Chainway Labs

Clementine

Security Assessment Report

Version: 2.0

August, 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 these components, 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

Clementine is a trust-minimized, collateral-efficient, and scalable bridge designed to connect the Bitcoin blockchain with other systems, specifically the Citrea zk-rollup.

It enables users to securely deposit and withdraw their funds to and from Citrea and Bitcoin mainnet through Bitcoin multi-sig covenants and an optimistic Bitcoin light client on Bitcoin itself, using an advanced cryptographic construction called BitVM2.

Security Assessment Summary

Scope

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

The scope of this time-boxed review was strictly limited to the following files at commit 937e9f4:

- `core/src/main.rs`
- `core/src/extended_rpc.rs`
- `core/src/deposit.rs`
- `core/src/cli.rs`
- `core/src/citrea.rs`
- `core/src/verifier.rs`, excluding:
  - `is_kickoff_malicious()`
  - `handle_kickoff()`
  - `send_watchtower_challenge()`
  - `queue_watchtower_challenge()`
  - `send_unspent_kickoff_connectors()`
- `core/src/bitvm_client.rs`
- `core/src/actor.rs`
- `core/src/operator.rs`
- `core/src/aggregator.rs`
- `core/src/servers.rs`
- `core/src/musig2.rs`
- `core/src/rpc/*`
- `core/src/config/*`
- `core/src/builder/*`
- `core/src/database/*`

The fixes of the identified issues were assessed at commit aa76265.

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

Approach

The security assessment covered components written in Rust.

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 36 issues during this assessment. Categorised by their severity:

- High: 2 issues.
- Medium: 13 issues.
- Low: 13 issues.
- Informational: 8 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 components(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
CMT-01	Unhandled Panics From Out-Of-Bounds Slices Lead To Denial Of Service	High	Resolved
CMT-02	Arithmetic Overflow Panic In Amount Subtraction Operations	High	Resolved
CMT-03	Unnecessary Work Due To Lack Of UTXO Spendability Checks	Medium	Resolved
CMT-04	Operator Can Avoid Slashing By Breaking Transaction Chain	Medium	Resolved
CMT-05	Unverified Light Client Proof Can Lead To Lost Or Stolen Funds	Medium	Resolved
CMT-06	Collateral Spend In Mempool Results In Denial Of Service	Medium	Resolved
CMT-07	Database Transaction Isolation Vulnerability In Payout Processing	Medium	Resolved
CMT-08	Resource Exhaustion Via Malicious Winternitz Keys In Script Generation	Medium	Resolved
CMT-09	Incorrect Profitability Check May Lead To Operator Fund Loss	Medium	Closed
CMT-10	Unbounded Memory Growth In Verifier's `AllSessions` Leads To Denial Of Service	Medium	Resolved
CMT-11	Unbounded Nonce Generation In `nonce_gen()` Leads To Denial Of Service	Medium	Resolved
CMT-12	Predictable Nonce Session ID Leads To Denial Of Service	Medium	Resolved
CMT-13	Index-Out-Of-Bounds Panic In `parse_deposit_sign_session()` Allows Remote DoS	Medium	Resolved
CMT-14	Nonce Session Hijacking In `sign_optimistic_payout()`	Medium	Closed
CMT-15	Slashed Operator Halts New Deposits, Causing Denial Of Service	Low	Closed
CMT-16	Operator Collateral Validation Accepts Mempool Transactions Leading To Potential Spoofing	Low	Resolved
CMT-17	Missing HTTP Timeouts In Citrea Client	Low	Resolved
CMT-18	Missing gRPC Server Hardening In `create_grpc_server()`	Low	Resolved
CMT-19	Unbounded `max_encoding_message_size()` Allows Excessive gRPC Responses	Low	Resolved
CMT-20	Missing Input Validation For Empty Public Keys In MuSig2 Key Aggregation	Low	Resolved
CMT-21	Empty Nonce Array Panic In MuSig2 Aggregation	Low	Resolved
CMT-22	Inefficient Winternitz Key Validation Order Allows Unnecessary Processing	Low	Resolved
CMT-23	Delayed Signature Verification Can Lead To DoS In `optimistic_payout()`	Low	Resolved

CMT-24	Overly Broad Error Handling in Citrea RPC Calls May Lead to Missed Onchain Data	Low	Resolved
CMT-25	`is_deposit_valid()` Does Not Check For Extraneous Operators In Deposit Data	Low	Resolved
CMT-26	Winternitz Key Derivation Uses Simple Concatenation Instead Of KDF	Low	Resolved
CMT-27	Missing Rate Limiting Protection On gRPC API Endpoints	Low	Resolved
CMT-28	Challenge Fee Recovery Mechanism Not Implemented	Informational	Resolved
CMT-29	Hardcoded Block Limit In `get_logs()` Should Be Configurable	Informational	Closed
CMT-30	Outstanding `TODO` Comments Throughout Codebase	Informational	Closed
CMT-31	`set_operator_keys()` Does Not Validate Deposit Before Database Write	Informational	Resolved
CMT-32	Lack Of Validation For Protocol Parameters	Informational	Closed
CMT-33	Unchecked Arithmetic Operations In Round Transaction Value Calculation	Informational	Resolved
CMT-34	Withdrawal UTXOs Has Incorrect Endianness	Informational	Resolved
CMT-35	Miscellaneous General Comments	Informational	Closed
CMT-36	Direct Panic Vulnerabilities In Verifier RPC Endpoints	Medium	Resolved

CMT-01	Unhandled Panics From Out-Of-Bounds Slices Lead To Denial Of Service		
Asset	core/src/operator.rs, core/src/aggregator.rs, core/src/deposit.rs, core/src/builder/address.rs		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The codebase contains multiple vulnerabilities where user-provided data is sliced without proper validation, leading to panics that can cause a denial of service for operators, verifiers, and other bridge services.

The core issue is the assumption that user-provided Bitcoin addresses or UTXOs are always Pay-to-Taproot (P2TR), which have a `script_pubkey` of at least 34 bytes. The code directly slices the `script_pubkey` to extract a 32-byte key, which will panic if a user provides a different address type (e.g., P2WPKH) with a shorter `script_pubkey`.

This vulnerability pattern appears in multiple different places, affecting various components of the bridge infrastructure.

1. The `generate_deposit_address()` function in `core/src/builder/address.rs` is used to create a new deposit address for a user. It takes a `recovery_taproot_address` from the user, which is intended for fund recovery. The function slices the `script_pubkey` of this address without validation.

   core/src/builder/address.rs::generate_deposit_address()

   ```
   let recovery_script_pubkey = recovery_taproot_address
       .clone()
       .assume_checked()
       .script_pubkey();

   // @audit This will panic if recovery_taproot_address is not a P2TR address
   let recovery_extracted_xonly_pk =
       XOnlyPublicKey::from_slice(&recovery_script_pubkey.as_bytes()[2..34])
           .wrap_err("Failed to extract xonly public key from recovery taproot address")?;
   ```

2. A similar vulnerability exists in `core/src/deposit.rs` within the `get_deposit_scripts()` function. This function is called by verifiers when validating a new deposit. The function assumes this is a P2TR address and attempts to slice its script pubkey to extract the public key.

   core/src/deposit.rs::get_deposit_scripts()

   ```
   let recovery_script_pubkey = original_deposit_data
       .recovery_taproot_address
       .clone()
       .assume_checked()
       .script_pubkey();

   let recovery_extracted_xonly_pk =
       XOnlyPublicKey::from_slice(&recovery_script_pubkey.as_bytes()[2..34])
           .wrap_err(
               "Failed to extract xonly public key from recovery script pubkey",
           )?;
   ```

3. The `withdraw()` function in `core/src/operator.rs` is vulnerable to a panic due to an out-of-bounds slice on a user-provided `script_pubkey`. This can be triggered remotely by any user to crash all operators, causing a network-wide denial of service. The `aggregator.rs::withdraw()` RPC endpoint forwards withdrawal requests to all operators. Each operator's `withdraw()` function processes the request, which includes an outpoint to a

UTXO (`in_outpoint`) that the user provides. The implementation of `operator.rs::withdraw()` assumes that this UTXO has a P2TR `script_pubkey` .

core/src/operator.rs::withdraw()

```
let user_xonly_pk =
    XOnlyPublicKey::from_slice(&input_utxo.txout.script_pubkey.as_bytes()[2..34])
        .wrap_err("Failed to extract xonly public key from input utxo script pubkey")?;
```

In all of the three cases above, a user can provide a non-P2TR address to cause a panic due to an out-of-bounds slice.

The impact is medium because this vulnerability allows an attacker to cause a temporary denial-of-service against operators and verifiers. While the nodes can be restarted, they would likely crash again if they re-process the malicious deposit or withdrawal, effectively halting bridge operations until a fix is deployed. The likelihood is high because these vulnerabilities can be triggered by any user with minimal or no cost. For the `get_deposit_scripts()` case, a user can provide parameters for a deposit transaction that does not exist onchain or the mempool, such that the user does not need to spend any BTC or place any onchain transactions to perform this attack.

Recommendations

Before slicing a `script_pubkey` to extract a public key, validate that it is a P2TR output. The `bitcoin::Script` type provides the `is_p2tr()` method, which checks for the correct length and prefix.

Additionally, ensure that the `recovery_taproot_address` is a Taproot address at the beginning of the deposit/withdrawal flow. This can be done when converting the Protobuf struct to its internal type by adding the check into the `TryFrom` trait implementation for `DepositType` in `core/src/rpc/parser/mod.rs` .

Resolution

The issue has been address in commit db65d75, by adding checks to ensure that the `script_pubkey` is a valid P2TR output before slicing it to extract the public key. Furthermore, checks have been added to ensure block hashes are at least 20 bytes long when extract the last 20 bytes.

CMT-02	Arithmetic Overflow Panic In Amount Subtraction Operations		
Asset	`core/src/operator.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The `is_profitable()` function performs unchecked `Amount` subtraction operations that can cause arithmetic overflow panics, leading to denial-of-service attacks against operator nodes when an operator withdrawal is requested.

The `bitcoin-units` crate's `Amount` type implements arithmetic operations from `core::ops` that panic on overflow/underflow. According to the crate documentation: *"The operations from `core::ops` that `Amount` implements will panic when overflow or underflow occurs. Also note that since the internal representation of amounts is unsigned, subtracting below zero is considered an underflow and will cause a panic."*

The `is_profitable()` function does not use checked subtraction, resulting in a panic if the following subtraction were to underflow:

core/src/operator.rs::is_profitable()

```
/// Checks if the withdrawal amount is within the acceptable range.
fn is_profitable(
    input_amount: Amount,
    withdrawal_amount: Amount,
    bridge_amount_sats: Amount,
    operator_withdrawal_fee_sats: Amount,
) -> bool {
    if withdrawal_amount
        .to_sat()
        .wrapping_sub(input_amount.to_sat()) // @audit prefer checked_sub
        > bridge_amount_sats.to_sat()
    {
        return false;
    }

    // Calculate net profit after the withdrawal.
    // @audit This subtraction will panic on underflow
    let net_profit = bridge_amount_sats - withdrawal_amount;

    // Net profit must be bigger than withdrawal fee.
    net_profit >= operator_withdrawal_fee_sats
}
```

This function is called in the operator payout flow to determine if an operator should process the provided withdrawal request. If a malicious user provides a `withdrawal_amount` that exceeds the `bridge_amount_sats`, this subtraction will panic, crashing the operator node.

Keep in mind that the attacker does not need capital to perform this attack as they do not need to perform a peg-out by burning cBTC. They can request a withdrawal by calling the aggregator with the following parameters:

- A valid and finalized `withdrawal_index` that points to a UTXO with non-zero value. This withdrawal can be one that has already been paid out and does not need to belong to the attacker.
- Any `in_signature`. This parameter is not relevant this early in the `withdraw()` function

- The `in_outpoint` corresponding to the chosen `withdrawal_index`. This UTXO should hold a positive amount of value to pass the `withdrawal_amount - input_amount > bridge_amount_sats` check in `is_profitable()` when `withdrawal_amount > bridge_amount_sats`.
- Any `out_script_pubkey`. This parameter is not relevant this early in the `withdraw()` function.
- An `out_amount` equal to `bridge_amount_sats + amount_in_in_outpoint`.

core/src/operator.rs::withdraw()

```
pub async fn withdraw(
    &self,
    withdrawal_index: u32,
    in_signature: schnorr::Signature,
    in_outpoint: OutPoint,
    out_script_pubkey: ScriptBuf,
    out_amount: Amount,
) -> Result {
    // ...

    // Prepare input and output of the payout transaction.
    // @audit this will succeed as the attacker provided a valid in_outpoint
    let input_prevout = self.rpc.get_txout_from_outpoint(&in_outpoint).await?;

    // ...

    // Check Citrea for the withdrawal state.
    // @audit this will succeed as the attacker provided a valid withdrawal_index, even though it has already been processed
    let withdrawal_utxo = self
        .db
        .get_withdrawal_utxo_from_citrea_withdrawal(None, withdrawal_index)
        .await?;

    match withdrawal_utxo {
        Some(withdrawal_utxo) => {
            // @audit this will pass as the attacker provided the correct in_outpoint for the withdrawal_index
            if withdrawal_utxo != input_utxo.outpoint {
                return Err(eyre::eyre!("Input UTXO does not match withdrawal UTXO from Citrea: Input Outpoint: {0}, Withdrawal
                ↪ Outpoint (from Citrea): {1}", input_utxo.outpoint, withdrawal_utxo).into());
            }
        }
        // ...
    }

    // ...

    // @audit underflow panic happens here
    if !Self::is_profitable(
        input_utxo.txout.value,
        output_txout.value,
        self.config.protocol_paramset().bridge_amount,
        operator_withdrawal_fee_sats,
    ) {
        return Err(eyre::eyre!("Not enough fee for operator").into());
    }

    // ...
}
```

This issue is rated as medium impact as although it allows an attacker to crash all Clementine operators by requesting an operator payout through the aggregator, no funds are lost. Its likelihood is high as the attacker can perform the attack without any capital.

Recommendations

Update the function `is_profitable()` to use checked arithmetic operations. Additionally, use `checked_sub()` rather than `wrapping_sub()` when performing subtractions over `withdrawal_amount` and `input_amount`. Either propagate the error or return false if a negative overflow occurs.

Resolution

The issues has been resolved in commit 3a340d3. Raw mathematical operations have been replaced with checked arithmetic operations in the `is_profitable()` function. Any overflows will return `false`.

CMT-03	Unnecessary Work Due To Lack Of UTXO Spendability Checks		
Asset	`core/src/aggregator.rs`, `core/src/verifier.rs`, `core/src/extended_rpc.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `new_deposit()` and `optimistic_payout()` functions in `core/src/rpc/aggregator.rs` do not adequately verify if their respective input UTXOs are unspent before initiating complex and resource-intensive signing processes. This oversight can lead to significant waste of computational resources, as the resulting transactions will be invalid if the input UTXO is no longer available in the canonical chain's UTXO set.

In the `new_deposit()` flow, before initiating the multi-stage signing ceremony, the aggregator attempts to validate the deposit transaction by fetching its block hash:

core/src/rpc/aggregator.rs::new_deposit()

```
// @audit This check is insufficient. It confirms the deposit transaction was mined,
// @audit but it does not verify that the deposit UTXO is unspent. The signing
// @audit ceremony will proceed even if the UTXO is already spent.
let deposit_blockhash = self
    .rpc
    .get_blockhash_of_tx(&deposit_data.get_deposit_outpoint().txid)
    .await
    .map_to_status()?;

let verifiers_public_keys = deposit_data.get_verifiers();
```

However, this check is insufficient. While it confirms the transaction was mined, it does not verify that the specific deposit output (UTXO) remains unspent on the canonical chain. If the deposit UTXO has already been spent, the aggregator, verifiers, and operators will perform the entire resource-intensive signing process for nothing, culminating in an invalid `MoveToVault` transaction that the Bitcoin network will reject.

A similar issue exists in the `optimistic_payout()` function. The withdrawal outpoint and the `MoveToVault` output are not checked for their spendability before generating a payout transaction.

core/src/rpc/aggregator.rs::optimistic_payout()

```
async fn optimistic_payout(
    &self,
    request: tonic::Request<super::WithdrawParams>,
) -> std::result::Result<tonic::Response<super::RawSignedTx>, tonic::Status> {
    let withdraw_params = request.into_inner();
    let (deposit_id, input_signature, input_outpoint, output_script_pubkey, output_amount) =
        parser::operator::parse_withdrawal_sig_params(withdraw_params.clone()).await?;
    // get which deposit the withdrawal belongs to
    let withdrawal = self
        .db
        .get_move_to_vault_txid_from_citrea_deposit(None, deposit_id)
        .await?;
    if let Some(move_txid) = withdrawal {
        // check if withdrawal utxo is correct
        // @audit withdrawal_utxo is not checked for spendability
        let withdrawal_utxo = self
            .db
            .get_withdrawal_utxo_from_citrea_withdrawal(None, deposit_id)
            .await?
            .ok_or(Status::invalid_argument(format!(
                "Withdrawal utxo not found for deposit id {}",
                deposit_id
            )))?;
        // ...
    }
    // ...
}
```

This can lead to wasted work if the UTXOs are no longer spendable, which can occur in two primary scenarios:

1. The `move_txid` is fetched from the aggregator's database. If the `MoveToVault` UTXO has already been spent (e.g., due to a non-finalised deposit replacement), the database might contain a stale `move_txid`. Any attempt to spend an output from this stale transaction will fail.
2. The user's withdrawal UTXO is spent by a past payout.

In both `new_deposit()` and `optimistic_payout()`, the root cause is the failure to check whether the relevant UTXOs have already been spent.

This issue is classified as medium impact because it does not lead to a direct loss of funds but causes a waste of resources, creating a denial-of-service vector. The likelihood is medium because although it is infinitely repeatable on an unprotected end point, as long as there has been a valid past deposit or optimistic payout, it is not guaranteed to significantly impact the liveness of the system.

Recommendations

Add checks at the beginning of `new_deposit()` and `optimistic_payout()` to verify that the respective input UTXOs are unspent before proceeding with signing. Similar checks should also be added to the verifier's methods to make sure verifiers also check the UTXOs are unspent.

The existing function `is_utxo_spent()` in `extended_rpc.rs`, which correctly uses the `get_tx_out` RPC call, is suitable for this purpose as it confirms the UTXO exists on the canonical chain and is unspent.

Resolution

An additional RPC call to check the spendability of the UTXOs has been added to the `optimistic_payout()` and `sign_optimistic_payout()` functions. Updates have been made to in commit e975eab.

The decision to not add the check to `new_deposit()` was made to allow users to call the endpoint to restore deposit data.

CMT-04	Operator Can Avoid Slashing By Breaking Transaction Chain		
Asset	core/src/builder/transaction/operator_collateral.rs & operator_reimburse.rs		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Operators can avoid slashing penalties by directly spending certain outputs to themselves, effectively breaking the transaction chain that enables the slashing mechanism. It requires malicious operators to forfeit future reimbursements but avoids the intended economic penalties for misbehaviour.

Two specific attack vectors have been identified:

1. **Round Transaction**

   The first output of `TransactionType::Round` transactions is the `BurnConnector` UTXO at lines [**103-111**]. This output is designed to be slashable through various timeout mechanisms (assert timeout, disprove, etc.), but operators can spend this output directly to themselves using the key spending path, preventing any slashing transactions from being executed.

   core/src/builder/transaction/operator_collateral.rs::create_round_txhandler()

   ```
   // The vulnerable BurnConnector output in Round transactions
   builder = builder.add_output(UnspentTxOut::from_scripts(
     input_amount
       - (paramset.kickoff_amount + paramset.default_utxo_amount())
           * (paramset.num_kickoffs_per_round as u64)
       - paramset.anchor_amount(),
     vec![],
     Some(operator_xonly_pk), // @audit Operator can spend this directly
     paramset.network,
   ));
   ```

2. **Ready To Reimburse**

   The first output of `TransactionType::ReadyToReimburse` transactions contains the operator's collateral at `UtxoVout::CollateralInReadyToReimburse`. This collateral is intended to be slashed if a kickoff is not finalised through the `create_kickoff_not_finalized_txhandler()` function. However, operators can front-run the slashing transaction by spending this collateral output directly to themselves before the kickoff finalisation period expires.

   core/src/builder/transaction/operator_collateral.rs::create_ready_to_reimburse_txhandler()

   ```
   // The vulnerable collateral in ReadyToReimburse transactions
   .add_output(UnspentTxOut::from_scripts(
       prev_value - paramset.anchor_amount(),
       vec![],
       Some(operator_xonly_pk), // @audit Operator can spend this directly
       paramset.network,
   ))
   ```

While this breaks the chain and forfeits any existing or future reimbursements, it allows the operator to escape slashing penalties, undermining the economic security model of the protocol.

Recommendations

Add further protection to prevent operators entirely avoiding slashing. Such prevention may include having additional stake on L2 which is slashed when an operator misbehaves.

Resolution

The issues have been resolved in commit 2968856. Operators are required to deposit 1 BTC into a smart contract on Citrea, which will allow slashing if the operator misbehaves on the Bitcoin layer.

Therefore, the slashing mechanism are not strictly necessary on the Bitcoin layer. The ability to consume the `BurnConnector` and `CollateralInReadyToReimburse`, by the protocol if the operator misbehaves, is still necessary to prevent the operator from progressing to the next round and claiming reimbursements for fraudulent kickoffs.

CMT-05	Unverified Light Client Proof Can Lead To Lost Or Stolen Funds		
Asset	core/src/citrea.rs, core/src/verifier.rs, core/src/operator.rs		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The system accepts light client proofs (LCPs) from the light client prover service without cryptographically verifying them. A malicious or compromised prover could provide a fraudulent proof, causing the bridge to process transactions from an incorrect or non-finalised L2 state. This could lead to a permanent denial-of-service or the loss of user funds.

The `get_light_client_proof()` function in `core/src/citrea.rs` is responsible for fetching proofs from the prover. While it deserialises the returned data into a `risc0_zkvm::Receipt`, it never calls the `verify()` method to verify the validity of the proof.

core/src/citrea.rs::get_light_client_proof()

```
let ret = if let Some(proof_result) = proof_result {
    let decoded: InnerReceipt = bincode::deserialize(&proof_result.proof)
        .wrap_err("Failed to deserialize light client proof from citrea lcp")?;
    let receipt = receipt_from_inner(decoded)
        .wrap_err("Failed to create receipt from light client proof")?;
    // @audit The proof and its outputs are never verified
    let l2_height = u64::try_from(proof_result.light_client_proof_output.last_l2_height)
        .wrap_err("Failed to convert l2 height to u64")?;
    // ...
}
```

This unverified proof is used by the verifier to determine the finalised L2 block range for updating its database of deposits and withdrawals. A malicious prover can provide false L2 block ranges for blocks that aren't finalised, resulting in the verifier processing deposits and withdrawals on Citrea that may get reorged out of the chain.

The impact of this vulnerability is high, as it undermines the core security of the bridge by processing non-finalised deposits and withdrawals, which could lead to theft or loss of funds if these are not later finalised. The likelihood is considered low because it requires the compromise of the light client prover, which is assumed to be a trusted entity.

Recommendations

The light client proof and its receipt must be cryptographically verified before being used.

The `get_light_client_proof()` function in `core/src/citrea.rs` should verify the receipt immediately after deserialisation and before it is passed to any other component. This ensures that modules like the `Verifier` do not act on unverified data.

Resolution

The issue has been resolved in PRs #971 and #980. The `get_light_client_proof()` function now verifies the receipt immediately and performs the required validation of the receipt journal.

CMT-06	Collateral Spend In Mempool Results In Denial Of Service
Asset	`core/src/extended_rpc.rs`
Status	**Resolved:** See Resolution
Rating	Severity: Medium / Impact: Medium / Likelihood: Medium

Description

The `collateral_check()` function incorrectly fails if an operator's collateral UTXO is spent by a transaction that is still in the mempool. This can cause legitimate deposit requests to be rejected, leading to a temporary denial-of-service during an operator's normal operations.

The `collateral_check()` function in `extended_rpc.rs` is invoked during the deposit process to ensure an operator has sufficient collateral and is still participating in the protocol. The function correctly identifies an operator's latest onchain collateral UTXO by traversing the series of `Round` and `ReadyToReimburse` transactions. However, the final validation step checks if this UTXO has been spent by looking at the mempool in addition to the blockchain.

core/src/extended_rpc.rs::collateral_check()

```
// if the collateral utxo we found latest in the round tx chain is spent, operators collateral is spent from Clementine
// bridge protocol, thus it is unusable and operator cannot fulfill withdrawals anymore
// if not spent, it should exist in chain, which is checked below
Ok(self
    .client
    .get_tx_out(
        &current_collateral_outpoint.txid,
        current_collateral_outpoint.vout,
        Some(true), // include mempool too
    )
    .await
    .wrap_err("Failed to get transaction output")?
    .is_some())
```

When an operator broadcasts a transaction to advance to the next round (e.g., a `Round` or `ReadyToReimburse` transaction), this new transaction spends the current collateral UTXO. While this transaction is in the mempool, the `get_tx_out()` call with `include_mempool` set to `true` will return `None`. This causes `collateral_check()` to return `false`, and as a result, any deposit attempt that includes this operator will be rejected until their transaction is confirmed. A malicious operator could exploit this by broadcasting a transaction with a very low fee, effectively blocking all new deposits for themselves indefinitely.

The impact is classified as medium because it can lead to a denial of service for the deposit functionality, which is a core feature of the protocol. It does not, however, lead to a direct loss of funds. The likelihood is medium as this issue can be triggered unintentionally by any operator during normal operations, or intentionally by a malicious operator with minimal effort.

Recommendations

The `collateral_check()` function should only consider onchain data to determine the validity of a collateral UTXO. By changing the `include_mempool` parameter in the `get_tx_out()` call to `false`, the check will ignore pending mempool transactions.

Resolution

The issue has been resolved in commit 633646c4. Further RPC calls are made to ensure a transaction has be mined in a block before it is considered valid. Note that these checks have only been added to the Bitcoin mainnet, excluding testnets.

CMT-07	Database Transaction Isolation Vulnerability In Payout Processing		
Asset	`core/src/operator.rs, core/src/builder/transaction/sign.rs, core/src/task/payout_checker.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

A database transaction isolation vulnerability exists in the payout processing system where `create_and_sign_txs()` reads data outside the main transaction context, leading to potential data inconsistency and transaction signing failures.

The `handle_finalized_payout()` function maintains a database transaction (`dbtx`) for atomicity, but calls `create_and_sign_txs()` with a new database connection at lines [872-879], breaking transaction isolation.

core/src/operator.rs::handle_finalized_payout()

```
#[cfg(feature = "automation")]
if current_round_index != round_idx {
    // ...
    // start the next round to be able to get reimbursement for the payout
    self.end_round(dbtx).await?; // @audit end round changes occur in dbtx
}

// ...

let signed_txs = create_and_sign_txs(
    self.db.clone(), // @audit fresh db instance does not include changes to dbtx
    &self.signer,
    self.config.clone(),
    transaction_data,
    Some(payout_tx_blockhash),
)
.await?;
```

The issue is that any transaction writes occurring in `dbtx` will not be reflected in `create_and_sign_txs()` . Furthermore, it is possible that `end_round()` may be called before `create_and_sign_txs()` in certain automation cases. However, at this time missed writes from `end_round()` have not been found conflict against queries in `create_and_sign_txs()` .

The issue is rated as medium severity due to the significant potential for inconsistent database operations. The likelihood is rated as a medium as it requires automations to be on and for `end_round()` to be triggered.

Recommendations

Pass transaction context to `create_and_sign_txs()` and perform all database reads in a single transaction.

Resolution

The issue has been resolved in commit 771a811c. The database transaction context is now passed to `create_and_sign_txs()` to ensure any `ReimburseDbCache` reads are made within this transaction context. `get_deposit_signatures()` and other similar calls do not require transaction context and are therefore made outside the transaction.

CMT-08	Resource Exhaustion Via Malicious Winternitz Keys In Script Generation		
Asset	`core/src/verifier.rs`, `core/src/rpc/parser/mod.rs`, `core/src/bitvm_client.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `set_operator_keys()` function allows aggregators to submit Winternitz public keys with unlimited digit counts, causing memory and CPU exhaustion during script generation processes.

The vulnerability occurs when an aggregator submits `OperatorKeys` containing `WinternitzPubkey` entries with excessively large `digit_pubkey` arrays. While the total count of Winternitz keys is validated to be exactly 381, there are no limits on the number of 20-byte entries within each individual key. This allows an attacker to submit keys containing millions of digit entries, leading to resource exhaustion attacks.

The attack begins in `set_operator_keys()` where oversized keys are processed without size validation:

core/src/verifier.rs::set_operator_keys()

```
// core/src/verifier.rs
let winternitz_keys: Vec = keys
    .winternitz_pubkeys
    .into_iter()
    .map(|x| x.try_into()) // @audit No limits on individual key sizes
    .collect::>()?;
```

The `TryFrom` implementation only validates individual byte lengths but allows unlimited array sizes:

core/src/rpc/parser/mod.rs::TryFrom<WinternitzPubkey>

```
inner
    .into_iter()
    .enumerate()
    .map(|(i, inner_vec)| {
        inner_vec
            .try_into()
            .map_err(|e: Vec<_>| eyre::eyre!("Incorrect length {:?}, expected 20", e.len()))
            // @audit No validation on total count of digit_pubkey entries
    })
    .collect::, eyre::Report>>()
```

These oversized keys cause resource exhaustion in multiple ways. First, they consume excessive memory during processing - an attacker could submit 381 keys with 1 million digits each, requiring approximately 7.6 GB of memory allocation. Second, they cause CPU exhaustion during the script replacement process in `replace_disprove_scripts()` where millions of byte replacement operations occur:

core/src/bitvm_client.rs::replace_disprove_scripts()

```
for (digit, places) in replacement_places.bitvm_pks.0.iter().enumerate() {
    for (pk_idx, places) in places.iter().enumerate() { // @audit Millions of iterations possible
        for (script_idx, pos) in places.iter() {
            result[*script_idx][*pos..*pos + 20]
                .copy_from_slice(&pks.bitvm_pks.0[digit][pk_idx]); // @audit Memory intensive operations
        }
    }
}
```

The likelihood is medium because while the attack requires aggregator-level access, it can be executed with a single malformed gRPC call that causes significant service degradation through memory and CPU exhaustion.

Recommendations

Add validation limits on the size of individual Winternitz keys in the `TryFrom` implementation:

core/src/rpc/parser/mod.rs::TryFrom<WinternitzPubkey>

```
impl TryFrom for winternitz::PublicKey {
    fn try_from(value: WinternitzPubkey) -> Result {
        let inner = value.digit_pubkey;

        // Add reasonable size limit per key
        const MAX_DIGITS_PER_KEY: usize = 100;
        if inner.len() > MAX_DIGITS_PER_KEY {
            return Err(BridgeError::ExcessiveKeySize(inner.len()));
        }

        // Add total memory limit check
        const MAX_BYTES_PER_KEY: usize = MAX_DIGITS_PER_KEY * 20;
        let total_bytes = inner.len() * 20;
        if total_bytes > MAX_BYTES_PER_KEY {
            return Err(BridgeError::MemoryLimitExceeded(total_bytes));
        }

        // ... existing validation
    }
}
```

Additionally, add memory monitoring during script generation to detect and prevent excessive resource usage:

core/src/bitvm_client.rs::replace_disprove_scripts()

```
pub fn replace_disprove_scripts(pks: &ClementineBitVMPublicKeys) -> Result, BridgeError> {
    let estimated_operations = calculate_replacement_operations(pks);
    if estimated_operations > MAX_REPLACEMENT_OPERATIONS {
        return Err(BridgeError::ExcessiveComputationRequired);
    }

    // ... existing logic
}
```

Resolution

Commit fb47318 resolves the issue by adding validation limits on the size of individual Winternitz keys, including the number of digits for each key and the total bytes used across all keys.

CMT-09	Incorrect Profitability Check May Lead To Operator Fund Loss		
Asset	`core/src/operator.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `withdraw()` function in `core/src/operator.rs` contains a flawed profitability check that does not account for Bitcoin transaction fees. This could lead to operators losing funds when processing withdrawals, especially during periods of high network fees.

The `is_profitable()` function is called before the `Payout` transaction is fully constructed and funded. The transaction fee is only determined later by the `fund_raw_transaction()` RPC call. This means the initial profitability check does not include a major cost for the operator. A user can request a withdrawal that appears profitable, but becomes unprofitable for the operator once network fees are included.

The current check is performed in the `withdraw()` function as shown below:

core/src/operator.rs::withdraw()

```rust
if !Self::is_profitable(
    input_utxo.txout.value,
    output_txout.value,
    self.config.protocol_paramset().bridge_amount,
    operator_withdrawal_fee_sats,
) {
    return Err(eyre::eyre!("Not enough fee for operator").into());
}
```

The `is_profitable()` function itself has misleading logic and does not factor in transaction fees:

core/src/operator.rs::is_profitable()

```rust
/// Checks if the withdrawal amount is within the acceptable range.
fn is_profitable(
    input_amount: Amount,
    withdrawal_amount: Amount,
    bridge_amount_sats: Amount,
    operator_withdrawal_fee_sats: Amount,
) -> bool {
    if withdrawal_amount
        .to_sat()
        .wrapping_sub(input_amount.to_sat())
        > bridge_amount_sats.to_sat()
    {
        return false;
    }

    // Calculate net profit after the withdrawal.
    // @audit net_profit does not take into account transaction fees
    let net_profit = bridge_amount_sats - withdrawal_amount;

    // Net profit must be bigger than withdrawal fee.
    net_profit >= operator_withdrawal_fee_sats
}
```

The operator's actual profit is the difference between the bridge amount (`bridge_amount_sats`) and the withdrawal amount that the operator needs to cover including the Bitcoin transaction fee (`withdrawal_amount - input_amount + transaction_fee`). By not including the transaction fee in the calculation, an operator may end up with a negative `net_profit`.

Furthermore, the operation does not consider `input_amount` as part of profit. This will usually be zero however, the user may set this to any value.

This vulnerability has a medium impact as it can cause operators to slowly lose funds. The likelihood is medium, as it is dependent on network fee conditions and could be accidentally or deliberately triggered by a user crafting a specific withdrawal request during high-fee periods.

Recommendations

Update `is_profitable()` implementation to account for the Bitcoin transaction fee.

Resolution

The issue has been closed as won't fix. The operator fee is configured to be sufficient to cover the transaction fees, and the operator is expected to handle the transaction fees when processing withdrawals.

CMT-10	Unbounded Memory Growth In Verifier's `AllSessions` Leads To Denial Of Service		
Asset	core/src/verifier.rs, core/src/aggregator.rs		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The verifier's `AllSessions` mapping, which stores MuSig2 nonce sessions, grows indefinitely without a mechanism to clear old or abandoned sessions. This can lead to unbounded memory usage, eventually causing the verifier to crash, resulting in a denial of service.

In `core/src/verifier.rs`, the `nonce_gen()` function is responsible for creating new nonce sessions. It generates secret nonces for a MuSig2 signing session, stores them in a `NonceSession` struct, and then inserts this session into the `AllSessions.sessions` `HashMap` with a unique `session_id`.

core/src/verifier.rs::nonce_gen()

```
pub async fn nonce_gen(&self, num_nonces: u32) -> Result<(u32, Vec), BridgeError> {
    let (sec_nonces, pub_nonces): (Vec, Vec) = (0..num_nonces)
        .map(|_| {
            // nonce pair needs keypair and a rng
            let (sec_nonce, pub_nonce) =
                musig2::nonce_pair(&self.signer.keypair, &mut secp256k1::rand::thread_rng())?;
            Ok((sec_nonce, pub_nonce))
        })
        .collect::, BridgeError>>()?
        .into_iter()
        .unzip(); // TODO: fix extra copies

    let session = NonceSession { nonces: sec_nonces };

    // save the session
    let session_id = {
        let all_sessions = &mut *self.nonces.lock().await;
        let session_id = all_sessions.cur_id;
        all_sessions.sessions.insert(session_id, session);
        all_sessions.cur_id += 1;
        session_id
    };

    Ok((session_id, pub_nonces))
}
```

The issue is that there is no corresponding cleanup logic to remove a `NonceSession` from `all_sessions` after it has been used or if it's abandoned. An aggregator might request nonces from verifiers but fail to proceed with the signing process due to network issues, a verifier being offline, or other errors.

For example, in the `optimistic_payout()` function in `core/src/rpc/aggregator.rs`, `create_nonce_streams()` is called, which in turn calls `nonce_gen()` on each verifier. If a subsequent operation like `get_next_pub_nonces()` fails, the aggregator returns an error, and the nonce sessions created in the verifiers are never used or cleared, leading to a memory leak.

core/src/rpc/aggregator.rs::optimistic_payout()

```
// get which verifiers participated in the deposit to collect the optimistic payout tx signature
let verifiers = self.get_participating_verifiers(&deposit_data).await?;
let (first_responses, mut nonce_streams) = {
    create_nonce_streams(
        verifiers.clone(),
        1,
        #[cfg(test)]
        &self.config,
    )
    .await?
};
// collect nonces
// @audit if this fails, then the nonces are never cleared from memory
let pub_nonces = get_next_pub_nonces(&mut nonce_streams)
    .await
    .wrap_err("Failed to aggregate nonces for optimistic payout")
    .map_to_status()?;
```

The impact is medium as this memory leak will cause the verifier process to consume increasing memory over time, eventually leading to a crash and a denial of service that requires a manual restart. The likelihood is medium because while it requires a failure condition, such failures are plausible in a distributed system.

Recommendations

To mitigate this, a cleanup mechanism for stale sessions should be implemented.

One approach is to add a timestamp to each `NonceSession` upon creation. A separate background task could then periodically scan `all_sessions` and remove any sessions that have expired (e.g., are older than a reasonable timeout like 5 minutes).

This would prevent abandoned sessions from accumulating and ensure the verifier's memory usage remains bounded.

Resolution

The issue was resolved in commit 44bd7cf by enforcing limits on session size and count. The fix includes a cleanup mechanism that clears older sessions when these limits are exceeded.

CMT-11	Unbounded Nonce Generation In `nonce_gen()` Leads To Denial Of Service		
Asset	`core/src/verifier.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `nonce_gen()` function in `core/src/verifier.rs` lacks input validation on the `num_nonces` parameter. A malicious or compromised aggregator can request an extremely large number of nonces, causing the verifier process to attempt a massive memory allocation, which will likely lead to a crash and a denial-of-service.

The `nonce_gen()` function is an RPC endpoint that an aggregator calls to obtain public nonces from a verifier for a MuSig2 signing session. The function takes a `u32` argument, `num_nonces`, and generates that many secret/public nonce pairs. The secret nonces are stored in memory on the verifier for the duration of the signing session.

core/src/verifier.rs::nonce_gen()

```
pub async fn nonce_gen(&self, num_nonces: u32) -> Result<(u32, Vec), BridgeError> {
    let (sec_nonces, pub_nonces): (Vec, Vec) = (0..num_nonces)
        .map(|_| {
            // nonce pair needs keypair and a rng
            let (sec_nonce, pub_nonce) =
                musig2::nonce_pair(&self.signer.keypair, &mut secp256k1::rand::thread_rng())?;
            Ok((sec_nonce, pub_nonce))
        })
        .collect::, BridgeError>>()?
        .into_iter()
        .unzip(); // TODO: fix extra copies

    let session = NonceSession { nonces: sec_nonces };

    // save the session
    let session_id = {
        let all_sessions = &mut *self.nonces.lock().await;
        let session_id = all_sessions.cur_id;
        all_sessions.sessions.insert(session_id, session);
        all_sessions.cur_id += 1;
        session_id
    };

    Ok((session_id, pub_nonces))
}
```

A malicious aggregator can call this function with a large value for `num_nonces`, such as `u32::MAX`. This forces the verifier to allocate memory for `4,294,967,295` secret nonces. An attacker could perform this attack multiple times and use this to systematically take verifiers offline, halting the bridge. Furthermore, by repeatedly requesting large numbers of nonces, an attacker could attempt a "birthday attack" to find a public nonce collision, which could potentially be used to compromise the verifier's signing key.

The impact is rated medium because taking verifiers offline constitutes a denial-of-service for the entire bridge. The likelihood is rated medium because the attack must be initiated by a malicious aggregator.

Recommendations

Introduce a strict upper bound on the `num_nonces` parameter within the `nonce_gen()` function. This limit should be a constant, chosen to be safely above the maximum number of nonces required for any legitimate signing session but low enough to prevent a DoS attack.

Additionally, consider implementing rate-limiting on the `nonce_gen()` RPC endpoint to prevent an attacker from spamming the function with multiple calls to work around the per-call limit.

Resolution

The issue was resolved in commit 44bd7cf by introducing a hardcoded limit on the number of `num_nonces` that can be generated.

CMT-12	Predictable Nonce Session ID Leads To Denial Of Service		
Asset	core/src/verifier.rs		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `nonce_gen()` function uses a simple incrementing `u32` counter for session IDs, which can overflow after 2^{32} calls. An attacker can repeatedly call a public endpoint like `optimistic_payout()` to trigger this overflow, causing the session ID counter to wrap around. This allows them to overwrite an existing, in-progress signing session's nonces in memory, leading to a Denial of Service for critical operations like deposits.

In `core/src/verifier.rs`, the verifier manages nonce signing sessions using the `AllSessions` struct. This struct contains `cur_id`, a `u32` counter that serves as the next session ID.

core/src/verifier.rs::AllSessions

```
pub struct AllSessions {
    pub cur_id: u32,
    pub sessions: HashMap<u32, NonceSession>,
}
```

Each time the `nonce_gen()` function is called to create a new session, it uses the current value of `cur_id` as the `session_id` and then increments it by one.

core/src/verifier.rs::nonce_gen()

```
    pub async fn nonce_gen(&self, num_nonces: u32) -> Result<(u32, Vec), BridgeError> {
// ... existing code ...
        // save the session
        let session_id = {
            let all_sessions = &mut *self.nonces.lock().await;
            let session_id = all_sessions.cur_id;
            all_sessions.sessions.insert(session_id, session);
            // @audit this can overflow
            all_sessions.cur_id += 1;
            session_id
        };

        Ok((session_id, pub_nonces))
    }
```

An attacker can repeatedly call any gRPC endpoint that results in a call to `nonce_gen()`, such as `optimistic_payout()`. After 2^{32} calls, the `cur_id` will overflow and wrap around to 0. If a legitimate, long-running process like a deposit has an active session with a low ID (e.g., `session_id=1`), the attacker can force the creation of a new session that overwrites the nonces for the legitimate one. When the original process attempts to use its nonces, it will fail, resulting in a denial of service.

The impact is rated medium as this can halt core bridge functionality like deposits, but does not lead to a direct loss of funds. The likelihood is rated medium because the attack requires a very large number of transactions (2^{32}) to be sent, which is a significant undertaking. However, the vulnerability can be triggered by any external party.

Recommendations

To mitigate this, the session ID generation should be made non-sequential and resistant to overflow.

It is recommended to change the `session_id` type from `u32` to `u128` and generate a cryptographically secure random number for each new session ID. A check for collisions in the `sessions` map should be performed, with the ID being regenerated in the highly unlikely event of a collision.

Resolution

The issue was resolved in commit 44bd7cf by switching from sequential to randomly generated session IDs.

CMT-13	Index-Out-Of-Bounds Panic In `parse_deposit_sign_session()` Allows Remote DoS		
Asset	`core/src/rpc/parser/verifier.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

An out-of-bounds memory access vulnerability in the `parse_deposit_sign_session()` function can be triggered by a malicious aggregator, causing the verifier process to panic. This can lead to a Denial of Service (DoS) for the deposit functionality of the bridge.

The function `parse_deposit_sign_session()` is called by the `deposit_sign()` and `deposit_finalize()` RPC handlers in `core/src/rpc/verifier.rs`. It receives a `deposit_sign_session` object from the gRPC stream. Inside this function, an index `verifier_idx` is calculated based on the content of `deposit_sign_session.deposit_params`. This index is then used to directly access the `deposit_sign_session.nonce_gen_first_responses` array without performing any bounds checks.

core/src/rpc/parser/verifier.rs::parse_deposit_sign_session()

```rust
pub fn parse_deposit_sign_session(
    deposit_sign_session: clementine::DepositSignSession,
    verifier_pk: &PublicKey,
) -> Result<(DepositData, u32), Status> {
// ... existing code ...
    let verifier_idx = deposit_data
        .get_verifier_index(verifier_pk)
        .map_err(|e| Status::invalid_argument(e.to_string()))?;

    // @audit No bounds check is performed here.
    // @audit If `verifier_idx` >= `nonce_gen_first_responses.len()`, this will panic.
    let session_id = deposit_sign_session.nonce_gen_first_responses[verifier_idx].id;

    Ok((deposit_data, session_id))
}
```

A malicious aggregator can craft a malicious `DepositSignSession` where the length of the `nonce_gen_first_responses` array is less than the derived `verifier_idx`. This will cause an out-of-bounds access, crashing the verifier.

The impact is rated as medium because successfully exploiting this vulnerability would crash verifiers that are trying to parse the `DepositSignSession`. The likelihood is rated as medium, as it requires a malicious aggregator to send the malicious payload.

Recommendations

Perform a bounds check before accessing the `nonce_gen_first_responses` array. Use the `.get()` method, which returns an `Option`, and handle the `None` case gracefully by returning an error instead of panicking.

Resolution

The issue was resolved in commit 1cb048a by implementing the recommended fix, which properly handles potential index out-of-bounds errors to prevent panics.

CMT-14	Nonce Session Hijacking In `sign_optimistic_payout()`		
Asset	`core/src/verifier.rs`, `core/src/rpc/aggregator.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `sign_optimistic_payout()` function in `core/src/verifier.rs` is vulnerable to a nonce session hijacking attack, which can lead to a denial-of-service condition for other deposit processes. The function takes a `nonce_session_id` as a parameter to retrieve a secret nonce for signing, but it does not validate that this session ID was created for the specific `deposit_id` being processed.

A malicious aggregator with the ability to call the verifier's `sign_optimistic_payout()` RPC endpoint can exploit this. By providing valid arguments for an optimistic payout but substituting a `nonce_session_id` from a different, ongoing deposit session, the malicious aggregator can consume a nonce intended for that other session. Since nonces are single-use, this will cause the legitimate deposit process to fail when it attempts to use the now-consumed nonce, resulting in a denial of service for that deposit.

The following snippet from `sign_optimistic_payout()` shows where the nonce is retrieved without proper validation:

core/src/verifier.rs::sign_optimistic_payout()

```
let opt_payout_secnonce = {
    let mut session_map = self.nonces.lock().await;
    let session = session_map
        .sessions
        // @audit The nonce_session_id is used here without being verified
        .get_mut(&nonce_session_id)
        .ok_or_else(|| eyre::eyre!("Could not find session id {nonce_session_id}"))?;
    session
        .nonces
        // @audit A nonce is consumed from another session, causing a DoS.
        .pop()
        .ok_or_eyre("No move tx secnonce in session")?
};
```

The impact is classified as medium because this vulnerability leads to a denial of service for both deposits and withdrawals. The likelihood is medium, as only the malicious aggregator which can call the verifier's gRPC methods can exploit this. Keep in mind that they can use a past processed withdrawal as parameters to sign the optimistic payout. The attacker does not need to perform their own withdrawal.

Recommendations

To mitigate this vulnerability, the `nonce_session_id` should be cryptographically bound to the context in which it is intended to be used. One way to achieve this is to incorporate the `deposit_id` (or a hash of other relevant context) into the nonce generation process itself, and verify it during signing.

Specifically, when generating nonces in `nonce_gen()`, a context string or hash could be included in the data signed to create the nonce. This context should be passed to `sign_optimistic_payout()`.

Resolution

The issue has been closed as won't fix. The development team response is as follows:

> *We plan to use pre generated nonces to improve performance, so adding deposit related info to the nonce generation would prevent that optimization.*

CMT-15	Slashed Operator Halts New Deposits, Causing Denial Of Service		
Asset	`core/src/aggregator.rs`, `core/src/verifier.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

When an operator is slashed rendering their collateral unusable, the bridge is unable to process any new deposits. This results in a denial-of-service that requires manual intervention from all network participants to resolve.

The root cause is that the aggregator determines the set of operators for a new deposit based on a static configuration loaded at startup. It does not dynamically check the onchain status of an operator's collateral. When an operator is slashed, their collateral is spent, but the aggregator remains unaware of this change and continues to include the slashed operator in the `DepositData` for all new deposits.

In `core/src/rpc/aggregator.rs`, the `new_deposit()` function constructs `DepositData` using the full list of operators derived from its configuration:

core/src/rpc/aggregator.rs::new_deposit()

```
let deposit_data = DepositData {
    deposit: deposit_info,
    nofn_xonly_pk: None,
    actors: Actors {
        verifiers: self.get_verifier_keys(),
        watchtowers: vec![],
        operators: self.get_operator_keys(),
    },
    security_council: self.config.security_council.clone(),
};
```

This `DepositData` is then sent to all verifiers. Each verifier validates it by calling `is_deposit_valid()`. This function, defined in `core/src/verifier.rs`, checks if every operator included in the deposit still has usable collateral. For a slashed operator, this check will fail.

core/src/verifier.rs::is_deposit_valid()

```
    // @audit if an operator is in the protocol and deposit, but doesn't have usable collateral, the deposit is invalid
    if operator_xonly_pks.contains(&xonly_pk) && !is_collateral_usable {
        tracing::warn!(
            "Operator {:?} is in the deposit but its collateral is spent, operator cannot fulfill withdrawals anymore",
            xonly_pk
        );
        return Ok(false);
    }
}
```

Because `is_deposit_valid()` returns `false`, the deposit process is aborted. Since the aggregator will continue to include the slashed operator in all subsequent deposit requests, the bridge is effectively halted. The only way to recover is for the aggregator to manually remove the slashed operator from the `BridgeConfig` and restart the service.

This vulnerability causes a denial-of-service that is difficult to recover from, however it is still possible with manual intervention and coordination between network participants, hence the medium impact. The likelihood is considered low because it requires an operator to be slashed, which is an exceptional but expected event within the protocol's security model.

Note that the same issue exists for operators that are on the last round (the `num_round_txs + 1` round). The last round should only be used for reimbursements, as kickoffs in this round cannot be reimbursed. Operators on the last round should not be included in new deposits.

Recommendations

To mitigate this vulnerability, the aggregator should dynamically filter the list of operators before including them in a new deposit. Instead of using the static list from its configuration directly, the aggregator should first check the onchain collateral status for each operator. Only operators with usable collateral should be included in the `DepositData`.

The `is_deposit_valid()` function should also make sure that all the operators are not in the last round.

Resolution

The issue has been closed as won't fix. The development team response is as follows:

> *It's ok that the operator removal requires manual operation since there will be few.*

CMT-16	Operator Collateral Validation Accepts Mempool Transactions Leading To Potential Spoofing		
Asset	core/src/extended_rpc.rs, core/src/verifier.rs		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `collateral_check()` function uses `get_tx_of_txid()` to validate operator collateral, which internally calls Bitcoin Core's `getrawtransaction` RPC. This RPC method returns transactions from both the mempool and confirmed blocks, allowing operators to potentially spoof verifiers with collateral transactions that have insufficient fees and may never be included in a block.

The vulnerability occurs in the collateral validation flow where `get_tx_of_txid()` retrieves the collateral funding transaction:

core/src/extended_rpc.rs::collateral_check()

```
// @audit This call will succeed even if the transaction is only in mempool
let tx = self
    .get_tx_of_txid(&operator_data.collateral_funding_outpoint.txid)
    .await
    .wrap_err(format!(
        "Failed to find collateral utxo in chain for outpoint {:?}",
        operator_data.collateral_funding_outpoint
    ))?;
```

The `get_tx_of_txid()` function uses `get_raw_transaction(txid, None)` which, according to Bitcoin Core documentation, *"returns a transaction if it is in the mempool or any block"* when `-txindex` is enabled. This means operators can broadcast collateral transactions with very low fees that remain in the mempool indefinitely without being mined.

This issue affects the verifier's `is_deposit_valid()` function and `set_operator()` function, both of which rely on `collateral_check()` to validate operator eligibility. Verifiers will incorrectly accept operators whose collateral transactions exist only in the mempool.

While operators cannot post `Round` transactions if their collateral transaction is not actually confirmed onchain, this creates a suboptimal user experience and potential for confusion in the protocol's operation, and can lead to a denial of service for new deposits similar to what is described in CMT-15 if the transaction is purged from the mempool.

The impact is classified as low because it results in incorrect business logic where operators appear eligible when they shouldn't be, though the operator cannot start valid `Rounds` without the collateral transaction being onchain. The likelihood is medium because it requires operators to intentionally broadcast low-fee transactions, but this can be easily accomplished by any operator.

Recommendations

Modify the `get_tx_of_txid()` function to only accept confirmed transactions by using `get_raw_transaction_info` and checking for the presence of a `blockhash`, or add an explicit confirmation check in `collateral_check()`.

Alternatively, use the existing `is_tx_on_chain()` method to verify the transaction is confirmed before proceeding with collateral validation.

Resolution

The issue was resolved in commit 633646c by implementing the recommended fix, which adds a check using `is_tx_on_chain()` to ensure the transaction is confirmed onchain before proceeding with collateral validation.

CMT-17	Missing HTTP Timeouts In Citrea Client		
Asset	`core/src/citrea.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The Citrea client creates two HTTP clients without configuring proper timeout values, which can lead to indefinite blocking of operations during network issues or unresponsive remote services.

In the `CitreaClient::new()` function on lines [330, 336], two `HttpClientBuilder` instances are created using `HttpClientBuilder::default()` without any timeout configuration:

core/src/citrea.rs::CitreaClient::new()

```
let client = HttpClientBuilder::default()
    .build(citrea_rpc_url)
    .wrap_err("Failed to create Citrea RPC client")?;
```

core/src/citrea.rs::CitreaClient::new()

```
let light_client_prover_client = HttpClientBuilder::default()
    .build(light_client_prover_url)
    .wrap_err("Failed to create Citrea LCP RPC client")?;
```

Without explicit timeout configuration, these HTTP clients may hang indefinitely when making requests to unresponsive endpoints. This can cause the entire bridge operator to become unresponsive, particularly during network partitions, server outages, or when remote services experience high latency.

The impact is classified as low because while it doesn't directly compromise security, it can cause operational disruptions. The likelihood is low as network issues and service unavailability are common in distributed systems.

Recommendations

Configure appropriate timeout values for both HTTP clients using the `HttpClientBuilder::request_timeout()` and `HttpClientBuilder::connection_timeout()` methods:

core/src/citrea.rs::CitreaClient::new()

```
let client = HttpClientBuilder::default()
    .request_timeout(Duration::from_secs(30))
    .connection_timeout(Duration::from_secs(10))
    .build(citrea_rpc_url)
    .wrap_err("Failed to create Citrea RPC client")?;
```

core/src/citrea.rs::CitreaClient::new()

```
let light_client_prover_client = HttpClientBuilder::default()
    .request_timeout(Duration::from_secs(30))
    .connection_timeout(Duration::from_secs(10))
    .build(light_client_prover_url)
    .wrap_err("Failed to create Citrea LCP RPC client")?;
```

Consider making the timeout values configurable through the application configuration to allow fine-tuning based on deployment environment requirements.

Resolution

Timeouts have been added to the HTTP clients in commit 1bd1a36.

CMT-18	Missing gRPC Server Hardening In `create_grpc_server()`		
Asset	`core/src/servers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `create_grpc_server()` function in `core/src/servers.rs` constructs two different `tonic::transport::Server` builders: one for TCP and one for a Unix socket, yet neither applies hardening parameters such as message-size limits, timeouts or concurrency caps. An attacker that can open a connection to either endpoint can monopolise server resources with oversized payloads or slow-loris traffic, causing a recoverable but disruptive denial-of-service.

The TCP variant is currently:

core/src/servers.rs::create_grpc_server()

```
let server_builder = tonic::transport::Server::builder()
    .layer(AddMethodMiddlewareLayer)
    .tls_config(tls_config)
    .wrap_err("Failed to configure TLS")?
    .add_service(service);
```

The Unix-socket variant is similarly unprotected:

core/src/servers.rs::create_grpc_server()

```
let server_builder = tonic::transport::Server::builder()
    .layer(AddMethodMiddlewareLayer)
    .add_service(service);
```

Missing defences include:

1. `.max_decoding_message_size()` / `.max_encoding_message_size()` – prevent memory-exhaustion DoS.
2. `.timeout()` – abort long-running RPCs & stop slow-loris reads/writes.
3. `.concurrency_limit_per_connection()` – bound per-client parallelism.
4. `.tcp_keepalive()` – detect half-open sockets (TCP only).
5. `.http2_keepalive_*()` – keep HTTP/2 channels healthy.
6. Any analogous limits for the Unix builder (even though TCP keep-alive is N/A).

Note the issue is rated as low severity as most individual gRPC request include timeouts and there is a default incoming messages size of 4MB applied by gRPC.

Recommendations

Apply identical hardening to both builders. For example, for TCP:

core/src/servers.rs::create_grpc_server()

```
let server_builder = tonic::transport::Server::builder()
    .layer(AddMethodMiddlewareLayer)
    .max_decoding_message_size(config.max_grpc_message_size.unwrap_or(4 * 1024 * 1024))
    .max_encoding_message_size(config.max_grpc_message_size.unwrap_or(4 * 1024 * 1024))
    .timeout(Duration::from_secs(config.grpc_timeout_secs.unwrap_or(30)))
    .tcp_keepalive(Some(Duration::from_secs(config.tcp_keepalive_secs.unwrap_or(60))))
    .concurrency_limit_per_connection(config.grpc_concurrency_limit.unwrap_or(50))
    .http2_keepalive_interval(Some(Duration::from_secs(30)))
    .http2_keepalive_timeout(Some(Duration::from_secs(5)))
    .http2_adaptive_window(true)
    .tls_config(tls_config)
    .wrap_err("Failed to configure TLS")?
    .add_service(service);
```

For Unix sockets, implement the same limits minus TCP related configurations:

core/src/servers.rs::create_grpc_server()

```
let server_builder = tonic::transport::Server::builder()
    .layer(AddMethodMiddlewareLayer)
    .max_decoding_message_size(config.max_grpc_message_size.unwrap_or(4 * 1024 * 1024))
    .max_encoding_message_size(config.max_grpc_message_size.unwrap_or(4 * 1024 * 1024))
    .timeout(Duration::from_secs(config.grpc_timeout_secs.unwrap_or(30)))
    .concurrency_limit_per_connection(config.grpc_concurrency_limit.unwrap_or(50))
    .add_service(service);
```

Include fields in `BridgeConfig` for these tunables and document reasonable defaults. Integration tests should verify that oversized messages are rejected and that idle connections are closed after the configured timeout.

Resolution

Server hardening has been added in commit 1ab5803.

A range of hardening parameters have been added to both the TCP and Unix socket gRPC server builders, including message size limits, timeouts, concurrency limits, and keep-alive settings.

CMT-19	Unbounded `max_encoding_message_size()` Allows Excessive gRPC Responses		
Asset	`core/src/rpc/clementine.rs, core/src/servers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The gRPC client helpers generated in `core/src/rpc/clementine.rs` expose a `max_encoding_message_size()` builder that can be used to cap the size of *outbound* gRPC messages. However, the servers are instantiated without ever calling this method, leaving the default limit of `usize::MAX` ($2^{64} - 1$ bytes):

core/src/rpc/clementine.rs::max_encoding_message_size()

```
/// Limits the maximum size of an encoded message.
///
/// Default: `usize::MAX`
#[must_use]
pub fn max_encoding_message_size(mut self, limit: usize) -> Self {
    self.inner = self.inner.max_encoding_message_size(limit);
    self
}
```

The server creation code also omits any explicit size restriction:

core/src/servers.rs::create_grpc_server()

```
let server_builder = tonic::transport::Server::builder()
    .layer(AddMethodMiddlewareLayer)
    .tls_config(tls_config)
    .wrap_err("Failed to configure TLS")?
    .add_service(service);
```

Because of this, the service may allocate and serialize arbitrarily large responses if an attacker can coerce it into producing them, potentially exhausting memory or bandwidth and leading to a denial-of-service.
Note that inbound messages are already capped at the `tonic` default of 4 MiB, so the attacker must rely on normal-sized requests that trigger disproportionately large responses (for example by including very large vectors in legitimate response fields).

The effect is limited to availability; no funds can be lost, and recovery only requires restarting the process once memory pressure subsides, hence the impact is assessed as low.
Triggering such oversized responses requires specific knowledge of the system's data paths and may be hard in practice, therefore the likelihood is also low.

Recommendations

Explicitly set symmetric limits for both encoding and decoding on every gRPC server and client:

core/src/servers.rs::create_grpc_server()

```
let server_builder = tonic::transport::Server::builder()
    .max_encoding_message_size(4 * 1024 * 1024)      // 4 MiB
    .max_decoding_message_size(4 * 1024 * 1024)
    ...
```

and, after connecting with a client:

```
let client = ClementineAggregatorClient::connect(url).await?
    .max_encoding_message_size(4 * 1024 * 1024)
    .max_decoding_message_size(4 * 1024 * 1024);
```

Choose a value that matches the maximum size of any expected legitimate message (4 MiB aligns with `tonic` defaults).

Resolution

Message encoding sizes have been bounded in commit 1ab5803.

CMT-20	Missing Input Validation For Empty Public Keys In MuSig2 Key Aggregation		
Asset	`core/src/musig2.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `create_key_agg_cache()` function in `core/src/musig2.rs` does not validate that the input `public_keys` vector contains at least one public key before proceeding with MuSig2 key aggregation. While the underlying `KeyAggCache::new()` function does not immediately panic when passed an empty slice, it creates an invalid cryptographic state that violates the fundamental assumptions of the MuSig2 protocol, which requires at least one participant.

This lack of validation allows the creation of a `KeyAggCache` with an empty key set, which represents an undefined cryptographic state. Although this may not cause immediate crashes, it could lead to unpredictable behaviour in downstream cryptographic operations such as signature generation, verification, or key derivation processes.

core/src/musig2.rs::create_key_agg_cache()

```
fn create_key_agg_cache(
    mut public_keys: Vec,
    mode: Option,
) -> Result {
    public_keys.sort();
    let secp_pubkeys: Vec =
        public_keys.iter().map(|pk| to_secp_pk(*pk)).collect();
    let pubkeys_ref: Vec<&secp256k1::PublicKey> = secp_pubkeys.iter().collect();
    let pubkeys_ref = pubkeys_ref.as_slice();

    let mut musig_key_agg_cache = KeyAggCache::new(SECP256K1, pubkeys_ref);
```

If this condition is triggered, it would result in the creation of an invalid MuSig2 aggregation context that could produce undefined cryptographic behaviour in subsequent operations. This represents a violation of protocol invariants that could potentially lead to cryptographic failures or inconsistent results during signing operations.

However, the likelihood of this occurring is very low in normal operation, as the system is designed to always have at least one verifier configured, and the input validation in calling functions should prevent empty key sets from reaching this point under typical usage scenarios.

Recommendations

Add input validation to ensure the public keys vector is non-empty:

core/src/musig2.rs::create_key_agg_cache()

```
fn create_key_agg_cache(
    mut public_keys: Vec,
    mode: Option,
) -> Result {
    if public_keys.is_empty() {
        return Err(BridgeError::from(eyre::eyre!(
            "Cannot create key aggregation cache: no public keys provided"
        )));
    }

    public_keys.sort();
    let secp_pubkeys: Vec =
        public_keys.iter().map(|pk| to_secp_pk(*pk)).collect();
    // ... rest of function unchanged
}
```

This change ensures that the function fails gracefully with a descriptive error message rather than creating an invalid cryptographic state, improving the robustness and debuggability of the system.

Resolution

The recommendation was implemented in commit 099abd0 by adding a check to handle an empty `public_keys` vector.

CMT-21	Empty Nonce Array Panic In MuSig2 Aggregation		
Asset	`core/src/musig2.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The `aggregate_nonces()` function in `core/src/musig2.rs` can panic when called with an empty array of public nonces. This vulnerability occurs when the underlying MuSig2 library's `AggregatedNonce::new()` function receives an empty slice, causing the aggregator service to crash during the nonce aggregation process.

The issue stems from the `get_next_pub_nonces()` function in the aggregator, which can return an empty `Vec` when no verifiers are participating in the signing process. This can happen due to configuration issues, network problems, or if `get_participating_verifiers()` returns an empty participant set. When this empty vector is passed to `aggregate_nonces()`, the MuSig2 library panics instead of gracefully handling the error condition.

If this vulnerability is triggered, the aggregator service will crash and become unavailable, requiring manual restart to restore bridge functionality. This represents a denial of service condition that could disrupt normal bridge operations and prevent users from processing deposits or withdrawals.

However, the likelihood of this issue occurring is very low as it requires either configuration problems or complete network failure that prevents communication with all verifiers. In normal operating conditions with properly configured verifiers, this code path should not be reachable.

Recommendations

Update `aggregate_nonces()` to return a `Result` and propagate errors properly:

core/src/musig2.rs::aggregate_nonces()

```rust
pub fn aggregate_nonces(pub_nonces: &[&PublicNonce]) -> Result {
    if pub_nonces.is_empty() {
        return Err(BridgeError::from(eyre::eyre!(
            "Cannot aggregate nonces: no public nonces provided"
        )));
    }
    Ok(AggregatedNonce::new(SECP256K1, pub_nonces))
}
```

This ensures graceful error handling instead of panics, allowing the system to return appropriate error responses to clients.

Resolution

Error handling has been improved in commit 099abd0. The function `aggregate_nonces()` now checks for an empty array of public nonces and returns an error instead of panicking.

CMT-22	Inefficient Winternitz Key Validation Order Allows Unnecessary Processing		
Asset	core/src/verifier.rs		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `set_operator_keys()` function performs expensive Winternitz key processing before validating the expected key count, allowing unnecessary resource consumption when invalid payloads are submitted.

The current implementation processes all submitted Winternitz keys through the `TryFrom` conversion before checking if the count matches the expected value of 381 keys. This means that an attacker can submit an incorrect number of keys (e.g., thousands) and force the system to process every single key before the validation failure occurs.

The problematic code flow in `set_operator_keys()`:

core/src/verifier.rs::set_operator_keys()

```
let winternitz_keys: Vec = keys
    .winternitz_pubkeys
    .into_iter()
    .map(|x| x.try_into()) // @audit Processes ALL keys regardless of count
    .collect::>()?;

if winternitz_keys.len() != ClementineBitVMPublicKeys::number_of_flattened_wpks() {
    // @audit Count validation happens AFTER expensive processing
    return Err(...);
}
```

Each `try_into()` call performs validation and memory allocation for individual key digits. When an attacker submits 10,000 keys instead of the expected 381, the system will process all 10,000 keys, perform memory allocations, and execute validation logic before discovering the count mismatch and rejecting the request.

This creates an opportunity for minor denial-of-service attacks where attackers can waste CPU cycles and memory allocation on obvious invalid requests. The impact is limited since the processing eventually fails, but it represents inefficient resource utilisation.

Recommendations

Move the count validation before the expensive key processing to fail fast on invalid payloads:

core/src/verifier.rs::set_operator_keys()

```
// Check count first before any processing
if keys.winternitz_pubkeys.len() != ClementineBitVMPublicKeys::number_of_flattened_wpks() {
    tracing::error!(
        "Invalid number of winternitz keys received from operator {:?}: got: {} expected: {}",
        operator_xonly_pk,
        keys.winternitz_pubkeys.len(),
        ClementineBitVMPublicKeys::number_of_flattened_wpks()
    );
    return Err(eyre::eyre!(
        "Invalid number of winternitz keys received from operator {:?}: got: {} expected: {}",
        operator_xonly_pk,
        keys.winternitz_pubkeys.len(),
        ClementineBitVMPublicKeys::number_of_flattened_wpks()
    ).into());
}

// Only process keys if count is correct
let winternitz_keys: Vec = keys
    .winternitz_pubkeys
    .into_iter()
    .map(|x| x.try_into())
    .collect::>()?;
```

This simple reordering eliminates unnecessary processing for obvious invalid requests while maintaining the same validation logic.

Resolution

The order of validation has been changed in commit 7684945. The count check is now performed before processing the Winternitz keys, preventing unnecessary resource consumption for invalid payloads.

CMT-23	Delayed Signature Verification Can Lead To DoS In `optimistic_payout()`		
Asset	`core/src/rpc/aggregator.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Medium

Description

The `optimistic_payout()` function is vulnerable to a Denial of Service (DoS) attack because it does not verify the user-provided `input_signature` before proceeding with computationally expensive operations.

The `optimistic_payout()` function is responsible for processing withdrawals from the Clementine bridge, allowing users to retrieve Bitcoin after burning corresponding `cBTC` tokens in Citrea. During this process, users must provide an `input_signature` that authorises the withdrawal.

The `input_signature` is not verified early in the execution flow. Instead, the signature validation only occurs implicitly when the transaction would be broadcast to the Bitcoin network, after numerous resource-intensive operations have already been performed. These operations include communication with multiple verifiers, cryptographic nonce generation and aggregation, and transaction signing etc.

Since the `optimistic_payout()` function is publicly accessible, an attacker can repeatedly invoke it with an invalid signature. This can lead to a denial-of-service (DoS) attack by overwhelming the aggregator and verifiers, consuming significant resources and potentially blocking legitimate withdrawal requests.

Recommendations

Implement early signature validation at the beginning of the `optimistic_payout()` function using the `SECP.verify_schnorr()` method.

Resolution

Signature verification has been added at the beginning of the `optimistic_payout()` function in commit 0baaa99. This ensures that the provided `input_signature` is valid before proceeding with any resource-intensive operations for preparing a multisig and making RPC calls to verifiers.

CMT-24	Overly Broad Error Handling in Citrea RPC Calls May Lead to Missed Onchain Data		
Asset	core/src/citrea.rs, core/src/verifier.rs		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The functions `collect_deposit_move_txids()` and `collect_withdrawal_utxos()` in `citrea.rs` are susceptible to prematurely terminating, which can lead to missed deposit and withdrawal data from the `Bridge.sol` contract. This occurs because a transient RPC error is incorrectly treated as the end of the data arrays being read from contract storage.

These functions iterate through the `depositTxIds` and `withdrawalUTXOs` public arrays in `Bridge.sol` to collect all deposit and withdrawal data. They are designed to stop when an RPC call reverts due to an index-out-of-bounds error, which is the expected way to determine that all items have been collected. However, the current implementation checks for any error using `.is_err()` to break the loop. This means that any RPC error, not just an "index out of bounds" revert, will cause the loop to terminate.

core/src/citrea.rs::collect_deposit_move_txids()

```
async fn collect_deposit_move_txids(
    &self,
    last_deposit_idx: Option<u32>,
    to_height: u64,
) -> Result<Vec<(u64, Txid)>, BridgeError> {
    // ...
    loop {
        let deposit_txid = self
            .contract
            .depositTxIds(U256::from(start_idx))
            .block(BlockId::Number(BlockNumberOrTag::Number(to_height)))
            .call()
            .await;
        // @audit This check is too broad. Any RPC error will cause the loop to break.
        // @audit It should specifically check for an out-of-bounds error.
        if deposit_txid.is_err() {
            tracing::trace!(
                "Deposit txid not found for index, error: {:?}",
                deposit_txid
            );
            break;
        }
        // ...
    }
    Ok(move_txids)
}
```

If an intermittent network issue causes the RPC call to fail, the verifier will silently stop collecting data for the current L2 block range. This results in missing entries for `move_txid` or `withdrawal_utxo` in the verifier's local database. Consequently, if a user requests an optimistic payout for a withdrawal corresponding to the missed data, the affected verifier will be unable to process the request. The `sign_optimistic_payout()` function will fail because its initial database checks for the `move_txid` and `withdrawal_utxo` will not find the required records.

While this degrades the user experience by preventing an optimistic payout through that specific verifier, the user can still rely on the standard operator-driven payout mechanism.

The verifier will also pick up the missed entries during the next L1 block's processing cycle, but the `withdrawal_batch_proof_bitcoin_block_height` associated with any missed withdrawal will be recorded with the later L1 block's height. While this field is not currently used elsewhere in the codebase, storing incorrect data could lead to latent bugs.

This issue is classified as low impact and low likelihood. The impact is low because the system is designed to recover from missed data in subsequent blocks, and the only long-term data inconsistency is in a field that is not currently utilized. The primary user-facing impact is a temporary inability to use the optimistic payout feature with an affected verifier. The likelihood is low as it requires a specific, non-terminal RPC failure to occur during the data collection process.

Recommendations

The error handling in `collect_deposit_move_txids()` and `collect_withdrawal_utxos()` should be updated to be more specific. Instead of breaking the loop on any error, the code should inspect the returned error and only break if it's an expected "index out of bounds" revert from the contract.

Other errors, such as transient network failures, should be handled differently, for example, by retrying the call or propagating the error to a higher-level error handling mechanism so that the entire block processing can be retried.

Resolution

The error handling for deposit transaction retrieval in `CitreaClient` has been improved in commit 37f3cf4. The functions now properly distinguish between expected index not found errors and transient RPC errors that should not terminate data collection loops.

CMT-25	`is_deposit_valid()` Does Not Check For Extraneous Operators In Deposit Data		
Asset	`core/src/verifier.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `is_deposit_valid()` function in `verifier.rs` incorrectly validates the operator set for a new deposit. It allows a deposit to be considered valid even if it includes unknown or invalid operators, so long as it also includes all known valid operators. This could allow a malicious aggregator to create deposits with a partially invalid operator set, potentially leading to griefing attacks where funds are temporarily locked.

The current implementation in `is_deposit_valid()` retrieves all operators from the verifier's database and iterates through them. For each known operator with usable collateral, it checks if they are included in the `deposit_data`. However, it does not perform the inverse check: verifying that every operator in `deposit_data` is known and has usable collateral.

core/src/verifier.rs::is_deposit_valid()

```
async fn is_deposit_valid(&self, deposit_data: &mut DepositData) -> Result {
    // ...
    let operator_xonly_pks = deposit_data.get_operators();
    // check if all operators that still have collateral are in the deposit
    let are_all_operators_in_deposit = self.db.get_operators(None).await?;
    // @audit The loop below only checks that all known operators with usable collateral are in the deposit.
    // @audit It does not check if the deposit contains extra, unknown operators.
    for (xonly_pk, reimburse_addr, collateral_funding_outpoint) in are_all_operators_in_deposit
    {
        // ...
        let is_collateral_usable = self
            .rpc
            .collateral_check(
                &operator_data,
                &kickoff_wpks,
                self.config.protocol_paramset(),
            )
            .await?;
        // if operator is not in deposit but its collateral is still on chain, return false
        if !operator_xonly_pks.contains(&xonly_pk) && is_collateral_usable {
            tracing::warn!(
                "Operator {:?} is is still in protocol but not in the deposit",
                xonly_pk
            );
            return Ok(false);
        }
        // if operator is in deposit, but the collateral is not usable, return false
        if operator_xonly_pks.contains(&xonly_pk) && !is_collateral_usable {
            tracing::warn!(
                "Operator {:?} is in the deposit but its collateral is spent, operator cannot fulfill withdrawals anymore",
                xonly_pk
            );
            return Ok(false);
        }
    }
    // ...
}
```

This issue is classified as low impact because the invalid deposit is later rejected by verifiers in `deposit_finalize()`

inside `create_operator_sighash_stream()` as the operator does not exist in the verifier's database, though there is still wasted computation. The likelihood is low because it requires a malicious aggregator to specifically craft the invalid deposit data.

Recommendations

To address this, `is_deposit_valid()` should be modified to ensure that the set of operators provided in `deposit_data` exactly matches the set of known operators with usable collateral in the verifier's database.

Resolution

The `is_deposit_valid()` function has been enhanced to properly validate that all operators in deposit data exist in the verifier's database in commit 8e81bbc. The `set_operator_keys()` function now validates deposits before database writes, and the `DepositInvalid` error has been improved to include the reason for invalidity, providing better error reporting.

CMT-26	Winternitz Key Derivation Uses Simple Concatenation Instead Of KDF		
Asset	`core/src/actor.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `get_derived_winternitz_sk()` function in the Actor implementation uses simple concatenation to derive Winternitz secret keys from a root key and derivation path, rather than employing a cryptographically proper key derivation function (KDF).

The function combines the root Winternitz secret key with the derivation path using basic concatenation on lines [**267-271**]:

core/src/actor.rs::get_derived_winternitz_sk()

```
pub fn get_derived_winternitz_sk(
    &self,
    path: WinternitzDerivationPath,
) -> Result<winternitz::SecretKey, BridgeError> {
    let wsk = self
        .winternitz_secret_key
        .ok_or_eyre("Root Winternitz secret key is not provided in configuration file")?;
    Ok([wsk.as_ref().to_vec(), path.to_bytes()].concat())
}
```

While this approach is not critically vulnerable due to the high entropy of the 32-byte root key and the adequate differentiation provided by the derivation paths, it represents suboptimal cryptographic hygiene. Standard cryptographic practice recommends using proper key derivation functions such as HKDF, PBKDF2, or similar constructions that provide additional security properties including domain separation and resistance to related-key attacks.

The current implementation is functional given the strong base entropy, but does not follow cryptographic best practices for key derivation.

Recommendations

Replace the simple concatenation with a proper key derivation function to improve cryptographic robustness:

```rust
use hkdf::Hkdf;
use sha2::Sha256;

pub fn get_derived_winternitz_sk(
    &self,
    path: WinternitzDerivationPath,
) -> Result<winternitz::SecretKey, BridgeError> {
    let wsk = self
        .winternitz_secret_key
        .ok_or_eyre("Root Winternitz secret key is not provided in configuration file")?;

    let hk = Hkdf::<Sha256>::new(None, wsk.as_ref());
    let path_bytes = path.to_bytes();
    let mut derived_key = vec![0u8; wsk.len()];

    hk.expand(&path_bytes, &mut derived_key)
        .map_err(|_| BridgeError::CryptographicError("Key derivation failed".to_string()))?;

    Ok(derived_key)
}
```

This approach provides better domain separation and follows established cryptographic standards for key derivation.

Resolution

The Winternitz key derivation has been upgraded to use HKDF (HMAC-based Key Derivation Function) in commit 41ab06f. The implementation now uses HKDF-SHA256 with proper domain separation via the derivation path, replacing the previous simple concatenation method. This follows RFC 5869 standards and provides better cryptographic security properties.

CMT-27	Missing Rate Limiting Protection On gRPC API Endpoints		
Asset	`core/src/servers.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The gRPC server implementations for Verifier, Operator, and Aggregator services lack rate limiting protections, potentially allowing malicious clients to overwhelm the services with excessive requests leading to denial of service conditions.

The `create_grpc_server()` function in `core/src/servers.rs` configures tonic gRPC servers with TLS and certificate-based authentication but does not implement any request rate limiting or throttling mechanisms:

core/src/servers.rs::create_grpc_server()

```
let server_builder = tonic::transport::Server::builder()
    .layer(AddMethodMiddlewareLayer)
    .tls_config(tls_config)?
    .add_service(service);
```

Without rate limiting, authenticated clients (or compromised certificates) could potentially:

- Flood the server with legitimate but excessive requests
- Exhaust server resources through request volume attacks
- Impact service availability for other legitimate users
- Cause performance degradation during high-load scenarios

While certificate-based authentication provides some access control, it does not protect against abuse from authenticated clients or scenarios where certificates are compromised.

Recommendations

Implement rate limiting middleware for gRPC endpoints to protect against request flooding:

```
use tower::limit::RateLimitLayer;
use std::time::Duration;

let server_builder = tonic::transport::Server::builder()
    .layer(AddMethodMiddlewareLayer)
    .layer(RateLimitLayer::new(
        100, // max 100 requests
        Duration::from_secs(60) // per minute
    ))
    .tls_config(tls_config)?
    .add_service(service);
```

Additionally, consider implementing:

- Per-client rate limiting based on certificate identity

- Adaptive rate limiting based on server load
- Request size limits to prevent oversized message attacks
- Connection limits to prevent connection exhaustion
- Monitoring and alerting for rate limit violations

The specific rate limits should be configured based on expected legitimate usage patterns and server capacity.

Resolution

Rate limiting middleware has been added in commit 1ab5803.

CMT-28	Challenge Fee Recovery Mechanism Not Implemented
Asset	`core/src/builder/transaction/challenge.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The challenge system currently lacks an on-chain fee recovery mechanism, which is a known design limitation of the current implementation. This creates a temporary economic vulnerability where operators could potentially extract challenge fees without proper recovery mechanisms.

The challenge transaction creates an unconditional 1 BTC payment to the operator with no escrow or conditional recovery mechanism in the current Bitcoin layer implementation. When a verifier challenges a malicious kickoff, the verifier funds the challenge with 1 BTC (paid from their wallet via RBF funding) while the operator immediately receives 1 BTC unconditionally. No on-chain mechanism exists to recover the 1 BTC if the challenge is proven valid.

However, this is a known design limitation that will be addressed through a smart contract implementation on the Citrea layer 2 network. The intended design requires operators to deposit 1 BTC into a smart contract on Citrea, which will automatically transfer the funds to the challenger if a challenge is proven successful, providing the necessary economic guarantees for the challenge system.

Additionally, an operator can avoid collateral slashing by spending their burn connector output to themselves, as described in CMT-04. Since the burn connector (worth approximately 2 BTC) is controlled by the operator's key, malicious operators can submit a malformed kickoff, receive the 1 BTC challenge fee, then spend their burn connector to their own address before being slashed. This results in a net profit of 1 BTC (challenge fee) plus 2 BTC of saved collateral.

core/src/builder/transaction/challenge.rs::create_challenge_txhandler()

```
pub fn create_challenge_txhandler(
    kickoff_txhandler: &TxHandler,
    operator_reimbursement_address: &bitcoin::Address,
    paramset: &'static ProtocolParamset,
) -> Result {
    Ok(TxHandlerBuilder::new(TransactionType::Challenge)
        .with_version(Version::non_standard(3))
        .add_input(
            NormalSignatureKind::Challenge,
            kickoff_txhandler.get_spendable_output(UtxoVout::Challenge)?,
            SpendPath::ScriptSpend(0),
            DEFAULT_SEQUENCE,
        )
        .add_output(UnspentTxOut::from_partial(TxOut {
            value: paramset.operator_challenge_amount,
            script_pubkey: operator_reimbursement_address.script_pubkey(),
        }))
        .finalize())
}
```

Furthermore, the comments state that `challenger_evm_address` is an argument to this function. However, it is not supplied and there are no script paths with contain `OP_RETURN` for the challenger to supply their EVM refund address. It is assumed here that the challenger will be adding their EVM address in a separate output.

core/src/builder/transaction/challenge.rs::create_challenge_txhandler()

```
/// # Arguments
///
/// * `kickoff_txhandler` - The kickoff transaction handler that the challenge belongs to.
/// * `operator_reimbursement_address` - The address to reimburse the operator to cover their costs.
/// * `challenger_evm_address` - The EVM address of the challenger, for reimbursement if the challenge is correct.
/// * `paramset` - Protocol parameter set.
///
```

Recommendations

The team should prioritise implementing the planned Citrea layer 2 smart contract solution that will provide proper challenge fee recovery mechanisms. This smart contract should:

- Require operators to deposit 1 BTC before participating in the challenge system
- Automatically transfer the deposited funds to successful challengers
- Provide clear economic incentives for honest behaviour

Until this mechanism is implemented, operators should be aware of the current design limitation and the temporary economic vulnerability it creates.

A complete challenger identification mechanism should also be implemented if it is not already present in the challenger output and funding processes.

Resolution

Challenge fee recovery has been implemented on the Bitcoin layer in commit 2968856.

An additional output is now included in the challenge transaction to include the EVM address and allow reimbursement of the challenger. The additional output will append the address of whichever user creates the challenge transaction.

When an operator creates a challenge, they will now include their EVM address, however, it is not relevant as they do not sign over the `OP_RETURN` output, only the first input and output.

Therefore, a challenger can still create a challenge transaction and replace the `OP_RETURN` output to include their EVM address.

CMT-29	Hardcoded Block Limit In `get_logs()` Should Be Configurable
Asset	`core/src/citrea.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `get_logs()` function contains a hardcoded block limit of 999 that prevents the system from adapting to different RPC provider limitations, potentially causing failed requests or suboptimal performance.

The function fetches logs in chunks to work around RPC provider limits on the number of blocks that can be queried in a single request. However, the chunk size is hardcoded to 999 blocks:

core/src/citrea.rs::get_logs()

```
// Block num is 999 because limits are inclusive.
let to_height = std::cmp::min(from_height + 999, to_height);
```

Different RPC providers have varying limits for log queries. Some providers may allow more than 1000 blocks per request, while others may have stricter limits (e.g., 100 or 500 blocks). The hardcoded value prevents the system from optimizing for the specific RPC provider being used and may cause requests to fail if the provider's limit is lower than 999 blocks.

Recommendations

Make the block limit configurable by adding it as a parameter to the `CitreaClient` struct or as a configuration option.

Resolution

The issue has been closed as will be fixed later. The hardcoded block limit is currently consistent with Citrea's implementation and will be made configurable in a future update when the underlying infrastructure supports it.

CMT-30	Outstanding `TODO` Comments Throughout Codebase
Asset	/*
Status	**Closed:** See Resolution
Rating	Informational

Description

Multiple `TODO` comments have been identified throughout the codebase indicating incomplete features, potential issues, or areas requiring future development. These comments represent technical debt and may indicate areas where additional implementation, testing, or verification is needed.

High Priority `TODO` s

- Security/Verification Issues:
  - `core/src/verifier.rs:1201` : "Use correct verification key and along with a dummy proof" - suggests dummy proof usage
  - `core/src/deposit.rs:325` : "remove this impl, this is done to avoid checking the address" - bypassing address validation
  - `core/src/builder/transaction/creator.rs:639` : "Extract directly from round tx - not safe" - unsafe implementation noted
- Performance/Blocking Operations:
  - `core/src/rpc/aggregator.rs:139, 176, 354` : Multiple "consider spawn_blocking here" comments indicating potential blocking operations in async contexts
- Error Handling:
  - `core/src/tx_sender/rbf.rs:462, 471` : "handle errors here and update the state" and "print better msg and update state" - incomplete error handling
  - `core/src/tx_sender/cpfp.rs:526` : "implement txid checking so we can save the correct error"

Implementation/Feature `TODO` s

- Fee Estimation:
  - `core/src/tx_sender/mod.rs:187, 192, 196` : Multiple weight estimation refinements needed
  - `core/src/tx_sender/cpfp.rs:114` : "Ensure all fee payer UTXO inputs are correctly signed"
- BitVM Integration:
  - `core/src/bitvm_client.rs:449` : "this might be wrong, add clementine specific ones too"
  - `core/src/builder/transaction/creator.rs:1052` : "add when we add actual disprove scripts"
- Database/State Management:

  - `core/src/database/operator.rs:827` : "check if AND ds.round_idx >= cr.round_idx is correct"
  - `core/src/verifier.rs:966` : "It can create problems if the deposit fails at the end"

Test/Development `TODO` s

- `core/src/builder/address.rs:312` : Test marked as ignored with "Investigate this"
- `core/src/builder/address.rs:350` : "check this later" for address validation
- Multiple test-related TODOs indicating incomplete test coverage

Quick `TODO` s

- `core/src/citrea.rs:118` : "This is not the best way to do this, but it's a quick fix for now"
- `core/src/tx_sender/rbf.rs:938` : Placeholder "TODO" in OP_RETURN output

Some `TODO` s indicate fundamental architectural decisions that need resolution, while others represent minor implementation details that should be addressed for code quality and maintainability.

Recommendations

Prioritise and address the outstanding `TODO` comments, particularly those related to security, verification, and error handling.

Resolution

The issue has been acknowledged and will be addressed in future updates. The team will systematically review and resolve the outstanding TODO comments, prioritising those related to security, verification, and error handling.

CMT-31	`set_operator_keys()` Does Not Validate Deposit Before Database Write
Asset	`core/src/verifier.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `set_operator_keys()` function writes deposit data directly to the database without validating the deposit first, potentially allowing invalid deposits to be stored and processed.

The function calls `self.db.set_deposit_data()` immediately after receiving the deposit data, without performing any validation checks. This is inconsistent with other deposit-related functions in the same file that properly validate deposits before database operations.

core/src/verifier.rs::set_operator_keys()

```rust
pub async fn set_operator_keys(
    &self,
    mut deposit_data: DepositData,
    keys: OperatorKeys,
    operator_xonly_pk: XOnlyPublicKey,
) -> Result<(), BridgeError> {
    // @audit Missing is_deposit_valid() check
    self.db
        .set_deposit_data(None, &mut deposit_data, self.config.protocol_paramset())
        .await?;
    // ...
}
```

The `is_deposit_valid()` function performs critical validation checks including:

- Verifying the security council matches the configuration
- Ensuring all active operators are included in the deposit
- Validating the deposit script and amount
- Confirming the deposit transaction exists onchain

Without this validation, the `set_operator_keys()` function may accept and store invalid deposits that could lead to incorrect business logic execution or processing of malformed deposit data.

This issue is rated as informational as it was discovered by the client project team during the engagement of the review. In practice, this is a low impact issue as it could result in a temporary denial of service if the `deposit_data` stored is incorrect for the `deposit_outpoint`. The likelihood is medium as it requires a malicious operator to provide invalid deposit data.

Recommendations

Add deposit validation before writing to the database in the `set_operator_keys()` function.

Resolution

The issue has been resolved in commit 8e81bbc by adding `check_nofn_correctness()` and `is_deposit_valid()` checks before writing to the database in the `set_operator_keys()` function.

CMT-32	Lack Of Validation For Protocol Parameters
Asset	core/src/config/protocol.rs
Status	**Closed:** See Resolution
Rating	Informational

Description

The `ProtocolParamset` struct can be deserialized from a TOML file or environment variables without subsequent validation of its parameters. This could lead to a misconfiguration where actors operate with incompatible parameters, causing transaction failures.

Specifically, the `from_toml_file()` and `from_env()` methods do not enforce protocol invariants. For instance, `winternitz_log_d` must equal the global `WINTERNITZ_LOG_D` constant, and `num_signed_kickoffs` must not be greater than `num_kickoffs_per_round`. A misconfiguration of these values would not be caught, potentially disrupting bridge operations.

core/src/config/protocol.rs::from_toml_file()

```rust
pub fn from_toml_file(path: &Path) -> Result {
    let contents = fs::read_to_string(path).wrap_err("Failed to read config file")?;

    let paramset: Self = toml::from_str(&contents).wrap_err("Failed to parse TOML")?;

    // @audit The paramset should be verified here.
    Ok(paramset)
}
```

This issue is classified as informational because it stems from a potential misconfiguration rather than a flaw in the protocol logic itself. It does not directly risk user funds but affects the ease of use and reliability of the system.

Recommendations

It is recommended to add a `verify()` method to `ProtocolParamset` that is called after deserialization in both `from_toml_file()` and `from_env()`. This method should check for parameter invariants to prevent misconfigurations.

Resolution

The issue has been closed as not requiring immediate fix. Since the first deposit sign operation would fail if protocol parameters are misconfigured, this provides an implicit validation mechanism. Additional explicit validation may be added in future updates.

CMT-33	Unchecked Arithmetic Operations In Round Transaction Value Calculation
Asset	core/src/builder/transaction/operator_collateral.rs
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `create_round_txhandler()` function performs unchecked arithmetic operations when calculating the output value for round transactions, which could potentially lead to integer underflow panics or unexpected behavior if input values are malformed.

The function calculates the remaining value after subtracting kickoff amounts and anchor amounts from the input amount using standard arithmetic operators:

```
builder = builder.add_output(UnspentTxOut::from_scripts(
    input_amount
        - (paramset.kickoff_amount + paramset.default_utxo_amount())
            * (paramset.num_kickoffs_per_round as u64)
        - paramset.anchor_amount(),
    vec![],
    Some(operator_xonly_pk),
    paramset.network,
));
```

This calculation on lines [**104-107**] in `core/src/builder/transaction/operator_collateral.rs` involves multiple arithmetic operations without explicit overflow/underflow checks. If the sum of the below operation exceeds `input_amount`, the subtraction will underflow:

```
(paramset.kickoff_amount + paramset.default_utxo_amount()) * (paramset.num_kickoffs_per_round as u64) + paramset.anchor_amount()
```

While Bitcoin `Amount` types may have some built-in protections, using unchecked arithmetic operations can still lead to panics in debug builds or wraparound behavior in release builds, potentially causing the transaction builder to create invalid transactions or crash the operator.

Recommendations

Replace the unchecked arithmetic operations with checked arithmetic methods to provide explicit error handling:

```
let total_required = (paramset.kickoff_amount + paramset.default_utxo_amount())
    .checked_mul(paramset.num_kickoffs_per_round as u64)
    .and_then(|kickoff_total| kickoff_total.checked_add(paramset.anchor_amount()))
    .ok_or_else(|| BridgeError::ArithmeticOverflow("Total required amount calculation overflow"))?;

let remaining_amount = input_amount
    .checked_sub(total_required)
    .ok_or_else(|| BridgeError::InsufficientFunds("Input amount insufficient for required outputs"))?;

builder = builder.add_output(UnspentTxOut::from_scripts(
    remaining_amount,
    vec![],
    Some(operator_xonly_pk),
    paramset.network,
));
```

This approach provides explicit validation of arithmetic operations and clear error messages when calculations fail.

Resolution

The issue has been resolved in commit c848a38 by replacing unchecked arithmetic operations with checked methods, ensuring proper error handling for potential overflows and underflows.

CMT-34	Withdrawal UTXOs Has Incorrect Endianness
Asset	core/src/citrea.rs
Status	**Resolved:** See Resolution
Rating	Informational

Description

There is an endianness inconsistency when deserializing the output index (`vout`) from withdrawal UTXOs fetched from the Citrea bridge contract. The function `withdrawal_utxos()` incorrectly interprets the `bytes4` value as big-endian, while other parts of the codebase, such as `collect_withdrawal_utxos()`, correctly treat it as little-endian.

The `withdrawal_utxos()` function in `core/src/citrea.rs` uses `u32::from_be_bytes()` to decode the `outputId` field. This assumes the bytes are in big-endian order.

core/src/citrea.rs::withdrawal_utxos()

```
async fn withdrawal_utxos(&self, withdrawal_index: u64) -> Result<OutPoint, BridgeError> {
    // ...
    let vout = withdrawal_utxo.outputId.0;
    // @audit This should be little-endian
    let vout = u32::from_be_bytes(vout);

    Ok(OutPoint { txid, vout })
}
```

However, Bitcoin transaction data, including the `vout`, is encoded in little-endian format. The correct implementation is found in the `collect_withdrawal_utxos()` function within the same file, which uses `u32::from_le_bytes()`. The data is stored on the EVM chain as a `bytes4`, but the byte ordering respects the native format of the Bitcoin protocol.

core/src/citrea.rs::collect_withdrawal_utxos()

```
async fn collect_withdrawal_utxos(
    &self,
    last_withdrawal_idx: Option<u32>,
    to_height: u64,
) -> Result<Vec<(u64, OutPoint)>, BridgeError> {
    // ...
        let vout = withdrawal_utxo.outputId.0;
        let vout = u32::from_le_bytes(vout);
        let utxo = OutPoint { txid, vout };
    // ...
}
```

This issue is classified as informational because the `withdrawal_utxos()` function is not currently called from any production code. Only the correctly implemented `collect_withdrawal_utxos()` function is in use. While this function is currently unused in the production codebase, it could lead to incorrect behavior if it were to be used in the future.

Recommendations

To prevent accidental use of the incorrect function and to maintain a clean codebase, it is recommended to either remove the `withdrawal_utxos()` function or correct its implementation to use little-endian decoding.

To correct the implementation, change the following line in `core/src/citrea.rs`:

core/src/citrea.rs::withdrawal_utxos()

```
let vout = u32::from_be_bytes(vout);
```

to:

core/src/citrea.rs::withdrawal_utxos()

```
let vout = u32::from_le_bytes(vout);
```

Resolution

The issue has been resolved in commit 4082bf5 by removing the unused `withdrawal_utxos()` function.

CMT-35	Miscellaneous General Comments
Asset	/*
Status	**Closed:** See Resolution
Rating	Informational

Description

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

1. **Redundant Taproot Address Calculation In `create_taproot_address` Function**

   Related Asset(s): address.rs

   The `create_taproot_address()` function performs redundant address calculation by manually calling `Address::p2tr()` after already obtaining the finalised taproot information from `taproot_builder.finalize()`.

   The `finalize()` method returns a `TaprootSpendInfo` struct that already contains the tweaked output key in its `output_key` field. This tweaked key represents the final taproot address and can be used directly to create the address, eliminating the need for the manual `Address::p2tr()` calculation that duplicates the same tweaking process.

   core/src/builder/address.rs::create_taproot_address()

   ```
   // @audit Redundant calculation - finalize() already computes the tweaked key
   let taproot_address = match internal_key {
       Some(xonly_pk) => Address::p2tr(&SECP, xonly_pk, tree_info.merkle_root(), network),
       None => Address::p2tr(
           &SECP,
           *bitvm_client::UNSPENDABLE_XONLY_PUBKEY,
           tree_info.merkle_root(),
           network,
       ),
   };
   ```

   The `TaprootSpendInfo` struct contains an `output_key` field of type `TweakedPublicKey` which is the result of tweaking the internal key with the merkle root. The `Address::p2tr()` function performs the same tweaking operation internally, making this calculation redundant.

   Replace the redundant `Address::p2tr()` calls with direct address creation from the `output_key` field of the `TaprootSpendInfo` returned by `finalize()`. This can be done by converting the `TweakedPublicKey` to an address directly.

2. **Incorrect Variable Used In Debug Log For Operator Signature Verification**

   Related Asset(s): core/src/verifier.rs

   The debug log at line [827] in the operator signature verification section incorrectly uses `nonce_idx + 1` to display the operator signature number, when it should use `op_sig_count + 1`.

core/src/verifier.rs::deposit_finalize()

```
// @audit: Incorrect usage of nonce_idx + 1 for operator signatures
tracing::debug!(
    "Verifying Final operator signature {} for operator {}, signature info {:?}",
    nonce_idx + 1,  // @audit: Should be op_sig_count + 1
    operator_idx,
    typed_sighash.1
);
```

Replace `nonce_idx + 1` with `op_sig_count + 1` in the debug log at line [**827**]:

3. **Spelling & Grammar**

   Related Asset(s): core/src/operator.rs, core/src/builder/address.rs, core/src/database/operator.rs, core/src/test/deposit_and_withdraw_e2e.rs

   Multiple spelling and grammatical errors have been identified throughout the codebase in comments, documentation, and variable names that should be corrected for code quality and professionalism.

   Spelling Errors:

   - `witdhrawal` should be `withdrawal` in `core/src/operator.rs:509`
   - Incomplete comment `operator_idx: 0, // dummy value, doesn't` in `core/src/operator.rs:739` - remove `, doesn't`
   - `existant` should be `existent` in multiple locations in `core/src/database/operator.rs` (lines [**1033, 1066, 1118, 1138, 1146, 1200, 1240**])

   Grammatical Errors:

   - `Funds can be spend` should be `Funds can be spent` in `core/src/builder/address.rs:127`
   - `dont` should be `don't` in `core/src/test/deposit_and_withdraw_e2e.rs:1306`

   It is recommended to fix these minor spelling mistakes.

4. **Hardcoded Transaction Version In Multiple Locations**

   Related Asset(s): core/src/builder/transaction/operator_collateral.rs

   Throughout the codebase, transaction handlers are consistently created with a hardcoded version value of 3 using `Version::non_standard(3)`. This pattern appears in multiple transaction creation functions across the operator collateral module.

```
// Multiple instances of hardcoded version 3:

// Round transaction
TxHandlerBuilder::new(TransactionType::Round).with_version(Version::non_standard(3))

// Ready-to-reimburse transaction
TxHandlerBuilder::new(TransactionType::ReadyToReimburse).with_version(Version::non_standard(3))

// Assert timeout transactions
TxHandlerBuilder::new(TransactionType::AssertTimeout(idx)).with_version(Version::non_standard(3))

// Unspent kickoff transactions
TxHandlerBuilder::new(TransactionType::UnspentKickoff(idx)).with_version(Version::non_standard(3))
```

   Consider defining a constant or configuration parameter for the transaction version to centralise this value and make future changes easier.

5. **Unnecessary Clone of `ReimburseDbCache`**

   Related Asset(s): core/src/builder/transaction/creator.rs

   In the `create_txhandlers()` function, the entire `ReimburseDbCache` struct is cloned only to destructure and access the `paramset` field.

core/src/builder/transaction/creator.rs::create_txhandlers()

```
let ReimburseDbCache { paramset, .. } = db_cache.clone(); // @audit inefficient clone
```

`paramset` is a `&'static` reference, it can be copied cheaply without cloning the parent struct. It is recommended to access the field directly.

core/src/builder/transaction/creator.rs::create_txhandlers()

```
let paramset = db_cache.paramset;
```

6. **Unused RNG Parameter in MuSig2 Nonce Generation**

   Related Asset(s): Contract.sol

   The `nonce_pair()` function in `core/src/musig2.rs` accepts an `rng` parameter but doesn't actually use it for nonce generation:

   core/src/musig2.rs::nonce_pair()

```
pub fn nonce_pair(
    keypair: &Keypair,
    mut rng: &mut impl Rng,  // Parameter accepted but unused
) -> Result<(SecretNonce, PublicNonce), BridgeError> {
    let musig_session_sec_rand = SessionSecretRand::new();  // @audit Uses internal randomness

    Ok(new_nonce_pair(
        SECP256K1,
        musig_session_sec_rand,  // Uses this instead of rng parameter
        None,
        None,
        to_secp_kp(keypair).public_key(),
        None,
        None,
    ))
}
```

   Either utilise the provided RNG parameter by calling `SessionSecretRand::from_rng()` or remove it from the function signature to clarify the randomness source.

7. **Redundant Partial Signature Aggregation in MuSig2 Implementation**

   Related Asset(s): core/src/musig2.rs

   The `aggregate_partial_signatures()` function in `core/src/musig2.rs` calls `session.partial_sig_agg(&partial_sigs)` twice:

   core/src/musig2.rs::aggregate_partial_signatures()

```
let final_sig = session.partial_sig_agg(&partial_sigs);  // Line 177

// ... verification code ...

Ok(from_secp_sig(
    session.partial_sig_agg(&partial_sigs).assume_valid(),  // Line 188 - redundant call
))
```

   Refactor to perform signature aggregation only once and reuse the result.

8. **Replace `expect()` With Graceful Error Handling In `sign_optimistic_payout()`**

   Related Asset(s): core/src/verifier.rs

   In `sign_optimistic_payout()`, the use of `expect()` can be avoided by adopting a chained method pattern with early unwrapping and proper error handling.

core/src/verifier.rs::sign_optimistic_payout()

```
// check if withdrawal is valid first
let move_txid = self
   .db
   .get_move_to_vault_txid_from_citrea_deposit(None, deposit_id)
   .await?;
if move_txid.is_none() {
   return Err(eyre::eyre!("Deposit not found for id: {}", deposit_id).into());
}

// ...later in the code...
let move_txid = move_txid.expect("Withdrawal must be Some");
```

Update the code as shown below.

core/src/verifier.rs::sign_optimistic_payout()

```
// More idiomatic approach
let move_txid = self
    .db
    .get_move_to_vault_txid_from_citrea_deposit(None, deposit_id)
    .await?
    .ok_or_else(|| eyre::eyre!("Deposit not found for id: {}", deposit_id).into())?;
```

9. **Unused `operator_clients` Variable**

   Related Asset(s): core/src/rpc/aggregator.rs

   The `operator_clients` variable declared in the `new_deposit()` function is never used. Instead, the `operators` variable is directly passed to the `collect_operator_sigs()` function. Removing the unused variable would improve code clarity.

   Remove the unused `operator_clients` variable.

10. **Incorrect Async Usage**

    Related Asset(s): core/src/rpc/parser/operator.rs

    The `parse_withdrawal_sig_params()` function does not perform any asynchronous operations. It only parses and converts fields from a struct synchronously. Removing the `async` keyword would improve clarity and readability.

    Removing the `async` keyword.

11. **Incorrect Error Message In `parse_schnorr_sig()`**

    Related Asset(s): core/src/rpc/parser/operator.rs

    When the expected signature type is not received from the stream, the `parse_schnorr_sig()` function returns the error `expected_msg_got_none("WinternitzPubkeys")`, which is misleading and incorrect.

    Return `expected_msg_got_none("UnspentKickoffSig")` to accurately reflect the expected signature type.

12. **Redundant `serialize()` Call In Loop Causes Inefficiency**

    Related Asset(s): core/src/rpc/aggregator.rs

    In the `optimistic_payout()` function, the `serialize()` method is currently being called on `agg_nonce` inside a loop, which is inefficient. This call can be moved outside the loop, immediately after `agg_nonce` is generated.

    Move the call to the `serialize()` function on `agg_nonce` outside the loop and invoke it immediately after `agg_nonce` is generated.

Recommendations

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

Resolution

The miscellaneous issues have been acknowledged and will be addressed in future updates. These code quality improvements, while not security-critical, will enhance the maintainability and clarity of the codebase.

CMT-36	Direct Panic Vulnerabilities In Verifier RPC Endpoints		
Asset	`core/src/rpc/verifier.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The verifier's `deposit_finalize()` RPC endpoint contains direct `panic!()` calls that can be triggered by a malicious aggregator, leading to denial-of-service attacks against verifier nodes during the deposit finalisation process.

The `deposit_finalize()` function expects to receive a specific number of signatures from the aggregator based on the deposit configuration. However, if the aggregator sends fewer signatures than expected, the verifier will panic and crash instead of returning a proper gRPC error response.

Two separate panic conditions exist in the signature parsing logic:

core/src/rpc/verifier.rs::deposit_finalize()

```
tokio::spawn(async move {
    let num_required_nofn_sigs = verifier.config.get_num_required_nofn_sigs(&deposit_data);
    let mut nonce_idx = 0;

    while let Some(sig) = parse_next_deposit_finalize_param_schnorr_sig(&mut in_stream).await? {
        // ... process signature
        nonce_idx += 1;
        if nonce_idx == num_required_nofn_sigs {
            break;
        }
    }

    // @audit First panic: insufficient N-of-N signatures
    if nonce_idx < num_required_nofn_sigs {
        panic!(
            "Expected more nofn sigs {} < {}",
            nonce_idx, num_required_nofn_sigs
        )
    }

    // ... operator signature parsing loop

    // @audit Second panic: insufficient operator signatures
    if total_op_sig_count < num_required_total_op_sigs {
        panic!(
            "Not enough operator signatures. Needed: {}, received: {}",
            num_required_total_op_sigs, total_op_sig_count
        );
    }
});
```

A malicious aggregator can exploit this by calling `deposit_finalize()` and deliberately sending fewer signatures than the verifier expects. This causes the verifier process to panic and terminate, disrupting bridge operations.

The aggregator is centralised and controlled by Citrea, reducing the likelihood of malicious behaviour.

This issue is rated as medium impact because although it enables denial-of-service attacks against verifier nodes, no funds are lost and the bridge can recover once verifier processes are restarted. The likelihood is medium due to the centralised nature of the aggregator, though the technical feasibility remains high.

Recommendations

Replace the direct `panic!()` calls with proper gRPC error handling that returns appropriate status codes to the aggregator:

Recommended fix for core/src/rpc/verifier.rs

```
// Replace first panic
if nonce_idx < num_required_nofn_sigs {
    return Err(Status::invalid_argument(format!(
        "Insufficient N-of-N signatures received: got {}, expected {}",
        nonce_idx, num_required_nofn_sigs
    )).into());
}

// Replace second panic
if total_op_sig_count < num_required_total_op_sigs {
    return Err(Status::invalid_argument(format!(
        "Insufficient operator signatures received: got {}, expected {}",
        total_op_sig_count, num_required_total_op_sigs
    )).into());
}
```

Additionally, consider implementing timeout protection and request validation to prevent malformed or incomplete signature streams from reaching the validation logic.

Resolution

The issue has been resolved by replacing the direct `panic!()` calls with proper error handling that returns gRPC status codes to the aggregator. This ensures that the verifier does not crash and can handle unexpected input gracefully, maintaining bridge availability. Changes are reflected in commit aa76265.

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.

Impact \ Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium

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

σ'