Zellic

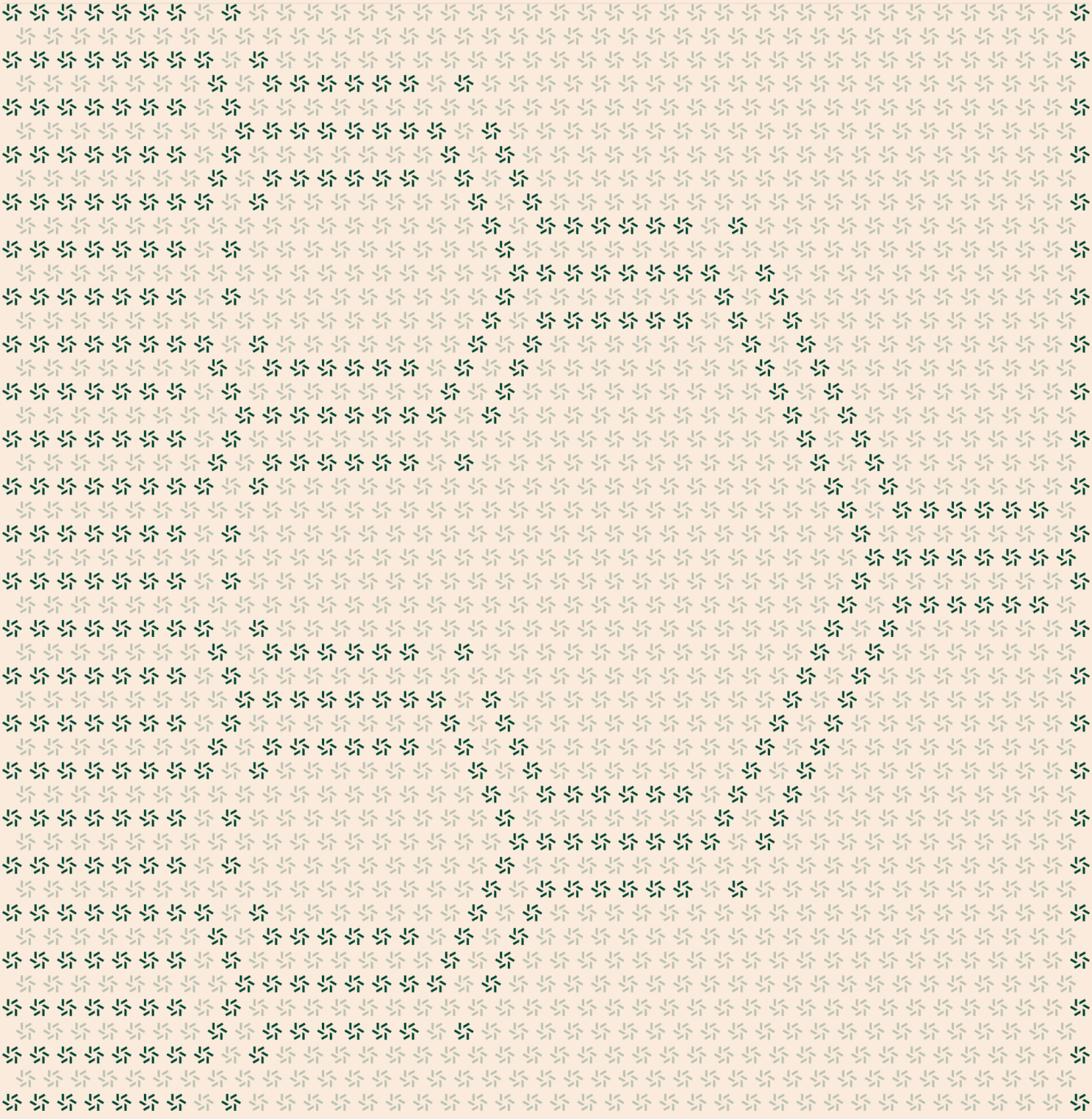
Prepared for
Eugenia Acosta
Celestia

Prepared by
Malte Leip
Mohit Sharma
Zellic

March 4, 2024

Blobstream X

Zero Knowledge 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 infosec 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 Celestia from February 29th to March 4th, 2024. During this engagement, Zellic reviewed Blobstream X'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:

- Assuming soundness of the Tendermint X circuits, do the CombinedSkipCircuit and CombinedStepCircuit circuits prove authenticity of the data commitments based on their trusted inputs as the sole root of trust?

1.3. Non-goals and Limitations

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

- Front-end components
- Infrastructure relating to the project
- On-chain contracts interacting with proofs for the circuits

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

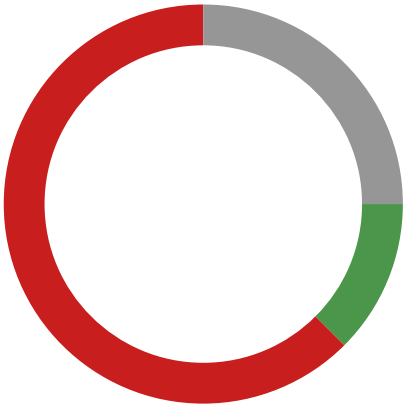
1.4. Results

During our assessment on the scoped Blobstream X circuits, we discovered eight findings. Five critical issues were found. One was of low impact and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	5
High	0
Medium	0
Low	1
Informational	2



2. Introduction

2.1. About Blobstream X

Celestia is a modular data availability network that makes it easy for anyone to securely launch their own blockchain. Celestia is optimized for ordering transaction data and making it available for anyone to download and verify with a light node.

Formerly known as the Quantum Gravity Bridge (QGB), Blobstream relays commitments to Celestia's data root to an on-chain light client, enabling EVM developers to create high-throughput L2s as easily as they develop smart contracts.

2.2. Methodology

During a security assessment, Zellic works through various testing methods along with a manual review. In some cases for a ZKP circuit, we also provide some proofs for soundness. The majority of the time is spent on a 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:

Underconstrained circuits. The most common type of vulnerability in a ZKP circuit is not adding sufficient constraints to the system. This leads to proofs generated with incorrect witnesses in terms of the specification of the project being accepted by the ZKP verifier. We manually check that the set of constraints satisfies soundness, enough to remove all such possibilities, and in some cases, provide a proof of the fact.

Overconstrained circuits. While rare, it is possible that a circuit is overconstrained. In this case, appropriately assigning witness will become impossible, leading to a vulnerability. To prevent this, we manually check that the constraint system is set up with completeness so that the proofs generated with the correct set of witnesses indeed pass the ZKP verification.

Missing range checks. This is a popular type of an underconstrained circuit vulnerability. Due to the usage of field arithmetic, overflow checks and range checks serve a huge purpose to build applications that work over the integers. We manually check the code for such missing checks, and in certain cases, provide a proof that the given set of range checks are sufficient to constrain the circuit up to specification.

Cryptography. ZKP technology and their applications are based on various aspects of cryptography. We manually review the cryptography usage of the project and examine the relevant studies and standards for any inconsistencies or vulnerabilities.

Code Maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards.

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 circuits 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:

Blobstream X Circuits

Repository	https://github.com/succinctlabs/blobstreamx ↗
Version	blobstreamx: `3b3a518a3ff273a56fb0544306cc9c1c4e9602a0`
Programs	• builder.rs L81-L102 • builder.rs L104-L139 • builder.rs L141-L236 • builder.rs L238-L350 • builder.rs L352-L383 • next_header.rs L25-L47 • header_range.rs L32-L54
Type	Rust
Platform	Plonky2x
Audit Fix Version[1]	`c20d19cc8a5f8eca3ea039858847326405e3f346`

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of 1.1 person-weeks. The assessment was conducted over the course of 5 calendar days.

Contact Information

[1] Version including fixes for the findings in this report, as well as other changes that have not been fully reviewed by the Zellic team.

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:

Malte Leip
Engineer
malte@zellic.io ↗

Mohit Sharma
Engineer
mohit@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

February 29, 2024	Kick-off call
February 29, 2024	Start of primary review period
March 4, 2024	End of primary review period

3. Detailed Findings

3.1. Data hash in return value of `prove_next_header_data_commitment` under-constrained

Target	builder		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `prove_next_header_data_commitment` takes trusted inputs `prev_header_hash` and `prev_block_number` and assumes that `prev_header_hash` is the Merkle root hash of the block at height `prev_block_number`. It is intended to return the Merkle root hash of a Merkle tree containing a sole leaf `abi.encode(prev_block_number, DataHash)`, where the second argument is the `DataHash` field of the block with header Merkle-root hash `prev_header_hash`.

However, while the `DataHash` field associated to `prev_header_hash` is constrained to be the protobuf-decoding of `data_comm_proof.data_hash_proofs[0].leaf`, an unconstrained witness `data_comm_proof.data_hashes[0]` is instead used when constructing the return value.

A similar oversight is also present in `prove_subchain`.

The `prove_subchain` function returns a `MapReduceSubchainVariable` containing the Merkle-root hash of the batches' data commitments, calculated as follows:

```
let data_merkle_root = self.get_data_commitment::<BATCH_SIZE>(
    &data_comm_proof.data_hashes,
    batch_start_block,
    end_block_num,
);
```

The leaf data used here is `data_comm_proof.data_hashes`, which is, however, completely unconstrained.

Elsewhere in the function, a Merkle inclusion proof for the `DataHash` field of the enabled Tendermint blocks is checked:

```
let data_hash_proof_root = self
    .get_root_from_merkle_proof::<HEADER_PROOF_DEPTH,
    PROTOBUF_HASH_SIZE_BYTES>(
        &data_comm_proof.data_hash_proofs[i],
        &data_hash_path,
    );
```

Later verification of `data_hash_proof_root` for enabled blocks shows that `data_comm_proof.data_hash_proofs[i].leaf` is the protobuf encoding of the `DataHash` field of the associated Tendermint block. However, there is no check to ensure that `data_comm_proof.data_hashes[i]` is the protobuf decoding of `data_comm_proof.data_hash_proofs[i].leaf`.

Impact

An attacker could generate a proof with arbitrary data hashes of their choosing in the data commitments.

Recommendations

Consider using `data_comm_proof.data_hash_proofs[i].leaf` instead of `data_comm_proof.data_hashes[i]` to construct the data commitments. Note that `data_comm_proof.data_hash_proofs[i].leaf` is the protobuf encoding of the `DataHash` field, so it consists of two bytes of protobuf header followed by the actual 32 bytes of the `DataHash` field.

Remediation

This issue has been acknowledged by Succinct, and a fix was implemented in the following commit: `bbcae1c7` ↗.

3.2. End block's header hash is unconstrained for a subchain proof

Target	DataCommitmentBuilder		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The output for `prove_subchain` circuit contains the following values.

```
MapReduceSubchainVariable {
    is_enabled: is_batch_enabled,
    start_block: batch_start_block,
    start_header: batch_start_header_hash,
    end_block: batch_end_block,
    end_header: batch_end_header_hash,
    data_merkle_root,
}
```

The `end_header` is the header hash of the last block in the subchain. However, the value it contains (i.e., `batch_end_header_hash`) is taken directly from the input and left completely unconstrained.

Impact

A malicous prover can supply an arbitrary value for the end header, which allows for the Blobstream state to differ from the on-chain state.

Recommendations

The loop in `prove_subchain` calculates the header for each block in the chain and verifies chain integrity using Merkle proofs.

```
curr_header = last_block_id_proof_root;
```

At the end of the loop, `curr_header` will contain the hash of the last block, and it should be asserted that this hash is equal to the value being returned.

```
let end_hash_matches_calculated_hash = self.is_equal(batch_end_header_hash,
```

```
    curr_header);
self.assert_is_equal(end_hash_matches_calculated_hash, true_bool);
```

Remediation

This issue has been acknowledged by Succinct, and a fix was implemented in the following commit: 34a9d2eb ↗.

3.3. Inputs to `prove_subchain` are unconstrained

Target	DataCommitmentBuilder		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `prove_data_commitment` makes multiple calls to `prove_subchain` and merges the results together. The inputs for these calls are generated via the hints mechanism.

```
let start_block =
    builder.add(map_ctx.start_block, map_relative_block_nums.as_vec()[0]);
let last_block = builder.add(
    map_ctx.start_block,
    map_relative_block_nums.as_vec()[BATCH_SIZE - 1],
);

// The query_end_block = min(batch_end_block, global_end_block) for fetching
    the data commitment inputs.
// If batch_end_block > global_end_block, data_comm_proof will be filled with
    dummy values.
// This is because prove_subchain only checks up to global_end_block, and doing
    so reduces RPC calls.
let batch_end_block = builder.add(last_block, one);
let past_global_end = builder.gt(batch_end_block, global_end_block);
let query_end_block = builder.select(past_global_end, global_end_block,
    batch_end_block);

// Fetch and read the data commitment inputs.
let mut input_stream = VariableStream::new();
input_stream.write(&start_block);
input_stream.write(&query_end_block);
let data_comm_fetcher = DataCommitmentOffchainInputs::<BATCH_SIZE> {};
let output_stream = builder
    .async_hint(input_stream, data_comm_fetcher);
let data_comm_proof = output_stream
    .read::<DataCommitmentProofVariable<BATCH_SIZE>>(builder);
```

There are constraints enforced on `start_block` and `batch_end_block` before construction. However, once the arguments are constructed in `data_comm_proof`, they are left unconstrained and

passed directly to `prove_subchain`.

Impact

Since the inputs to `prove_subchain` are left completely unconstrained, the prover can violate any assumptions made by the function and cause undefined behavior as demonstrated in Findings 3.6. and 3.7..

Recommendations

Add proper validation logic for consistency between the initial arguments to the map block and the constructed `data_comm_proof` object.

```
builder.assert_is_equal(start_block, data_comm_proof.start_block_height);
builder.assert_is_equal(query_end_block, data_comm_proof.end_block_height);
```

Remediation

This issue has been acknowledged by Succinct, and a fix was implemented in the following commit: 92e913e2.

3.4. End block header is never checked if end block is too far

Target	DataCommitmentBuilder		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `prove_data_commitment` function receives as arguments start and end block heights `start_block` and `end_block` as well as `end_header_hash`, the hash of the header at block height `end_block` (it also receives a block-header hash `start_header_hash`, which is however unused).

The block-header hash `start_header_hash` acts as a root of trust: in the mapping function, `prove_subchain` is called, which will, for the batch in which the block with block height `end_block - 1` lies, check that the next block-header hash of the block in the chain with claimed height `end_block - 1` is equal to the `end_header_hash` that was passed to `prove_data_commitment` (in the following snippet, `global_end_header_hash` is what was passed as `end_header_hash` to `prove_data_commitment`):

```
let root_matches_end_header =
    self.is_equal(last_block_id_proof_root, *global_end_header_hash);
let end_header_check = self.or(is_not_last_block, root_matches_end_header);
self.assert_is_equal(end_header_check, true_bool);
```

However, if `end_block > start_block + NB_MAP_JOBS * BATCH_SIZE`, then no batch will contain the block with block height `end_block - 1`.

The missing check of the last enabled block against a trusted header hash as a root of trust thus means that it is possible to use an arbitrary chain of blocks instead, with data hashes of an attacker's choosing.

Impact

An attacker could generate a proof for the CombinedSkipCircuit in which `target_block > trusted_block + NB_MAP_JOBS * BATCH_SIZE` and return a data commitment with arbitrary data hashes of the attacker's choosing.

Recommendations

Consider adding a constraint in `prove_data_commitment` that verifies `end_block <= start_block + NB_MAP_JOBS * BATCH_SIZE`.

Remediation

This issue has been acknowledged by Succinct, and a fix was implemented in the following commit: `3e2be3f4` ↗.

3.5. Tendermint X TendermintSkipCircuit `skip` function's return value need not be related to parameters

Target	CombinedSkipCircuit		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

In the CombinedSkipCircuit's `define` function, a trusted block height and header hash are provided as public inputs (`trusted_block` and `trusted_header_hash`) as well as a target block height (`target_block`). The target block's header hash is then obtained by using the Tendermint X circuit, TendermintSkipCircuit:

```
let target_header_hash = builder.skip::<MAX_VALIDATOR_SET_SIZE,
    CHAIN_ID_SIZE_BYTES>(
    C::CHAIN_ID_BYTES,
    C::SKIP_MAX,
    trusted_block,
    trusted_header_hash,
    target_block,
);
```

For soundness of the CombinedSkipCircuit, the `skip` function should ensure that, assuming that `trusted_block` and `trusted_header_hash` are legitimate, the `target_header_hash` is the hash of a legitimate Tendermint block with block height `target_block` according to the Tendermint light client consensus rules.

However, the `skip` function is implemented as follows.

```
fn skip<const MAX_VALIDATOR_SET_SIZE: usize, const CHAIN_ID_SIZE_BYTES:
    usize>(
    &mut self,
    chain_id_bytes: &[u8],
    skip_max: usize,
    trusted_block: U64Variable,
    trusted_header_hash: Bytes32Variable,
    target_block: U64Variable,
) -> Bytes32Variable {
    let mut input_stream = VariableStream::new();
    input_stream.write(&trusted_block);
```

```
    input_stream.write(&trusted_header_hash);
    input_stream.write(&target_block);
    let output_stream = self.async_hint(
        input_stream,
        SkipOffchainInputs::<MAX_VALIDATOR_SET_SIZE> {},
    );
    let skip_variable =
    output_stream.read::<VerifySkipVariable<MAX_VALIDATOR_SET_SIZE>>(self);

    let target_header = skip_variable.target_header;

    self.verify_skip::<MAX_VALIDATOR_SET_SIZE, CHAIN_ID_SIZE_BYTES>(
        chain_id_bytes,
        skip_max,
        &skip_variable,
    );
    target_header
}
```

No constraints are introduced by this function itself, only by the following call.

```
self.verify_skip::<MAX_VALIDATOR_SET_SIZE, CHAIN_ID_SIZE_BYTES>(
    chain_id_bytes,
    skip_max,
    &skip_variable,
);
```

Note that before this call, while the arguments `trusted_block`, `trusted_header_hash`, and `target_block` are used to calculate `skip_variable` in this implementation of the prover, there are no constraints introduced in the circuit enforcing a relation between `skip_variable` and these arguments. An attacker can thus use an arbitrary chain of Tendermint blocks that pass the consensus checks — but using a fake initial block `skip_variable.trusted_header` of their choosing — and thereby have wide latitude in what to choose as `skip_variable.target_header` while fuilfilling the `verify_skip` constraints. For example, the attacker could use arbitrary validators they made up to create a chain of Tendermint blocks, with arbitrary data hashes of their choosing included in these blocks.

Impact

The `target_header_hash` used by CombinedSkipCircuit can be set to the hash of a Tendermint block for which the attacker could set the data hash and data hash of preceeding blocks in a valid chain to a value of their choosing. As `target_header_hash` is used as the root of trust for the `prove_data_commitment` function, this ultimately means that the attacker can produce a CombinedSkipCircuit proof for arbitrary data commitments.

Recommendations

In the `skip` function, add constraints that check that `skip_variable.trusted_block`, `skip_variable.trusted_header`, and `skip_variable.target_block` are equal to the parameters `trusted_block`, `trusted_header_hash`, and `target_block` that were passed to the function, respectively.

Remediation

This issue has been acknowledged by Succinct, and a fix was implemented in the following commit: `1800e7c8` ↗.

3.6. Enabled leaves in `get_data_commitment` can underflow

Target	DataCommitmentBuilder		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The following assumption in `get_data_commitment` is wrong.

```
// If end_block < start_block, then this data commitment will be marked as
    disabled, and the
// output of this function is not used. Therefore, the logic assumes
// nb_blocks is always positive.
```

The batch is disabled if and only if `start_block >= global_end_block`, but this function is actually called with `batch_end`, which can still be lower. This means you can end up with a case where `end_block > start_block`.

While this implicit assumption has no significant impact by itself, it can be chained with another unenforced assumption in `get_data_commitment`:

```
let nb_blocks_in_batch = self.sub(end_block, start_block);
// Note: nb_blocks_in_batch is assumed to be less than 2^32 (which is a
    reasonable
// assumption for any data commitment as in practice, the number of blocks in a
    data
// commitment range will be much smaller than 2^32). This is fine as
// nb_blocks_in_batch.limbs[1] is unused.
let nb_enabled_leaves = nb_blocks_in_batch.limbs[0].variable;
```

The number of enabled leaves is directly taken from the first limb, assuming the second to be zero. However, in the case where `end_block > start_block`, `nb_blocks_in_batch` can have an integer underflow.

Impact

A malicous prover can supply an arbitrary value for `batch_end_block` that is less than `batch_start_block`, which causes `nb_enabled_leaves` to underflow and lead to undefined behavior.

This vulnerability is however only exploitable when combined with another vulnerability such as finding 3.3. ↗ or finding 3.8. ↗, and is therefore marked as low impact.

Recommendations

Consider checking that `end_block > start_block` explicitly in `get_data_commitment`:

```
let is_valid = self.gt(end_block, start_block);
self.assert_is_equal(is_valid, true_bool);
```

Additionally, we recommend explicitly constraining the second limb to 0 as a sanity check

```
self.assert_is_equal(nb_blocks_in_batch.limbs[1].variable,
    self.constant::<U64Variable>(0u64));
```

Remediation

This issue has been acknowledged by Succinct, and a fix was implemented in the following commit: `7bb83e77` ↗.

3.7. Discrepancy in length of header chain and data commitment

Target	DataCommitmentBuilder		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	High	**Impact**	Informational

Description

The `prove_subchain` processes blocks in the range from `batch_start_block` (inclusive) to `min(batch_start_block + BATCH_SIZE, global_end_block)` (exclusive). The end of the range given here arises from two facts: processing stops with the block at height `global_end_block`, and the loop starts at `batch_start_block` and runs for `BATCH_SIZE` iterations.

However, when calling `get_data_commitment`, the range passed is the one from `batch_start_block` to `min(batch_end_block, global_end_block)`:

```
let is_less_than_target = self.lte(batch_end_block, *global_end_block);
  let end_block_num = self.select(is_less_than_target, batch_end_block,
    *global_end_block);
  let data_merkle_root = self.get_data_commitment::<BATCH_SIZE>(
      &data_comm_proof.data_hashes,
      batch_start_block,
      end_block_num,
  );
```

If `get_data_commitment` is passed a range larger than the maximum number of leaves, in this case `BATCH_SIZE`, then the return value will be the same as if the range had width equal to the number of leaves. Thus, in practice, the range used to calculate the Merkle root here is the one from `batch_start_block` to `min(batch_start_block + BATCH_SIZE, batch_end_block, global_end_block)`.

For the range used for processing and the range used to construct the data-commitment Merkle tree to be the same, it is thus required that `batch_end_block >= min(batch_start_block + BATCH_SIZE, global_end_block)`. Thus, if `prove_subchain` is called with arguments for which `batch_end_block < min(batch_start_block + BATCH_SIZE, global_end_block)`, then the header hash chain will be verified for a different range than the range used to calculate the corresponding `data_merkle_root`.

Impact

A malicous prover can supply an arbitrary value for `batch_end_block` which is less than `min(batch_start_block + BATCH_SIZE, global_end_block)`, which allows for the `batch_end_header_hash` and `data_merkle_root` to correspond to different blocks in the on-chain state.

This vulnerability is however only exploitable if inputs to the `prove_subchain` function are not constrained properly as in finding 3.3. ↗ and therefore marked as informational.

Recommendations

This can be mitigated by adding a constraint to enforce a batch is of the proper length.

```
let nb_blocks = self.sub(batch_end_block, batch_start_block);
self.assert_is_equal(nb_blocks, BATCH_SIZE);
```

Remediation

This issue has been acknowledged by Succinct, and a fix was implemented in the following commit: `92e913e2` ↗.

3.8. Tendermint X TendermintVerify `verify_skip_distance` function contains ineffective inequality checks

Target	CombinedSkipCircuit		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The Tendermint X TendermintVerify `verify_skip_distance` function is implemented as follows:

```
fn verify_skip_distance(
    &mut self,
    skip_max: usize,
    trusted_block: &U64Variable,
    target_block: &U64Variable,
) {
    let one = self.one();
    let trusted_block_plus_one = self.add(*trusted_block, one);
    // Verify target block > trusted block.
    self.gt(*target_block, trusted_block_plus_one);

    let skip_max_var = self.constant::<U64Variable>(skip_max as u64);
    let max_block = self.add(*trusted_block, skip_max_var);
    // Verify target block <= trusted block + skip_max.
    self.lte(*target_block, max_block);
}
```

This function is intended to check that `target_block > trusted_block` and `target_block <= trusted_block + skip_max`. However, `self.gt(*target_block, trusted_block_plus_one);` only allocates a Boolean circuit variable and assigns it `True` if and only if the `*target_block > trusted_block_plus_one`. This Boolean variable is, however, not constrained to be `True`, so the inequality is not constrained to hold. This is analogous for the second inequality.

Impact

As Tendermint X was out of scope for this audit, we have not evaluated the impact for Tendermint X. With regards to in-scope parts of Blobstream X, this bug, depending on behavior of the Tendermint X `skip` function called as follows,

```
let target_header_hash = builder.skip::<MAX_VALIDATOR_SET_SIZE,
    CHAIN_ID_SIZE_BYTES>(
    C::CHAIN_ID_BYTES,
    C::SKIP_MAX,
    trusted_block,
    trusted_header_hash,
    target_block,
);
```

could allow an attacker to generate a CombinedSkipCircuit proof in which `target_block < trusted_block`.

In this case, the check (seen below) done in `prove_subchain` to check the last header hash against a trusted hash will never be carried out, allowing an attacker to use a chain of arbitrary header hashes of their choosing (note that `prove_data_commitment` never does any checks involving `start_header_hash`, the root of trust for the header hashes is purely the check on `end_header_hash` done in `prove_subchain`).

```
// If this is the last valid block, verify the last_block_id_proof_root (header
    hash of block curr_idx+1) is equal to the global_end_header_hash.
// This is the final step in the verification that global_start_block ->
    global_end_block is linked.
let root_matches_end_header =
    self.is_equal(last_block_id_proof_root, *global_end_header_hash);
let end_header_check = self.or(is_not_last_block, root_matches_end_header);
self.assert_is_equal(end_header_check, true_bool);
```

All batches in `prove_data_commitment` would be disabled, so the reduction steps will always choose the left subchains' data Merkle root. Thus, the data-commitment Merkle root ultimately returned would be the one from the very first batch, where a full batch worth of committed data would be returned, with the data hashes thus choosable by the attacker. See also the description of Finding 3.6. ↗.

Recommendations

As CombinedSkipCircuit exposes `trusted_block` and `target_block` publicly, the check that `target_block > trusted_block` can also be carried out outside the circuit. In this case, the circuit is intended to be used together with an EVM smart contract BlobstreamX.sol, which contains the following check in the `commitHeaderRange` function that appears to prevent providing a proof with `target_block < trusted_block`.

```
if (_targetBlock <= latestBlock || _targetBlock - latestBlock >
    DATA_COMMITMENT_MAX) {
    revert TargetBlockNotInRange();
```

```
}
```

As long as such checks are made by the verifier wherever the proofs of the CombinedSkipCircuit circuit are used, it is not necessary to make in-circuit changes to prevent impact on CombinedSkipCircuit.

As the ineffective inequality checks in Tendermint X's `verify_skip_distance` function were intended to be enforced, we recommend constraining the two Boolean variables returned by `self.gt` and `self.lte` to be `True`.

Remediation

This issue has been acknowledged by Succinct, and a fix was implemented in the following commit: 29945869 ↗

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. Dependency on soundness of Tendermint X circuits

The in-scope CombinedSkipCircuit and CombinedStepCircuit circuits of the Blobstream X project build upon the TendermintSkipCircuit and TendermintStepCircuit circuits of the Tendermint X project in a crucial way. Vulnerabilities in those Tendermint X circuits could significantly impact soundness of the Blobstream X circuits that we audited.

As Tendermint X was not in scope for this audit, we did not review Tendermint X code. However, while trying to answer the question of whether the TendermintSkipCircuit's `skip` function constrains the end-block height to be larger than the start-block height (which was relevant for the in-scope CombinedSkipCircuit circuit, see Finding 3.8. ↗, but not clear from the name), we found two bugs in the Tendermint X circuits, described in Findings 3.5. ↗ and 3.8. ↗, with Finding 3.5. ↗ having critical impact. As we found these issues by coincidence while only skimming a small part of the Tendermint X circuit code, it appears plausible that there could be further significant vulnerabilities in the Tendermint X circuits. We thus recommend to have the TendermintSkipCircuit and TendermintStepCircuit circuits of the Tendermint X project reaudited prior to deploying Blobstream X.

4.2. Confusing variable names

Clear and consistent names for variables and functions help prevent confusion and make it easier to reason about the code. In some parts of the codebase, names could be improved to increase consistency and reduce possibility of confusion.

The `prove_data_commitment` function has an argument called `start_block`. In the closure used as the map function in the `mapreduce` call, a variable called `start_block` occurs as well but with a different meaning — it is the first block of the batch. To reduce confusion, we recommend using more distinct names, such as `global_start_block` for the function argument and `batch_start_block` in the closure.

Variables containing hashes of block headers are inconsistent in the naming scheme, some called `something_header`, others `something_header_hash`.

In many places, names such as `start_block` are used for the height of a block in Tendermint terminology. A name such as `start_block_height` would thus make the content of the variable clearer.

5. Assessment Results

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

During our assessment on the scoped Blobstream X circuits, we discovered eight findings. Five critical issues were found. One was of low impact and the remaining findings were informational in nature. Succinct acknowledged all findings and implemented fixes.

5.1. 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.