Chainway Labs

Citrea

Security Assessment Report

Version: 2.2

July, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Chainway Labs components in scope. The review focused solely on the security aspects of the Rust and Solidity implementations of the Citrea project, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the components in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Chainway Labs components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Chainway Labs components in scope.

Overview

Citrea is a Type 2 zkEVM rollup that aims to scale Bitcoin without requiring changes to the base layer's consensus rules. It processes large batches of transactions off-chain and generates STARK proofs, which are subsequently wrapped into SNARK proofs to attest to their validity.

These proofs, which include the state differences for each batch, are inscribed on the Bitcoin blockchain. This allows any full node to reconstruct the rollup's state, establishing Bitcoin as the Data Availability layer.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the chainwayxyz/citrea repository.

The scope of this time-boxed review was strictly limited to the following files at commit ee7505a:

- `bitcoin-da`
  - `crates/bitcoin-da/src/helpers/parsers.rs`
  - `crates/bitcoin-da/src/verifier.rs`
  - `crates/bitcoin-da/src/helpers/merkle_tree.rs`
- `citrea-stf`
  - `crates/citrea-stf/src/*`
- `evm`
  - including:
    - `crates/evm/*`
    - `crates/evm/src/evm/system_contracts/lib/bitcoin-spv/solidity/contracts/ValidateSPV.sol`
    - `crates/evm/src/evm/system_contracts/lib/bitcoin-spv/solidity/contracts/BTCUtils.sol`
    - `crates/evm/src/evm/system_contracts/lib/WitnessUtils.sol`
  - excluding:
    - `crates/evm/src/smart_contracts/*`
    - `crates/evm/src/signer/*`
    - `crates/evm/src/evm/test_data/*`
    - `crates/evm/src/rpc_helpers/*`
    - `crates/evm/src/evm/system_contracts/lib/*`
- `l2-block-rule-enforcer`
  - `crates/l2-block-rule-enforcer/src/*`
- `light-client-prover-circuit`
  - `crates/light-client-prover/src/circuit/*`
- `primitives`
  - `crates/primitives/src/*`
- `risc0`
  - `crates/risc0/src/guest.rs`
  - `guests/risc0/*`
- `short-header-proof-provider`

  - `crates/short-header-proof-provider/*`

- `sovereign-sdk-integration`
  - `crates/sovereign-sdk/rollup-interface/src/state_machine/transaction.rs`
  - `crates/sovereign-sdk/rollup-interface/src/state_machine/block.rs`
  - `crates/sovereign-sdk/rollup-interface/src/fork/*`
  - `crates/sovereign-sdk/module-system/sov-keys/*`
  - `crates/sovereign-sdk/module-system/sov-modules-stf-blueprint/*`

The fixes for the identified issues were assessed at commit 650ed94.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity and Rust.

For the Solidity components, the manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team may use the following automated testing tools:

- Aderyn: `https://github.com/Cyfrin/aderyn`
- Slither: `https://github.com/trailofbits/slither`
- Mythril: `https://github.com/ConsenSys/mythril`

For the Rust components, the manual review focused on identifying issues associated with the business logic implementation of the components in scope. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Rust language.

Additionally, the manual review process focused on identifying vulnerabilities related to known Rust anti-patterns and attack vectors, such as unsafe code blocks, integer overflow, floating point underflow, deadlocking, error handling, memory and CPU exhaustion attacks, and various panic scenarios including index out of bounds, `panic!()`, `unwrap()`, and `unreachable!()` calls.

To support the Rust components of the review, the testing team may use the following automated testing tools:

- Clippy linting: `https://doc.rust-lang.org/stable/clippy/index.html`

- Cargo Audit: `https://github.com/RustSec/rustsec/tree/main/cargo-audit`
- Cargo Outdated: `https://github.com/kbknapp/cargo-outdated`
- Cargo Geiger: `https://github.com/rust-secure-code/cargo-geiger`
- Cargo Tarpaulin: `https://crates.io/crates/cargo-tarpaulin`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 24 issues during this assessment. Categorised by their severity:

- High: 5 issues.
- Medium: 6 issues.
- Low: 4 issues.
- Informational: 9 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Chainway Labs components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the components, including optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
CTR-01	Duplicate Deposit Transactions Can Halt Block Production	High	Resolved
CTR-02	Inconsistent Witness Commitment Handling Can Crash Light Client Prover	High	Resolved
CTR-03	Incorrect Simulation Of Deposit Transactions Can Lead To Chain Halt Or Invalid Rejection	High	Resolved
CTR-04	Chain Halt Via Unbounded Return Data In `deposit()` Function	High	Resolved
CTR-05	Uncontrolled Decompression In `decompress_blob()` Can Lead To Denial Of Service	High	Resolved
CTR-06	Security Bypass In System Transaction Verification For First L2 Block	Medium	Resolved
CTR-07	Denial Of Service In System Transaction Verification	Medium	Resolved
CTR-08	Potential Mutex Poisoning In `NativeShortHeaderProofProviderService`	Medium	Resolved
CTR-09	Malicious Sequencer Can Cause Chain Split By Forging `prev_hash`	Medium	Resolved
CTR-10	Malicious Sequencer And Prover Can Forge L1 State On `BitcoinLightClient`	Medium	Closed
CTR-11	Missing Finality Proof Allows Prover To Stall L2 Chain For Light Clients	Medium	Closed
CTR-12	Incorrect Handling Of KZG Point Evaluation Precompile (0x0A) In Cancun Fork	Low	Closed
CTR-13	Potential Over-Minting of `cBTC` Due To Missing `depositAmount` Check	Low	Resolved
CTR-14	Merkle Tree Hash Collision Due To Lack Of Domain Separation	Low	Resolved
CTR-15	Incorrect Check For Schnorr Signature Validity	Low	Resolved
CTR-16	Missing Equality Check In `apply_timestamp_rule()`	Informational	Closed
CTR-17	Handle Unsupported EIP-4844 Transactions Gracefully During Execution	Informational	Closed
CTR-18	Redundant Sequencer Public Key Check In `begin_l2_block()`	Informational	Resolved
CTR-19	Missing Check for Empty `batch_proof_method_ids`	Informational	Resolved
CTR-20	Use Of `wrapping_sub` Allows Silent Overflow In `spec_id` Comparison	Informational	Resolved
CTR-21	Inconsistent Endianness In Transaction Serialisation	Informational	Resolved
CTR-22	Incorrect L2 Height Used For Cache Pruning Check	Informational	Resolved
CTR-23	Inconsistent Endianness In `safeWithdraw()` Function	Informational	Resolved
CTR-24	Miscellaneous General Comments	Informational	Resolved

CTR-01	Duplicate Deposit Transactions Can Halt Block Production		
Asset	`crates/sequencer/src/rpc.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

An attacker can halt Citrea block production by submitting duplicate deposit transactions.

Deposit transactions are treated as system transactions and are executed at the beginning of block execution. For a block to be considered valid, all system transactions must succeed:

crates/evm/src/evm/executor.rs::execute_multiple_tx()

```
if !*should_be_end_of_sys_txs && !result_and_state.result.is_success() {
    native_error!(
        "System transaction not successful. Result: {:?}",
        result_and_state.result
    );
    return Err(L2BlockModuleCallError::EvmSystemTransactionNotSuccessful);
}
```

A malicious actor can exploit this by submitting the same deposit transaction twice in quick succession using `send_raw_deposit_transaction()` call. This function uses `evm.get_call()` to validate whether the deposit transaction would succeed before it is added to the `deposit_mempool`. Since both submissions are validated against the same EVM state from the latest block, they are both seen as valid and added to the mempool.

However, during execution via `evm.transact()`, the second deposit transaction would fail because the deposit has already been processed. This failure is triggered by the replay protection mechanism in the `deposit()` function of the Bridge contract.

crates/evm/src/evm/system_contracts/src/Bridge.sol::deposit()

```
function deposit(
    // ...

    // Nullify the move transaction based on txId
    bytes32 txId = ValidateSPV.calculateTxId(moveTx.version, moveTx.vin, moveTx.vout, moveTx.locktime);
    require(!processedTxIds[txId], "txId already spent");
    processedTxIds[txId] = true;
    depositTxIds.push(txId);

    // ...
}
```

As a result, the block becomes invalid due to the failure of a system transaction, effectively halting block production.

The impact is rated high, as the block production process can be completely halted. The likelihood is rated medium, since the `send_raw_deposit_transaction()` RPC call is not currently available to regular users, but it is intended to be opened in the future.

Recommendations

Modify implementation to ignore the failed deposit transactions during block production process.

Resolution

Failed deposit transactions are now ignored during the dry-running process and not included in the block.

The development team has fixed this issue in PR #2406.

CTR-02	Inconsistent Witness Commitment Handling Can Crash Light Client Prover		
Asset	`crates/bitcoin-da/src/verifier.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A malicious Bitcoin miner can crash the light client prover by crafting a coinbase transaction with a fake witness commitment output and witness reserved value.

In the `verify_transactions()` function, the code attempts to access the witness reserved value by calling `unwrap()` on the first element of the coinbase transaction's witness stack. If the witness stack is empty, calling `unwrap()` will cause a panic.

crates/bitcoin-da/src/verifier.rs::verify_transactions()

```
Some(mut commitment_idx) => {
    let merkle_root =
        merkle_tree::BitcoinMerkleTree::new(inclusion_proof.wtxids).root();

    let input_witness_value = coinbase_tx.input[0].witness.iter().next().unwrap();

    let mut vec_merkle = Vec::with_capacity(input_witness_value.len() + 32);
    // ...
}
```

Bitcoin Core validates the witness stack using the `CheckWitnessMalleation()` function, but only if a valid witness commitment is detected in the coinbase transaction. It determines validity using `GetWitnessCommitmentIndex()`, which checks two conditions:

1. The script starts with `WITNESS_COMMITMENT_PREFIX`.
2. The script length is at least `MINIMUM_WITNESS_COMMITMENT` (i.e., 38 bytes).

However, in `verify_transactions()`, the code that locates the commitment output only checks for the prefix and does not enforce the minimum length:

crates/bitcoin-da/src/verifier.rs::verify_transactions

```
// @audit missing minimum length check
let commitment_idx = coinbase_tx.output.iter().rev().position(|output| {
  output
      .script_pubkey
      .as_bytes()
      .starts_with(WITNESS_COMMITMENT_PREFIX)
});
```

This discrepancy allows an attacker to craft a Coinbase output that starts with `WITNESS_COMMITMENT_PREFIX` but is shorter than the required length, along with an empty witness field. In this case, Bitcoin Core would skip validation of the witness stack, while `verify_transactions()` would treat it as a valid commitment. As a result, when `verify_transactions()` tries to access the witness reserved value, it panics due to the missing witness data.

Furthermore, even if a witness reserved value is present, as the crafted witness commitment size in the output is less than 32 bytes, it leads to a panic due to an out-of-bounds access.

crates/bitcoin-da/src/verifier.rs::verify_transactions

```
if script_pubkey[6..38] != commitment {
  return Err(ValidationError::IncorrectWitnessCommitment);
}
```

The impact of this issue is high, as a malicious Bitcoin block can cause the light client prover to crash. The likelihood is medium, since the attack can only be carried out by Bitcoin block builders.

Recommendations

Add a check to ensure the commitment output is at least `MINIMUM_WITNESS_COMMITMENT` bytes long, making the implementation consistent with Bitcoin Core's `GetWitnessCommitmentIndex()` function.

Resolution

The `verify_transactions()` function has been updated to check for the minimum length of the witness commitment output as recommended above.

The development team has fixed this issue in PR #2422.

CTR-03	Incorrect Simulation Of Deposit Transactions Can Lead To Chain Halt Or Invalid Rejection		
Asset	`crates/sequencer/src/rpc.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The current implementation simulates deposit transactions using the block environment of the latest block, rather than the anticipated environment of the next pending block. If a system-critical deposit transaction that passed simulation, but fails on execution is included, it could halt the chain.

The issue lies within the `send_raw_deposit_transaction()` function of sequencer's `rpc.rs`. When a deposit transaction is received, it is simulated using `evm.get_call()`:

crates/sequencer/src/rpc.rs::send_raw_deposit_transaction()

```
fn send_raw_deposit_transaction(&self, deposit: Bytes) -> RpcResult<()> {
    // ...
    let mut working_set = WorkingSet::new(self.context.storage.clone());

    // ...

    // @audit `block_id` is `None', so the latest block environment is used
    let tx_res = evm.get_call(dep_tx, None, None, None, &mut working_set);

    match tx_res {
        Ok(hex_res) => {
            tracing::debug!("Deposit tx processed successfully {}", hex_res);
            self.context
                .deposit_mempool
                .lock()
                .add_deposit_tx(deposit.to_vec());
            Ok(())
        }
        // ...
    }
}
```

The `evm.get_call()` method, when not provided with a specific block environment, will default to the state and environment of the latest block. If a deposit transaction's validity is dependent on `block.number`, `block.timestamp` or other environment variables that change per block, this simulation can be inaccurate. For example, a transaction might be valid at `block.number = N` but invalid at `block.number = N+1`.

This issue is classified as high impact because it can break core functionality by either halting the chain if a failing system transaction is admitted, or by preventing valid user deposits if the deposit is valid but fails the simulation. The likelihood is medium as any deposit transaction sensitive to block-specific parameters could trigger this behaviour, without requiring special conditions.

Recommendations

Modify the simulation of deposit transactions within `send_raw_deposit_transaction()` such that it is performed using the `Pending` block, not `Latest`.

Resolution

The `send_raw_deposit_transaction()` function now calls `evm.get_call()` with the `Pending` block environment.

The development team has fixed this issue in PR #2526.

CTR-04	Chain Halt Via Unbounded Return Data In `deposit()` Function		
Asset	`crates/evm/src/evm/system_contracts/src/Bridge.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `deposit()` function in `Bridge.sol` is vulnerable to an out-of-gas (OOG) condition if the `recipient` address is a malicious contract that returns an excessively large amount of data. This OOG occurs in the `Bridge` contract itself when attempting to send `cBTC` to the recipient, causing the entire `deposit()` transaction to revert. Since `deposit()` is triggered by a system call, its failure leads to a chain halt.

The `deposit()` function processes deposits from Bitcoin. It validates the Bitcoin transaction and then attempts to transfer `depositAmount` of `cBTC` to the `recipient` address specified in the Bitcoin transaction's script. If this transfer fails, the funds are intended to be sent to a `failedDepositVault`. The relevant code is:

crates/evm/src/evm/system_contracts/src/Bridge.sol::deposit()

```
address recipient = extractRecipientAddress(script);

(bool success, ) = recipient.call{value: depositAmount}("");
if(!success) {
    // If the transfer fails, we send the funds to the failed deposit vault
    emit DepositTransferFailed(wtxId, txId, recipient, block.timestamp, depositTxIds.length - 1);
    (success, ) = failedDepositVault.call{value: depositAmount}("");
    require(success, "Failed to send to failed deposit vault");
} else {
    emit Deposit(wtxId, txId, recipient, block.timestamp, depositTxIds.length - 1);
}
```

A malicious user can craft a Bitcoin deposit transaction that designates their own smart contract on Citrea as the `recipient`. If this malicious recipient contract, upon receiving the `depositAmount` via the low-level call, executes successfully but returns a very large amount of data (a "return bomb"), the EVM attempts to copy this data into the `Bridge` contract's memory. Even though the return data is not explicitly assigned to a variable in the `Bridge` contract's code, the process of handling this large return data can consume all remaining gas in the `Bridge` contract. While the 1/64th gas rule is designed to leave enough gas for the caller to handle post-call logic, processing return values from low-level calls occurs in the caller's context, using up the reserved gas. As such, this results in the entire deposit transaction failing.

Since deposits to the bridge are system transactions and are expected to always succeed, this would lead to a chain halt, as indicated by Citrea's executor logic:

crates/evm/src/evm/executor.rs::execute_multiple_tx()

```
if !*should_be_end_of_sys_txs && !result_and_state.result.is_success() {
    native_error!(
        "System transaction not successful. Result: {:?}",
        result_and_state.result
    );
    return Err(L2BlockModuleCallError::EvmSystemTransactionNotSuccessful);
}
```

The impact is high because this vulnerability can lead to a chain halt, breaking core protocol functionality. The likelihood is medium as it requires a malicious actor to make a deposit into Citrea's bridge, such that they will lose their deposit as it cannot be recovered after the chain halts.

Recommendations

To mitigate this vulnerability, the call to the `recipient` in the `deposit()` function should be made using a mechanism that is resilient to "return bombs", which involves checking the size of the return data before attempting to copy it into memory. If the return data size is excessive, the operation can be aborted or treated as a failure without causing an OOG error in the calling contract.

Nomad's ExcessivelySafeCall provides functions for making external calls that safely handle large return data, preventing the OOG condition described.

Resolution

Failing deposit transactions are now not included in a block.

The development team has fixed this issue in PR #2406.

CTR-05	Uncontrolled Decompression In `decompress_blob()` Can Lead To Denial Of Service		
Asset	`crates/primitives/src/compression.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `decompress_blob()` function, used for decompressing brotli-compressed data, does not enforce any limits on the size of the output. This makes the system vulnerable to "decompression bombs," where a small, maliciously crafted compressed input can expand to an extremely large size upon decompression, potentially exhausting system memory and leading to a denial of service (DoS).

crates/primitives/src/compression.rs::decompress_blob()

```
pub fn decompress_blob(blob: &[u8]) -> Result, io::Error> {
    use brotli::DecompressorWriter;
    let mut writer = DecompressorWriter::new(Vec::new(), 4096);
    writer.write_all(blob)?; // @audit No limit on decompressed size
    match writer.into_inner() {
        Ok(result) => Ok(result),
        Err(_) => Err(io::Error::new(
            io::ErrorKind::InvalidData,
            "Brotli decompression failure",
        )),
    }
}
```

The `decompress_blob()` function is utilised in processing batch proofs submitted in Bitcoin transactions. A malicious batch prover could submit a specially crafted compressed proof designed to trigger this vulnerability. When the system attempts to decompress this proof using `decompress_blob()`, it could consume an excessive amount of memory, causing the light client prover to slow down, become unresponsive, or crash.

This impact is high because it could lead to a DoS for light client provers that need to parse these batch proofs. The likelihood is rated medium as `Complete` batch proofs are decompressed before any pubkey-based filtering in `crates/bitcoin-da/src/verifier.rs::verify_transactions()`, allowing anyone to perform this attack.

Recommendations

Implement a limit on the maximum allowed size of the decompressed output in the `decompress_blob()` function and decompress blobs in chunks of a reasonable size (not to be confused with the batch proof chunks).

During the decompression process, the function should check if the output has exceeded a predefined reasonable threshold. If the threshold is exceeded, the decompression should be aborted.

Resolution

A 100 MiB limit on decompressed blobs was implemented, and the decompressed data chunks are now read into a 4KiB buffer.

`Complete` batch proofs are now also decompressed after the pubkey-based filtering, matching the behaviour of `Aggregate` batch proofs.

The development team has fixed this issue in PR #2482.

CTR-06	Security Bypass In System Transaction Verification For First L2 Block		
Asset	`crates/evm/src/evm/executor.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `verify_system_tx()` function in `executor.rs` unconditionally bypasses any checks for any system transaction processed in the first L2 block. This allows potentially invalid data, such as a fraudulent L1 block hash and witness root, to be committed to the `BitcoinLightClientContract`, compromising the L2's view of the L1 chain from its inception.

The function `verify_system_tx()` contains an early exit condition on the first L2 block as shown below:

crates/evm/src/evm/executor.rs::verify_system_tx()

```
fn verify_system_tx(
    db: &mut EvmDb,
    tx: &Recovered,
    l2_height: u64,
) -> Result<(), L2BlockModuleCallError> {
    // Early return if this is the first block because sequencer will not have any L1 block hash in system contract before setblock
      ↪  info call
    if l2_height == 1 {
        // @audit This unconditionally skips SHP verification for all system transactions in the first L2 block.
        return Ok(());
    }
    // ...
    if function_selector == BitcoinLightClientContract::setBlockInfoCall::SELECTOR {
        // ...
        match shp_provider.get_and_verify_short_header_proof_by_l1_hash(
            // ...
        ) {
            // ...
        }
    }
}
```

System transactions that call `BitcoinLightClientContract::setBlockInfoCall()` in the first L2 block will not perform any Short Header Proof (SHP) verification normally performed by `get_and_verify_short_header_proof_by_l1_hash()`. This allows a `setBlockInfoCall` in the first L2 block to submit arbitrary block hash and witness root values without any validation against the L1 chain's actual state. This can lead to a compromised light client state on L2, potentially allowing for arbitrary deposits to be made through the bridge.

The impact is high because allowing unverified data into the `BitcoinLightClientContract` can break core L2 functionality that depends on an accurate L1 view, and could potentially lead to draining the bridge. The likelihood is low as it is only possible to be exploited by a malicious sequencer on the first L2 block.

Recommendations

Ensure that SHP verification occurs for the first `BitcoinLightClient.setBlockInfo()` call.

Resolution

The `l2_height == 1` check in `verify_system_tx()` has been removed and SHP verification is now performed for the first block.

The development team has fixed this issue in PR #2415.

CTR-07	Denial Of Service In System Transaction Verification		
Asset	`crates/evm/src/evm/executor.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `verify_system_tx()` function in `executor.rs` is susceptible to a denial of service (DoS) vulnerability. If the Bitcoin light client's initial L1 block hash is not set within the very first L2 block, all subsequent attempts to process system transactions will result in a panic, effectively halting critical L2 operations.

Consider the code snippet below:

crates/evm/src/evm/executor.rs::verify_system_tx()

```
fn verify_system_tx(
    db: &mut EvmDb,
    tx: &Recovered,
    l2_height: u64,
) -> Result<(), L2BlockModuleCallError> {
    // Early return if this is the first block because sequencer will not have any L1 block hash in system contract before setblock
      ↪  info call
    if l2_height == 1 {
        return Ok(());
    }
    // ...
    if function_selector == BitcoinLightClientContract::setBlockInfoCall::SELECTOR {
        // ...
        let (_last_l1_height, prev_hash) =
            get_last_l1_height_and_hash_in_light_client::(db.evm, db.working_set);

        // ...
        match shp_provider.get_and_verify_short_header_proof_by_l1_hash(
            l1_block_hash.0,
            // @audit This unwrap will panic if prev_hash is None and l2_height > 1
            prev_hash.unwrap().to_be_bytes(),
            next_l1_height,
            txs_commitment.0,
            coinbase_depth,
            l2_height,
        ) {
            // ...
        }
    }
    Ok(())
}
```

The function `verify_system_tx()` is invoked before processing any system transaction. For transactions targeting `BitcoinLightClientContract::setBlockInfoCall()`, it retrieves `prev_hash` from the light client. If the L2 height (`l2_height`) is greater than 1 and `prev_hash` is `None`, the code panics when trying to unwrap `prev_hash`. This occurs if `setBlockInfoCall()` is not called in the first L2 block, as short header proofs are only skipped in the first block.

This panic will prevent the current system transaction from being processed and, critically, will occur for any subsequent system transaction, leading to a permanent DoS. This DoS blocks system-critical components, including batch proofs and L2 deposits, which rely on the `BitcoinLightClientContract`.

The impact is high as this vulnerability leads to a DoS of all `setBlockInfoCall()` system transactions, breaking core L2 functionality. The likelihood is low because this DoS condition only occurs if the first L2 block does not include a `setBlockInfoCall()` system transaction.

Recommendations

Modify `verify_system_tx()` to gracefully handle the scenario where `prev_hash` is `None` when `l2_height > 1`. For `BitcoinLightClientContract::setBlockInfoCall` transactions, if there is no `prev_hash`, a default zero hash should be passed to `get_and_verify_short_header_proof_by_l1_hash()`.

Resolution

The `get_last_l1_height_and_hash_in_light_client()` function has been modified to return a default zero hash if the storage slot is empty as recommended above.

The `l2_height == 1` check in `verify_system_tx()` has also been removed.

The development team has fixed this issue in PR #2415.

CTR-08	Potential Mutex Poisoning In `NativeShortHeaderProofProviderService`		
Asset	`crates/short-header-proof-provider/src/native.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `NativeShortHeaderProofProviderService` is vulnerable to a denial-of-service attack due to improper handling of mutex locks. Panics within critical sections, such as from out-of-memory errors during vector allocations, can poison a mutex, causing subsequent lock attempts to panic and crash the service.

The `NativeShortHeaderProofProviderService` utilizes a mutexed hashmap named `queried_and_verified_hashes` to store verified block hashes. Critical sections accessing this shared data employ `lock().unwrap()` or `lock().expect()` for acquiring the mutex. If a thread panics while holding this lock, for instance, due to an out-of-memory (OOM) error during a vector allocation, the mutex becomes poisoned. Consequently, any subsequent attempt by another thread to acquire the lock via `unwrap()` or `expect()` will also panic, leading to a halt in the service's operation and a denial of service for the node.

The primary concern lies with operations that can allocate memory (and thus potentially panic on OOM) within the locked critical section. Specific instances include:

1. In `get_and_verify_short_header_proof_by_l1_hash()`:

   crates/short-header-proof-provider/src/native.rs::get_and_verify_short_header_proof_by_l1_hash()

   ```
   self.queried_and_verified_hashes
       .lock()
       .expect("Should lock queried and verified hashes") // @audit Problematic expect()
       .entry(l2_height)
       .and_modify(|f| f.push(block_hash)) // @audit Potential OOM panic here
       .or_insert(vec![block_hash]);    // @audit Potential OOM panic here
   ```

   Both the `push()` operation within `and_modify()` and the creation of `vec![block_hash]` in `or_insert()` can cause allocations that might panic.

2. In `take_queried_hashes()`:

   crates/short-header-proof-provider/src/native.rs::take_queried_hashes()

   ```
   let queried_and_verified_hashes = self.queried_and_verified_hashes.lock().unwrap(); // @audit Problematic unwrap()
   let mut hashes = Vec::new();
   for l2_height in l2_range {
       if let Some(hash_list) = queried_and_verified_hashes.get(&l2_height) {
           hashes.extend(hash_list.clone()); // @audit Potential OOM panic during extend
       }
   }
   ```

   The `hashes.extend(hash_list.clone())` operation can panic due to OOM if the `hashes` vector needs to reallocate.

3. In `clear_queried_hashes()`:

   short-header-proof-provider/src/native.rs::clear_queried_hashes

   ```
   self.queried_and_verified_hashes.lock().unwrap().clear(); // @audit Problematic unwrap()
   ```

While `clear()` itself is unlikely to OOM-panic, the `unwrap()` will panic if the mutex was previously poisoned by operations in other functions.

An OOM error during these critical sections would poison the `queried_and_verified_hashes` mutex. Any subsequent call to `get_and_verify_short_header_proof_by_l1_hash()`, `clear_queried_hashes()`, or `take_queried_hashes()` would then result in a panic.

This issue has a high impact as it could lead to a denial of service, impacting core protocol functionality. The likelihood is low, as it relies on an OOM condition occurring during a very specific, locked operation.

Recommendations

To mitigate this issue, the following changes are recommended:

1. Gracefully handle mutex poisoning by replacing all instances of `lock().unwrap()` and `lock().expect()` with more explicit error handling. This allows the system to react to a poisoned mutex in a controlled manner, such as by logging the error and attempting recovery or returning a specific error to the caller.
2. For operations that involve memory allocation on data protected by the mutex (e.g., `Vec::push`, `Vec::extend`), ensure that allocation failures are handled as errors rather than panics. This can be achieved by using `try_reserve()` to pre-allocate memory before performing operations that might reallocate the vector.

Resolution

The development team has fixed this issue and implemented the recommended changes in PRs #2465 and #2524.

CTR-09	Malicious Sequencer Can Cause Chain Split By Forging `prev_hash`		
Asset	`crates/sovereign-sdk/module-system/sov-modules-stf-blueprint/src/lib.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `apply_l2_blocks_from_sequencer_commitments()` function fails to validate the `prev_hash` of the first L2 block within a batch of sequencer commitments. A malicious or compromised sequencer can exploit this to create a block that does not correctly link to the previous block in the chain, yet still obtain a valid state transition proof for it. This leads to a critical chain split, where light clients and full nodes diverge on the canonical chain.

The relevant code snippet is as follows:

sovereign-sdk/module-system/sov-modules-stf-blueprint/src/lib.rs::apply_l2_blocks_from_sequencer_commitments

```
// @audit `prev_l2_block_hash` is initialised to `None`.
let mut prev_l2_block_hash: Option<[u8; 32]> = None;

// ...

for sequencer_commitment in sequencer_commitments.into_iter() {
    // ...
    for _ in 0..state_change_count {
        // ...
        let (l2_block, state_witness, offchain_witness) =
                guest.read_from_host::<(L2Block, Witness, Witness)>();
        // ...
        // @audit This `if let` is false for the first L2 block because `prev_l2_block_hash` is `None`.
        // @audit As a result, `l2_block.prev_hash()` is not checked for the first block.
        if let Some(hash) = prev_l2_block_hash {
                assert_eq!(
                l2_block.prev_hash(),
                hash,
                "L2 block previous hash must match the hash of the block before"
                );
        }
        // ...
        prev_l2_block_hash = Some(l2_block.hash());
        // ...
    }
    // ...
}
```

Assume the canonical chain has been proven up to block `B`. The next block `C` should have `C.height = B.height + 1` and `C.prev_hash = hash(B)`. A malicious sequencer can instead construct and submit a fraudulent block `C'` with `C'.height = B.height + 1` but with `C'.prev_hash` pointing to an arbitrary hash instead of `hash(B)`. When the zkVM guest (the prover) processes the batch containing `C'`, the missing validation on the first block allows `C'` to be processed. The prover applies `C'`'s state transition on top of block `B`'s state and generates a valid ZK proof, as other checks like height and state root consistency can still pass.

Light clients, which validate state via ZK proofs, will accept this transition. However, full nodes, which validate the entire block, will detect the incorrect `prev_hash` in `C'` and reject it. This results in a permanent fork between different node types, compromising the chain's integrity.

The impact is high, as this can break the fundamental assumption of a single canonical chain for all participants. The likelihood is low, as it requires a malicious or compromised sequencer, which is a trusted role in the system architecture.

Recommendations

To fix this vulnerability, the `apply_l2_blocks_from_sequencer_commitments()` function must validate the `prev_hash` of the first L2 block in every batch. This can be achieved by tracking the last L2 block hash from the previous batch proof and using it to initialise the check for the current batch proof.

The `prev_l2_block_hash` variable should be initialised with the hash of the last block from the previous batch proof, or a genesis-defined parent hash if it is the very first batch.

Keep in mind that the light client prover will also need to verify the `prev_l2_block_hash` to ensure that the chain is unbroken.

Resolution

The `apply_l2_blocks_from_sequencer_commitments()` function now initialises the `prev_l2_block_hash` with the hash of the last block by taking in a `prev_hash_proof`. The provided hash is verified against the previous sequencer commitment's merkle root to ensure that the hash is correct.

The development team has fixed this issue in PRs #2469, #2493 and #2522.

CTR-10	Malicious Sequencer And Prover Can Forge L1 State On `BitcoinLightClient`		
Asset	`crates/bitcoin-da/src/spec/short_proof.rs, crates/citrea-stf/src/verifier.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

A malicious sequencer, in collusion with a malicious batch prover, can provide a fraudulent Short Header Proof (SHP) to the State Transition Function. This allows the attacker to corrupt the L2's onchain `BitcoinLightClient` with a forged L1 state, compromising the security of the L2 and its bridge, ultimately leading to the theft of funds locked in the bridge.

The root cause of this vulnerability lies in the `verify()` function within `short_proof.rs`. While this function checks for the internal consistency of a header and the inclusion of the coinbase transaction, it critically fails to verify that the provided block satisfies the following criteria:

1. The block hash is valid against the Proof-of-Work difficulty target specified in the header's `bits` field.
2. The header's `bits` field is correct and is consistent with the current epoch's target.
3. The block is "final" and not prone to being re-orged through short-range fork attacks.

crates/bitcoin-da/src/spec/short_proof.rs::verify()

```
    fn verify(&self) -> Result {
        // First verify that the precomputed (from circuit input) hash actually matches
        // the hash of the header
        if !self.header.verify_hash() {
            return Err(ShortHeaderProofVerificationError::InvalidHeaderHash);
        }

        // @audit There is no check here to verify that the header's hash
        // meets the difficulty target specified in `self.header.inner().bits`.
        // An attacker can create a header with a trivial difficulty and find a
        // valid nonce for it instantly, allowing for complete forgery of the proof.

        // ...
// ... existing code ...
```

The omission of checks (1) and (2) allows an attacker to "mine" a block with a trivially easy difficulty on a local machine. This block can contain any transactions the attackers desire, such as a fake `move_tx` that creates millions of dollars' worth of BTC out of thin air. The omission of check (3) allows an attacker to build off of the last finalised block and create a new chain. Even if this chain is shorter than the main chain, it can still be submitted as a short header proof.

This exploit requires collusion between the sequencer and the batch prover. The sequencer must craft an L2 block containing a `setBlockInfo` system transaction pointing to their forged L1 block. The batch prover must then provide the corresponding forged `ShortHeaderProof` when generating the ZK proof. This ensures that the batch prover's block execution is consistent with the sequencer, allowing the malicious block to pass the state root and block hash checks in `apply_l2_blocks_from_sequencer_commitments()`:

sovereign-sdk/module-system/sov-modules-stf-blueprint/src/lib.rs::apply_l2_blocks_from_sequencer_commitments

```
// @audit the pre and post state root checks will match up as the sequencer and batch prover
assert_eq!(current_state_root, result.state_root_transition.init_root);
// ...
assert_eq!(current_state_root, l2_block.state_root());
// ...
// @audit the batch prover's calculated root will match with the sequencer's
let calculated_root =
    MerkleTree::<Sha256>::from_leaves(l2_block_hashes.as_slice()).root();

assert_eq!(
    calculated_root,
    Some(sequencer_commitment.merkle_root),
    "Invalid merkle root"
);
```

It is important to keep in mind that although the light client prover checks the last L1 hash on the `BitcoinLightClient` before processing the batch proof, it does not check previous L1 hashes in the same batch proof, allowing this exploit to still be performed as long as it is followed by a valid SHP and L1 hash on the `BitcoinLightClient` in the same batch proof.

crates/light-client-prover/src/circuit/mod.rs::process_complete_proof()

```
if !BlockHashAccessor::<S>::exists(
    batch_proof_output.last_l1_hash_on_bitcoin_light_client_contract(),
    working_set,
) {
    return Err("Batch proof with unknown header chain");
}
```

This issue has a high impact as it allows the colluded actors to fabricate L1 state, allowing them to drain the bridge of all its `cBTC`. The likelihood is rated as low because it is only possible with a colluding sequencer and batch prover, which are both trusted entities.

Recommendations

Two changes are critical to securing the protocol:

1. The `verify()` function in `short_proof.rs` must be updated to check that the block header's hash is less than or equal to the difficulty target derived from the `bits` field and that the `bits` field is consistent with the current epoch's target. This can be done with `bitcoin-da/src/verifier.rs::BitcoinVerifier::verify_header_chain()`.
2. To prevent attacks using blocks from transient forks of the real Bitcoin network, the protocol should require a proof of finality. Instead of accepting a header in isolation, the SHP verification should require proof of multiple subsequent L1 blocks having been mined on top of it. These subsequent blocks should also have their difficulty targets verified to ensure that they belong to the canonical chain. This makes short-range fork attacks economically infeasible.

Resolution

The development team has acknowledged this finding with the following comment:

> *"Verifying the last L1 hash is sufficient to make sure the LCP and L2 state is on the correct fork. Since Bitcoin block headers include the previous L1 hashes, you cannot supply a single SHP at the end and go to a valid chain.*

If at any moment the L1 chain on the Bitcoin Light Client contract splits from the main chain with the proposed attack, then that chain cannot merge into the main chain without finding a hash collision."

CTR-11	Missing Finality Proof Allows Prover To Stall L2 Chain For Light Clients		
Asset	`crates/bitcoin-da/verifier.rs, crates/light-client-prover/src/circuit/mod.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

A malicious light client prover can craft a light client proof that points to a minority fork on Bitcoin. This would effectively halt the L2 chain from progressing for any light client that accepts the fraudulent proof.

In `run_circuit()`, `input.da_block_header` corresponds to the next finalised Bitcoin block that contains batch proof transactions to verify and prove over. However, there is no logic to verify that the block has actually been finalised.

crates/light-client-prover/src/circuit/mod.rs::run_circuit()

```
// @audit `new_da_state` doesn't need to be a finalised block
let new_da_state = da_verifier
    .verify_header_chain(
        previous_light_client_proof_output
            .as_ref()
            .map(|output| &output.latest_da_state),
        &input.da_block_header,
        network,
    )
    .map_err(|err| LightClientVerificationError::HeaderChainVerificationFailed(err))?;
```

This missing check allows a malicious light client prover to run the proving circuit on a minority fork, permanently halting light client proofs (LCPs).

The attack unfolds as follows:

1. The malicious light client prover takes a real, recent Bitcoin block that contains Citrea batch transactions.
2. The prover mines another Bitcoin block at the same height. This mined block is a legacy (non-segwit) block with zero transactions, constructed to bypass blob parsing and commitment and witness root checks in `verify_transactions()`.
3. The prover provides the following as inputs to the guest program:
   a. The `da_block_header` as the mined legacy block.
   b. An inclusion proof with no `wtxids` and an empty completeness proof.
4. The prover then runs the proving circuit and outputs a `LightClientCircuitOutput` with the same state as the previous proof, except with a changed `latest_da_state`.

Since the `latest_da_state` has been updated to point to a Bitcoin minority fork that has been mined by the malicious prover, batch proofs on the canonical Bitcoin chain will no longer be processed.

This issue has a high impact as light clients would no longer be able to verify the state of the L2. This would prevent withdrawals from being processed through Citrea's bridge. There is no way to point LCPs back to the canonical Bitcoin chain as canonical Bitcoin blocks would not build off of the minority fork, and hence, `verify_header_chain()` will fail as `prev_hash` will not match. The issue has a low likelihood as it requires the attacker to be able to mine their own valid Bitcoin block that is able to satisfy the correct difficulty target.

Recommendations

Ensure that the provided `da_block_header` belongs to a finalised Bitcoin block by also requiring a finality proof in `run_circuit()`. Instead of accepting a header immediately, the L2 should require proof of multiple subsequent L1 blocks having been mined on top of it. These subsequent blocks should also have their difficulty targets verified to ensure that they belong to the canonical chain.

Resolution

The development team has acknowledged this finding with the following comment:

> *"This is intended behavior. Citrea LCPs themselves do not implement the longest chain rule. Clementine bridge proofs which encapsulate Citrea LCPs will implement this rule and ignore LCPs for minority forks."*

CTR-12	Incorrect Handling Of KZG Point Evaluation Precompile (0x0A) In Cancun Fork		
Asset	`crates/evm/src/evm/handler.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The Cancun hardfork implementation in Citrea incorrectly omits the KZG point evaluation precompile at address `0x0A`. This results in calls to this precompile being treated as calls to an empty address, which silently succeed without returning data, instead of reverting as expected.

In `handler.rs`, the `cancun()` function, which is responsible for providing the precompiles for the Cancun specification, simply returns the set of precompiles from the Berlin hardfork.

crates/evm/src/evm/handler.rs::cancun()

```
// Returns precompiles for Citrea's Cancun spec.
pub fn cancun() -> &'static Precompiles {
    static INSTANCE: OnceBox = OnceBox::new();
    INSTANCE.get_or_init(|| {
        // Berlin because POINT_EVALUATION precompile(0x0A) is enabled in Cancun
        Box::new(Precompiles::berlin().clone())
    })
}
```

The point evaluation precompile (`0x0A`) has been omitted meaning that calls to its address are handled as calls to a normal address. This is different to how `revm` handles the disabled precompile, as it registers a precompile at `0x0A` that returns a `PrecompileError::Fatal` if the `c-kzg` feature is disabled. `revm`'s implementation is shown below:

revm-precompile-18.0.0/src/lib.rs::cancun()

```
/// Returns precompiles for Cancun spec.
///
/// If the `c-kzg` feature is not enabled KZG Point Evaluation precompile will not be included,
/// effectively making this the same as Berlin.
pub fn cancun() -> &'static Self {
    static INSTANCE: OnceBox<Precompiles> = OnceBox::new();
    INSTANCE.get_or_init(|| {
        let mut precompiles = Self::berlin().clone();

        // EIP-4844: Shard Blob Transactions
        cfg_if! {
            if #[cfg(any(feature = "c-kzg", feature = "kzg-rs"))] {
                let precompile = kzg_point_evaluation::POINT_EVALUATION.clone();
            } else {
                let precompile = PrecompileWithAddress(u64_to_address(0x0A), |_,_| Err(PrecompileError::Fatal("c-kzg feature is not
    ↪  enabled".into())));
            }
        }
        // ...
    })
}
```

Citrea's current implementation deviates from this, creating a potential pitfall for developers deploying Cancun-compatible contracts.

The impact of this issue is considered medium because it will cause contracts that call the KZG point evaluation precompile to behave incorrectly. The likelihood is assessed as low because it is unlikely that contracts on Citrea will call this precompile.

Recommendations

Modify the `cancun()` function in `handler.rs` to correctly handle the `0x0A` precompile. Even if the point evaluation functionality is not implemented, a precompile should be registered at the address `0x0A` that explicitly returns an error upon being called. This ensures that the behavior is consistent with `revm` and that contracts calling this precompile will fail as expected.

Furthermore, it is recommended to add documentation to inform smart contract developers about the lack of support for this precompile.

Resolution

The development team has acknowledged this issue with the following comments:

> *"Changing the behavior of this precompile would require a messy reorg or hard fork. We will document this behavior in our docs."*

CTR-13	Potential Over-Minting of `cBTC` Due To Missing `depositAmount` Check		
Asset	`crates/evm/src/evm/system_contracts/src/Bridge.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `initialize()` function does not ensure that the `_depositAmount` parameter is a multiple of 10^{10}, which represents one Satoshi in the `cBTC` denomination (`cBTC` having 18 decimals). This oversight can lead to a scenario where the bridge mints more `cBTC` to a user than the value of their actual bitcoin deposit, as the signature verification process for Bitcoin transactions effectively truncates the `depositAmount`.

The `Bridge` contract's `initialize()` function is responsible for setting the `depositAmount`, which is a fixed quantity of `cBTC` for deposit and withdrawal operations. This `depositAmount` is utilised within the `verifySigInTx()` function to derive `shaAmounts`, a component of the message hash for Bitcoin transaction signature verification:

crates/evm/src/evm/system_contracts/src/Bridge.sol::verifySigInTx()

```
bytes32 shaAmounts = sha256(abi.encodePacked(bytes8(BTCUtils.reverseUint64(uint64(depositAmount/(10**10))))); // 1000000000 in LE
```

In this calculation, `depositAmount` is divided by 10^{10} and then cast to a `uint64`. If `depositAmount` is not a clean multiple of 10^{10}, this division will truncate the value. For example, if `depositAmount` were set to 15×10^9 (representing 1.5 Satoshis worth of `cBTC`), the expression would evaluate to 1. The cryptographic verification would thus proceed as if the deposit corresponds to 1×10^{10} wei of `cBTC`.

However, during the processing of a deposit in the `deposit()` function, the full, potentially non-multiple, `depositAmount` is transferred to the recipient:

crates/evm/src/evm/system_contracts/src/Bridge.sol::deposit()

```
(bool success, ) = recipient.call{value: depositAmount}("");
```

Following the example, the user would receive 15×10^9 `cBTC`, while the bridge's internal accounting for the Bitcoin side (via `shaAmounts`) would reflect only 10×10^9 `cBTC`. This results in an over-minting of 5×10^9 `cBTC` that is not backed by an equivalent amount of bitcoin.

The impact is medium as this vulnerability could lead to depositors receiving slightly more `cBTC` than intended. The likelihood is low because the `initialize()` function is `onlySystem`, meaning the vulnerability can only be introduced if the `SYSTEM_CALLER` provides an improperly configured `depositAmount` during the initial setup.

Recommendations

The `initialize()` function should be updated to include a requirement that `depositAmount` is perfectly divisible by 10^{10}.

Resolution

The development team has fixed this issue and implemented the recommended changes in PR #2456.

CTR-14	Merkle Tree Hash Collision Due To Lack Of Domain Separation		
Asset	`crates/sovereign-sdk/rollup-interface/src/state_machine/transaction.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

A hash collision vulnerability exists in the transaction Merkle tree construction, allowing a transaction to have the same hash as an internal node in the tree. This is due to the lack of domain separation between leaf nodes (transactions) and internal nodes.

The hash of a transaction is calculated as `hash(runtime_msg | chain_id | nonce)`. The `runtime_msg` is a variable-length byte array, while `chain_id` and `nonce` are both 8 bytes. An internal node in the Merkle tree is calculated as `hash(left_child_hash | right_child_hash)`, where each child hash is 32 bytes.

An attacker can exploit this by crafting a transaction where the `runtime_msg` is 48 bytes long. This makes the total preimage for the transaction hash `48 + 8 + 8 = 64` bytes, which is the same length as the preimage for an internal node hash (`32 + 32 = 64` bytes).

If an attacker can find two sibling transaction hashes in the tree, `h_L` and `h_R`, and craft a `runtime_msg` such that `runtime_msg (48 bytes) | chain_id (8 bytes) | nonce (8 bytes) == h_L (32 bytes) | h_R (32 bytes)`, then the hash of this malicious transaction will be identical to the hash of the internal node `hash(h_L | h_R)`. This would allow the attacker to create a valid Merkle proof for a transaction that was never actually included in the list of transactions, or create another block with malformed transactions that would have the same Merkle root as the original block.

The `verify_tx_merkle_root()` function calculates transaction hashes and builds the Merkle tree:

crates/sovereign-sdk/module-system/sov-modules-stf-blueprint/src/lib.rs::verify_tx_merkle_root()

```
  fn verify_tx_merkle_root<C: Context + Spec>(
    l2_block: &L2Block,
) -> Result<(), StateTransitionError> {
    let tx_hashes: Vec<[u8; 32]> = l2_block
        .txs
        .iter()
        .map(|tx| tx.compute_digest::<<C as Spec>::Hasher>().into())
        .collect();

    let tx_merkle_root = if tx_hashes.is_empty() {
        EMPTY_TX_ROOT
    } else {
        MerkleTree::<Sha256>::from_leaves(&tx_hashes)
            .root()
            .expect("Couldn't compute merkle root")
    };
    // ...
}
```

The transaction hash is computed in `compute_digest()`:

crates/sovereign-sdk/rollup-interface/src/state_machine/transaction.rs::compute_digest()

```
pub fn compute_digest(&self) -> digest::Output {
    let mut hasher = D::new();
    hasher.update(self.runtime_msg());
    hasher.update(self.chain_id().to_be_bytes());
    hasher.update(self.nonce().to_be_bytes());
    hasher.finalize()
}
```

The `from_leaves()` function does not apply any domain separation, hashing the supplied leaf data directly. This allows for the collision between a transaction hash and an internal node hash.

The vulnerability is rated as medium impact because it allows for multiple block encodings to result in the same block hash, although the alternative blocks will likely be rejected by the state transition function. Depending on how full nodes process rejected blocks, this could potentially lead to chain halts or splits. The likelihood is rated as low because the attacker needs to mine a preimage to ensure that the `chain_id` and `nonce` of the resultant hash is equivalent to the preimage.

Recommendations

To mitigate this vulnerability, domain separation should be introduced between leaf and internal nodes in the Merkle tree. The recommended approach is to prefix the data with a domain separator before hashing.

1. For leaf nodes, the hash should be calculated as `hash(0x00 | transaction_hash)`.
2. For internal nodes, the hash should be calculated as `hash(0x01 | left_child_hash | right_child_hash)`.

Resolution

The development team has fixed this issue and implemented the recommended changes in PR #2491.

CTR-15	Incorrect Check For Schnorr Signature Validity		
Asset	`crates/evm/src/evm/system_contracts/src/Bridge.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `isSchnorrSigValid()` function incorrectly checks `result.length != 0` to determine whether a signature is valid. This check is not specific enough and could potentially be bypassed, leading to unauthorised `deposit()` and `replaceDeposit()` call that put user funds at risk.

The `isSchnorrSigValid()` function in the `Bridge` contract is responsible for verifying Schnorr signatures by calling a precompiled contract. The current implementation is as follows:

crates/evm/src/evm/system_contracts/src/Bridge.sol::isSchnorrSigValid()

```
/// @notice Checks if a Schnorr signature is valid by calling Citrea's Schnorr signature verification precompile at 0x200
function isSchnorrSigValid(bytes memory pubKey, bytes32 messageHash, bytes memory signature) internal view returns (bool isValid) {
    require(signature.length == 64 || signature.length == 65, "Invalid signature length");
    signature = signature.slice(0, 64);
    (, bytes memory result) = address(SCHNORR_VERIFIER_PRECOMPILE).staticcall(abi.encodePacked(pubKey, messageHash, signature));
    isValid = result.length != 0;
}
```

This implementation has two main weaknesses:

1. It ignores the `success` boolean returned by `staticcall`. If the precompile call reverts (e.g., due to out-of-gas), `success` will be `false` and `result` will be empty. The function will correctly return `false` in this case, but it is better to explicitly handle the call failure.
2. It only checks that the returned `result` is not empty. According to the Citrea implementation of the Schnorr precompile, it should return `0x00...01` (a 32-byte value) for a valid signature and empty bytes for an invalid one, as seen in `crates/evm/src/evm/precompiles/schnorr.rs`:

   ```
   // ...
       let result = verify_sig(input).map_or_else(Bytes::new, |_| B256::with_last_byte(1).into());
   // ...
   ```

   The current check would incorrectly accept any non-empty return value as valid. For example, if a bug in the precompile caused it to return `0x02`, the signature would be considered valid.

This issue is currently not exploitable, as the Schnorr precompile correctly returns either an empty byte array for invalid signatures or a fixed value (`0x00...01`) for valid signatures. The check `result.length != 0` is therefore currently sufficient. However, this is a weak check that relies on the precompile's specific implementation. Should the precompile be updated in the future to return different values, this check might become insufficient, potentially leading to forged signatures being accepted. Because there is no immediate threat and the vulnerability is contingent on future changes or a bug in a different component, the impact and likelihood are rated as low.

This issue has also been discovered by the Citrea team during the duration of the engagement.

Recommendations

Refactor the `isSchnorrSigValid()` function to perform a stricter check. The function should:

1. Check the `success` flag of the `staticcall`.
2. Verify that the `result` has a length of 32 bytes.
3. Compare the `result` with the expected value of `0x00...01`.

Resolution

The `isSchnorrSigValid()` function has been refactored to implement the checks recommended above.

The development team has fixed this issue in PR #2380.

CTR-16	Missing Equality Check In `apply_timestamp_rule()`
Asset	`crates/l2-block-rule-enforcer/src/hooks.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `apply_timestamp_rule()` function does not adequately ensure that the current block's timestamp is greater than the previous block's timestamp. It does not check for equality, allowing the current block to have the same timestamp as the previous block:

crates/l2-block-rule-enforcer/src/hooks.rs::apply_timestamp_rule()

```
/// Checks that the current block's timestamp.
/// This is to make sure that the set timestamp is greater than the last block's timestamp.
#[cfg_attr(feature = "native", instrument(level = "trace", skip_all, err, ret))]
fn apply_timestamp_rule(
    &self,
    l2_block: &HookL2BlockInfo,
    last_timestamp: &mut u64,
) -> Result<(), L2BlockHookError> {
    let current_timestamp = l2_block.timestamp();

    // @audit should be <= instead of <
    if current_timestamp < *last_timestamp {
        return Err(L2BlockHookError::TimestampShouldBeGreater);
    }

    *last_timestamp = current_timestamp;

    Ok(())
}
```

Recommendations

Consider adding the equality check in the `if` condition.

Resolution

The development team has acknowledged this issue with the following comment:

> *"We have a bunch of consecutive blocks with the same timestamp. This is due to how we handled `setBlockInfo` events before. Hence, we won't fix this issue for now."*

CTR-17	Handle Unsupported EIP-4844 Transactions Gracefully During Execution
Asset	`crates/evm/src/evm/executor.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

In the `execute_multiple_tx()` function, if a single transaction is of type EIP-4844, the entire block execution fails:

crates/evm/src/evm/executor.rs::execute_multiple_tx()

```
// if tx is eip4844 error out
if tx.is_eip4844() {
    native_error!("EIP-4844 transaction is not supported");
    return Err(L2BlockModuleCallError::EvmTxTypeNotSupported(
        "EIP-4844".to_string(),
    ));
}
```

Although the sequencer filters out EIP-4844 transactions before adding them to the mempool, relying on this check during block execution is not a best practice. It introduces a potential DoS vector, for example, if the sequencer code is updated or if forced transactions are introduced, a single unsupported transaction could halt block production.

Recommendations

Instead of halting the entire block production, ignore transactions if they are of type EIP-4844.

Resolution

The development team has acknowledged this issue with the following comment:

> *"We don't think it's necessary to ignore EIP-4844 transactions. If forced transactions are introduced, we can still filter out EIP-4844 transactions and not include them in a block. If we really need to allow them when we implement forced transactions, the relaxing of that rule would be backwards compatible."*

CTR-18	Redundant Sequencer Public Key Check In `begin_l2_block()`
Asset	`crates/sovereign-sdk/module-system/sov-modules-stf-blueprint/src/lib.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

In `begin_l2_block()`, the `sequencer_public_key` check is unnecessary, as the actual verification of whether the block originates from the sequencer is also performed in the `verify_l2_block()` function.

crates/sovereign-sdk/module-system/sov-modules-stf-blueprint/src/lib.rs::begin_l2_block()

```
// check if l2 block is coming from our sequencer
if l2_block_info.sequencer_pub_key() != sequencer_public_key {
    return Err(StateTransitionError::L2BlockError(
        L2BlockError::SequencerPublicKeyMismatch,
    ));
};
```

In the `apply_l2_block()` function, the `sequencer_public_key` passed to `self.begin_l2_block()` is exactly the same key used to construct `l2_block_info`.

crates/sovereign-sdk/module-system/sov-modules-stf-blueprint/src/lib.rs::apply_l2_block()

```
pub fn apply_l2_block(
    &mut self,
    current_spec: SpecId,
    sequencer_public_key: &K256PublicKey,
    pre_state_root: &StorageRootHash,
    pre_state: C::Storage,
    cumulative_state_log: Option,
    cumulative_offchain_log: Option,
    state_witness: Witness,
    offchain_witness: Witness,
    l2_block: &L2Block,
) -> Result, StateTransitionError> {
    let l2_block_info = HookL2BlockInfo::new(
        l2_block,
        *pre_state_root,
        current_spec,
        sequencer_public_key.clone(), // @audit the same sequencer public key is passed to the constructor
    );

    // ...
    native_debug!("Applying l2 block in STF Blueprint");

    // @audit verify_l2_block() verifies the block has been signed by the sequencer_public_key
    self.verify_l2_block(l2_block, sequencer_public_key)?;

    self.begin_l2_block(sequencer_public_key, &mut working_set, &l2_block_info)?;

    // ...
}
```

Since the same `sequencer_public_key` is being passed and used in both cases, the check within `begin_l2_block()` is effectively comparing the key to itself and is therefore redundant. This also holds true on the sequencer side, where

`begin_l2_block()` is called during block production.

Recommendations

Remove the redundant `sequencer_public_key` check in `begin_l2_block()`.

Resolution

The development team has fixed this issue and implemented the recommended changes in PR #2466.

CTR-19	Missing Check for Empty `batch_proof_method_ids`
Asset	`crates/light-client-prover/src/circuit/mod.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The function `process_complete_proof()` attempts to access the contents of `batch_proof_method_ids` without checking whether the vector is empty:

crates/light-client-prover/src/circuit/mod.rs::process_complete_proof()

```
let batch_proof_method_id = if batch_proof_method_ids.len() == 1 {
    batch_proof_method_ids[0].1
} else {
    let idx = match batch_proof_method_ids
        .binary_search_by_key(&batch_proof_output_last_l2_height, |(height, _)| *height)
    {
        Ok(idx) => idx,
        Err(idx) => idx.saturating_sub(1),
    };
    batch_proof_method_ids[idx].1  // @audit Potential panic if the list is empty
};
```

If `batch_proof_method_ids` is empty, accessing `batch_proof_method_ids[0]` will cause a panic.

Although it is possible to have an empty `batch_proof_method_ids` vector, this does not occur in practice as the vector is initialised with pre-defined values from `initial_values.rs`. Hence, this issue is rated as informational.

Recommendations

Consider adding a sanity check to prevent the non-emptiness of the vector.

Resolution

`initial_batch_proof_method_ids()` now returns a `NonEmptySlice` struct that enforces the non-emptiness of the slice in its constructor. As such, it's no longer possible to mistakenly initialise batch proof method IDs with an empty vector.

The development team has fixed this issue in PR #2527.

CTR-20	Use Of `wrapping_sub` Allows Silent Overflow In `spec_id` Comparison
Asset	`crates/sovereign-sdk/rollup-interface/src/fork/mod.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

In `verify_forks()`, the following check is used to ensure that `spec_ids` increase by exactly 1:

```
if (fork.spec_id as u8).wrapping_sub(forks[i - 1].spec_id as u8) != 1
```

This computation relies on `wrapping_sub()`, which performs subtraction with wrapping which can overflow silently. This allows values like `0.wrapping_sub(255)` to evaluate to 1, even though `spec_id` 0 and 255 are not consecutive.

This issue has been rated as informational because `spec_id` is an `enum` and it is unlikely there will be 256 variants of it.

Recommendations

Consider replacing `wrapping_sub()` with a saturating subtraction. Any overflowing subtractions would instead evaluate to 0 which would fail the check.

Resolution

The `wrapping_sub()` has been replaced with a saturating subtraction as recommended above.

The development team has fixed this issue in PR #2467.

CTR-21	Inconsistent Endianness In Transaction Serialisation
Asset	`crates/sovereign-sdk/rollup-interface/src/state_machine/transaction.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `chain_id` and `nonce` fields are serialised with inconsistent endianness in different parts of the transaction lifecycle. For signature generation and verification, they are encoded as little-endian, while for transaction digest computation, they are encoded as big-endian.

In `TransactionV1::new()` and `TransactionV1::verify()`, the `chain_id` and `nonce` are appended to the message buffer using little-endian encoding. The signature is created over the little-endian representation, and verification correctly uses the same format.

crates/sovereign-sdk/rollup-interface/src/state_machine/transaction.rs::TransactionV1

```
fn new(priv_key: &K256PrivateKey, runtime_msg: Vec<u8>, chain_id: u64, nonce: u64) -> Self {
    // ...
    message.extend_from_slice(&chain_id.to_le_bytes());
    message.extend_from_slice(&nonce.to_le_bytes());
    // ...
}

fn verify(&self) -> anyhow::Result<()> {
    // ...
    serialized_tx.extend_from_slice(&self.chain_id.to_le_bytes());
    serialized_tx.extend_from_slice(&self.nonce.to_le_bytes());
    // ...
}
```

However, when computing the transaction digest in `Transaction::compute_digest()`, these same fields are encoded using big-endian.

crates/sovereign-sdk/rollup-interface/src/state_machine/transaction.rs::compute_digest()

```
pub fn compute_digest(&self) -> digest::Output {
    // ...
    hasher.update(self.chain_id().to_be_bytes());
    hasher.update(self.nonce().to_be_bytes());
    hasher.finalize()
}
```

This inconsistency introduces ambiguity in the serialisation logic, which may result in subtle and difficult to debug errors if developers rely on incorrect assumptions about the data format.

Recommendations

Enforce a consistent serialisation format for transaction data across the entire codebase by changing the `chain_id` and `nonce` serialisation to big-endian.

Resolution

A new `TransactionV2` struct has been created that uses big-endian encoding for the `chain_id` and `nonce` fields.

The development team has fixed this issue in PR #2484.

CTR-22	Incorrect L2 Height Used For Cache Pruning Check
Asset	`crates/sovereign-sdk/module-system/sov-modules-stf-blueprint/src/lib.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `cache_prune_l2_heights_iter` is checked against the incorrect `l2_height`, resulting in the state and offchain logs from being pruned on the incorrect L2 block heights.

The `l2_height` variable is incremented before being used in the cache pruning check. This causes the check against `l2_height` to effectively compare against `l2_height + 1` of the current block, instead of the actual current `l2_height`.

The relevant code is in the `apply_l2_blocks_from_sequencer_commitments` function within `lib.rs`:

sov-modules-stf-blueprint/src/lib.rs::apply_l2_blocks_from_sequencer_commitments()

```
l2_height += 1;

prev_l2_block_hash = Some(l2_block.hash());

l2_block_hashes.push(l2_block.hash());

let mut state_log = result.state_log;
let mut offchain_log = result.offchain_log;
// prune cache logs if it is hinted from native
if cache_prune_l2_heights_iter
    .next_if_eq(&&l2_height) // @audit This should use the l2_height *before* incrementing
    .is_some()
{
    state_log.prune_half();
    offchain_log.prune_half();
}
```

This issue was identified by the Citrea team during the engagement. It's classified as informational because it doesn't directly lead to loss of funds or break core functionality, but rather represents a logical error in cache management that might lead to slightly suboptimal cache pruning decisions.

Recommendations

Modify implementation such that the `l2_height` variable is used in the `next_if_eq()` check *before* it is incremented.

Resolution

The `l2_height` variable is now incremented after cache logs are pruned.

The development team has fixed this issue in PR #2390.

CTR-23	Inconsistent Endianness In `safeWithdraw()` Function
Asset	`crates/evm/src/evm/system_contracts/src/Bridge.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `safeWithdraw()` function incorrectly extracts the spent output when validating the transaction. This is due to the endianness mismatch between the Bitcoin transaction encoding (little-endian) and Solidity's integer interpretation (big-endian).

crates/evm/src/evm/system_contracts/src/Bridge.sol::safeWithdraw()

```
// @audit the `spentIndex` is not converted to big-endian before being passed to `extractOutputAtIndex()`
bytes4 spentIndex = payoutInput.extractTxIndexLE();
bytes memory spentOutput = prepareTx.vout.extractOutputAtIndex(uint32(spentIndex));
```

Since the `spentIndex` is encoded in little-endian, this results in an incorrect spent output being extracted by the `extractOutputAtIndex()` function.

This issue has been classified as informational, given that it was independently discovered by the Citrea team during the engagement.

Its actual impact is low as, although it would lead to a denial of service as the incorrect `spentOutput` would fail the `require()` checks, the user can bypass these checks by calling `withdraw()` instead. The likelihood is high as this denial of service would occur for all correct withdrawals.

Recommendations

Ensure that the transaction index is converted to big-endian before passing it to `extractOutputAtIndex()`.

Resolution

The `spentIndex` is now converted to big-endian before being passed to `extractOutputAtIndex()`.

The development team has fixed this issue in PR #2380.

CTR-24	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Unnecessary `prev_level_index_offset` Variable in Merkle Tree Construction**

   Related Asset(s): crates/bitcoin-da/src/helpers/merkle_tree.rs

   The `prev_level_index_offset` variable is declared and updated inside the `BitcoinMerkleTree::new()` function, but it is always set to 0 and never used meaningfully in any computation.

   crates/bitcoin-da/src/helpers/merkle_tree.rs::new()

   ```
   // @audit prev_level_index_offset is initialised to 0
   let mut prev_level_index_offset = 0;
   // ...
   // Continue building the tree until we reach a level with only one node (the root)
   while prev_level_size > 1 {
       // ...
       // @audit it is set to 0 again even though it was never modified in the loop
       prev_level_index_offset = 0;
   }
   ```

   Consider removing the variable `prev_level_index_offset`.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The development team has fixed the issues above in PR #2495.

Appendix A Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

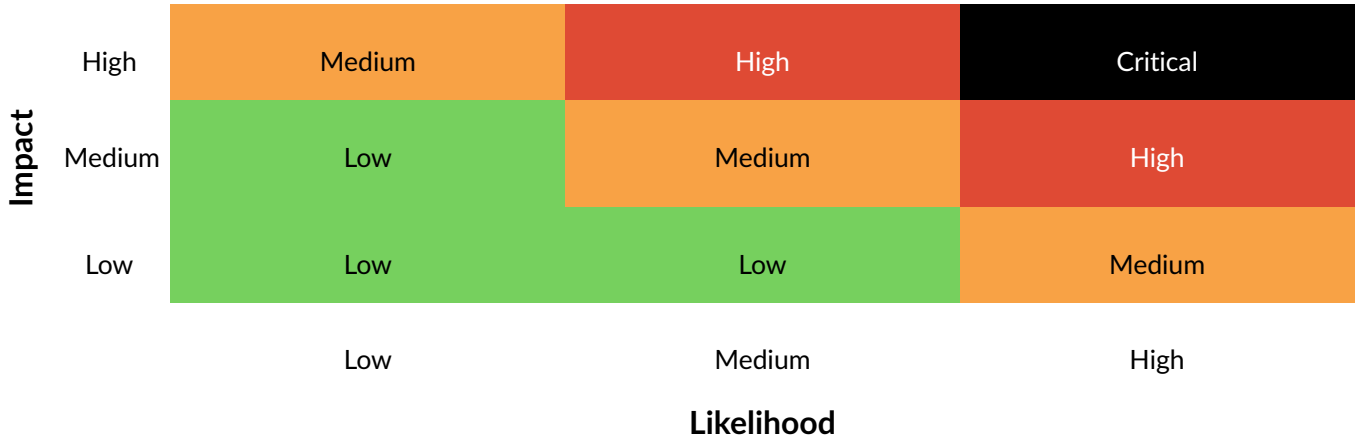

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].