach mirrors how the field is already handled in the main protocol’s Private Account Tree.
- Alternatively, the on-chain contract should implement additional validation to verify the relationship between the authenticated `accountIndex`, its `l1_address`, and the claimed `master_account_index`.

The circuit/contract boundary should be carefully reviewed to ensure that unauthenticated fields cannot be misused for fund routing or access control decisions.

Client response. Fixed in commit `f6818da` on the `spot` branch. The exit commitment now uses the authenticated `l1_address` field instead of the unauthenticated `master_account_index`.

#02 - Arithmetic overflows at type boundaries

Severity: Low **Location:** lighter-prover/desertexit/circuits/src/pubdata_market.rs, inner_circuit.rs, witness/utils.go

Description. The desert exit circuit and witness generator perform multi-precision arithmetic operations where intermediate results can exceed allocated limb sizes, causing assertion failures during proof generation. While the circuit correctly rejects invalid proofs (maintaining soundness), legitimate users with extreme but valid account states may be unable to generate proofs for emergency withdrawals, creating a completeness issue. Five distinct overflow scenarios were identified:

1. Position Notional Value Overflow (`pubdata_market.rs:42-51`): The calculation `position × mark_price × quote_multiplier` multiplies a 56-bit position by a 52-bit price factor (32-bit mark price × 20-bit quote multiplier) but constrains the result to `BIG_U64_LIMBS` (64 bits). This requires 108 bits, causing 44 bits of overflow. The `mul_bigint_with_biguint_non_carry` function asserts that limbs beyond the specified size are zero, triggering proof generation failure with maximum documented values.

2. Funding Delta Overflow (`pubdata_market.rs:53-83`): The funding payment calculation computes `funding_multiplier × funding_rate_delta` where funding_multiplier is 76 bits (56-bit position × 20-bit quote_multiplier) and funding_rate_delta is 63 bits. This produces a 139-bit result constrained to `BIG_U96_LIMBS` (96 bits), discarding 43 bits.

3. Witness Position Notional Overflow (`witness/utils.go:319`): The Go witness generator computes `big.NewInt(abs(position.Position) * int64(markPrice) * int64(quoteMultiplier))` where the multiplication occurs in int64 arithmetic (64 bits) before big.Int conversion. With 56-bit position, 32-bit mark price, and 20-bit quote multiplier, this requires 108 bits computed in 64-bit int64, causing 44 bits of overflow. This produces incorrect witnesses that fail circuit verification (DoS), but does not compromise soundness because the circuit independently validates all computations.

4. Pool TAV Calculation Overflows (`inner_circuit.rs:260-315`): Intermediate calculations in pool share valuation multiply `pool_position_value × share_amount` before division by `total_shares` . With pools holding large positions and users having significant share amounts, this multiplication can exceed `BIG_U160_LIMBS` .

5. Extended USDC Multiplication Overflows (`inner_circuit.rs:356-365`): The `usdc_to_extended_usdc` function multiplies amounts by `USDC_TO_COLLATERAL_MULTIPLIER` with caller-specified limb counts. If input amounts approach maximum values for their allocated limbs, the precision scaling can overflow the target limb allocation.

Impact. These overflows do not compromise soundness, since the circuit correctly rejects invalid computations via assertion failures in `mul_biguint_non_carry` and `trim_biguint`, maintaining cryptographic security. However, they may cause completeness issues where users with extreme but valid account states (large positions, high mark prices, significant funding deltas) cannot generate proofs, blocking emergency withdrawals via the Escape Hatch. The Go witness generator (scenario #3) computes incorrect intermediate values due to int64 overflow, producing witnesses that fail circuit verification, also causing denial of service.

Exploitation is unlikely because triggering these overflows requires authenticated on-chain state with extreme values that an attacker cannot directly forge without controlling the main protocol. However, legitimate market conditions such as volatile assets with large positions could trigger these organically. While 96-bit collateral values are unrealistic, the combination of maximum documented type values (56-bit positions, 32-bit mark prices, 20-bit quote multipliers) is within the system's design envelope, and without circuit-level enforcement of tighter business logic bounds, these overflows remain reachable.

The circuit currently relies on implicit assumptions that business logic will constrain values below type maximums, but no circuit-level validation enforces these bounds. The gap between type capacity and realistic business bounds creates a large attack surface for completeness failures, and future changes to market parameters or pricing models could unintentionally trigger these edge cases.

Recommendation. Apply defense-in-depth by increasing limb allocations to accommodate maximum documented type ranges. Alternatively add circuit-level bound checks to validate positions within realistic business bounds (not just type bounds) for higher robustness, and add comprehensive tests that exercise maximum documented type values across all arithmetic operations to ensure sufficient limb allocation post-fix.

Client response. Client will consider addressing this fix after internal evaluation of effort and efficiency impact, given its low severity.

#03 - Signed integer conversions in collateral computation obscure unsigned semantics

Severity: Informational **Location:** lighter-prover/desertexit/witness/utils.go

Description. The witness generator computes signed 96-bit collateral values by assembling two compressed limbs using signed int64 conversions:

```
limb1 := pdd.DecompressTarget()
isNeg := (limb1 & 1) == 1
limb2 := pdd.DecompressTarget()
collateral := big.NewInt(int64(limb2))                                    //
nolint:gosec
collateral.Lsh(collateral, 48).Or(collateral, big.NewInt(int64(limb1>>1))) //
nolint:gosec
if isNeg {
    collateral.Neg(collateral)
}
```

While this code is currently safe because the compression format (`PubdataDecoder.DecompressTarget`) limits each limb to ≤60 bits, using signed conversions for unsigned limb assembly has several drawbacks:

- It obscures the intent that the limbs are semantically unsigned values
- It requires `nolint:gosec` suppressions to silence security linter warnings about potential overflows
- The safety of the int64 cast depends on implicit knowledge of the compression format's 60-bit bound
- Future changes to the encoding format could silently break this assumption

Recommendation. Refactor the limb assembly to use `SetUint64()` for explicit unsigned arithmetic. This makes the code self-documenting by expressing the unsigned intent directly, removes the dependency on the compression format's internal 60-bit bound, and eliminates the need for linter suppressions.

Client response. Fixed in commit `f6818da` on the `spot` branch. The collateral code was replaced with multi-asset balance code that uses `SetUint64()` for explicit unsigned arithmetic as recommended.