CANTINA

Clementine

Competition

October 29, 2025

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above matrix. High severity findings represent the most critical issues that must be addressed immediately, as they either have high impact and high likelihood of occurrence, or medium impact with high likelihood.

Medium severity findings represent issues that, while not immediately critical, still pose significant risks and should be addressed promptly. These typically involve scenarios with medium impact and medium likelihood, or high impact with low likelihood.

Low severity findings represent issues that, while not posing immediate threats, could potentially cause problems in specific scenarios. These typically involve medium impact with low likelihood, or low impact with medium likelihood.

Lastly, some findings might represent improvements that don't directly impact security but could enhance the codebase's quality, readability, or efficiency (Gas and Informational findings).

2 Security Review Summary

Citrea is the first rollup that enhances the capabilities of Bitcoin blockspace with zero-knowledge technology, making it possible to build everything on Bitcoin.

From Aug 11th to Sep 8th Cantina hosted a competition based on Clementine. The participants identified a total of **117** issues in the following risk categories:

- High Risk: 2
- Medium Risk: 3
- Low Risk: 16
- Gas Optimizations: 0
- Informational: 96

The Cantina researcher n4nika reviewed and confirmed the fixes for the issues found in the competition. These fixes can be found in the following pull requests:

PR 1114	PR 1135	PR 1166	PR 1152	PR 1161	PR 1158	PR 1157	PR 1155	PR 1153	PR 1154
PR 1151	PR 1139	PR 1113	PR 1129	PR 1133	PR 1179	PR 1148	PR 1103	PR 1143	PR 1049
PR 1101	PR 1141	PR 1100	PR 1099	PR 1156	PR 1142	PR 1146	PR 1140	PR 1149	PR 1136
PR 1121	PR 1120	PR 1137	PR 1131	PR 1160	PR 1088	PR 1074	PR 1082	PR 1104	PR 1068
PR 1073	PR 1067	PR 1167	PR 1065	PR 1189	PR 1134	PR 1061	PR 1057	PR 1162	PR 1059
PR 1054	PR 1062								

Additional minor flaws were resolved in PR 1202.

The present report only outlines the **high** and **medium** risk issues.

3 Findings

3.1 High Risk

3.1.1 Incorrect usage of `get_num_verifiers` in `send_operator_asserts_if_ready` breaks state machine

Submitted by n4nika, also found by Rhaydden, kogekar, 0xnija, TheStryke, kogekar and rusiqe

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: Since `send_operator_asserts_if_ready` uses `get_num_verifiers` instead of `get_num_watchtowers` to determine whether the asserts can be sent, this state transition will not work in case the aggregator utilizes the `watchtowers` field of the `deposit_data`, allowing the aggregator to deadlock that state machine.

Finding Description: In `send_operator_asserts_if_ready` it is determined whether or not the operator should send the asserts upon receiving an event. The function incorrectly uses `get_num_verifiers` instead of `get_num_watchtowers`:

```
async fn send_operator_asserts_if_ready(&mut self, context: &mut StateContext<T>) {
    context
        .capture_error(async |context| {
            {
                // if all watchtower challenge utxos are spent and latest blockhash is committed, its safe to
                ↪  send asserts
                if self.spent_watchtower_utxos.len() == self.deposit_data.get_num_verifiers()
                    && self.latest_blockhash != Witness::default()
                {
                    context
                        .owner
                        .handle_duty(Duty::SendOperatorAsserts {
                            kickoff_data: self.kickoff_data,
                            deposit_data: self.deposit_data.clone(),
                            watchtower_challenges: self.watchtower_challenges.clone(),
                            payout_blockhash: self.payout_blockhash.clone(),
                            latest_blockhash: self.latest_blockhash.clone(),
                        })
                        .await?;
                }
                Ok::<(), BridgeError>(())
            }
            .wrap_err(self.kickoff_meta("on send_operator_asserts"))
        })
        .await;
}
```

Going up the call tree, it can be seen that the function is ONLY called here:

```
KickoffEvent::LatestBlockHashSent {
    latest_blockhash_outpoint,
} => {
    let witness = context
        .cache
        .get_witness_of_utxo(latest_blockhash_outpoint)
        .expect("Latest blockhash outpoint that got matched should be in block");
    // save latest blockhash witness
    self.latest_blockhash = witness;
    // can start sending asserts as latest blockhash is committed and finalized
    self.send_operator_asserts_if_ready(context).await;
    self.disprove_if_ready(context).await;
    Handled
}
```

This path is only triggered once the `LatestBlockhash` has been sent. It is very important to note here that the `LatestBlockhash` is only sent AFTER `get_num_watchtowers` watchtower UTXOs have been received:

```
async fn create_matcher_for_latest_blockhash_if_ready(
    &mut self,
    context: &mut StateContext<T>,
) {
```

```
    context
        .capture_error(async |context| {
            {
                // if all watchtower challenge utxos are spent, its safe to send latest blockhash commit tx
                if self.spent_watchtower_utxos.len() == self.deposit_data.get_num_watchtowers()
                {
                    // create a matcher to send latest blockhash tx after finality depth blocks pass from current
                    ↪  block height
                    self.matchers.insert(
                        Matcher::BlockHeight(
                            context.cache.block_height + context.paramset.finality_depth,
                        ),
                        KickoffEvent::TimeToSendLatestBlockhash,
                    );
                }
                Ok::<(), BridgeError>(())
            }
            .wrap_err(self.kickoff_meta("on check_if_time_to_commit_latest_blockhash"))
        })
        .await;
}
```

Since `get_num_watchtowers` is larger than `get_num_verifiers` if the aggregator includes explicit `watchtowers` in a deposit, this will effectively deadlock the state machine.

Impact: This will prevent the operator from sending the asserts, causing them to be slashed once the `AssertTimeout` has been sent.

Recommendation: Consider using `get_num_watchtowers` instead of `get_num_verifiers` here.

3.1.2 Premature sending of disprove transactions allows malicious operator to steal bridge funds

Submitted by n4nika

Severity: High Risk

Context: *(No context files were provided by the reviewer)*

Summary: Due to `disprove_if_ready` not checking whether `operator_challenge_acks` have been received, challengers will try to send the `Disprove` tx without knowing the operator's preimage, causing the transaction to not be included onchain since it's not valid. Since the challenger's state machine only executes the `Disprove` once, this means no disprove will be sent once it would actually be possible to send it, allowing the operator to send the `DisproveTimeout`, getting reimbursed even though they acted maliciously.

Finding Description: `disprove_if_ready` checks if the conditions to send the disprove are met and sends the disprove if so:

```
async fn disprove_if_ready(&mut self, context: &mut StateContext<T>) {
    if self.operator_asserts.len() == ClementineBitVMPublicKeys::number_of_assert_txs()
        && self.latest_blockhash != Witness::default()
        && self.spent_watchtower_utxos.len() == self.deposit_data.get_num_watchtowers()
    {
        self.send_disprove(context).await;
    }
}
```

The problem is that the disprove is sent immediately once the operator received the `latest_blockhash`, all `operator_asserts` and all `watchtower_utxos` have been spent.

In the current system the usual flow looks like this:

- Kickoff gets challenged.
- Watchtowers send challenge transactions.
- Operator sends asserts.
- Operator sends last blockhash.
- *Operator needs to acknowledge the watchtower challenges*.

As we can see, `disprove_if_ready` does NOT wait until it received all necessary challenge acknowledgements which are necessary for the `Disprove`.

Therefore an operator can abuse this behaviour to drain funds from the bridge:

- Send malicious kickoff.
- Kickoff gets challenged.
- Watchtowers send challenges.
- Operator sends asserts and last blockhash.
- Operator *waits* with sending the acknowledgement.
- Since, at this point, all conditions for `disprove_if_ready` are met, the challenger constructs the `Disprove` TX and queues it to be sent onchain.
- Since a challenger will only send the `Disprove` once (once they believe they have all necessary information to do so), they will NOT send it again after the operator sent their acknowledgments.
- Now the operator can send their acknowledgements, preventing getting slashed.
- Now once the `DISPROVE_TIMEOUT_TIMELOCK` expires, the operator can send the `DisproveTimeout` TX, getting reimbursed for a malicious kickoff.

Recommendation: Consider adapting `disprove_if_ready` to wait until the operator received all `OperatorChallengeAck` transactions and only send the `Disprove` afterwards.

3.2 Medium Risk

3.2.1 Incorrect logic in `calculate_bump_feerate` prevents transaction fees from being bumped properly

Submitted by n4nika

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Since there are logic errors in `calculate_bump_feerate`, some RBF transactions may not get a high enough fee set, causing them to not be included in blocks.

Finding Description: Looking at `calculate_bump_feerate` there seem to be two problems:

```
pub async fn calculate_bump_feerate(
    &self,
    txid: &Txid,
    new_feerate: FeeRate,
) -> Result<Option<Amount>> {
    // [...]

    // Conservative vsize calculation
    let original_tx_vsize = original_tx_weight.to_vbytes_floor();
    let original_feerate = original_tx_fee.to_sat() as f64 / original_tx_vsize as f64;

    // Use max of target fee rate and original + incremental rate
    let min_bump_feerate = original_feerate + (222f64 / original_tx_vsize as f64); // [1]

    let effective_feerate_sat_per_vb = std::cmp::max(
        new_feerate.to_sat_per_vb_ceil(),
        min_bump_feerate.ceil() as u64,
    );

    // If original feerate is already higher than target, avoid bumping
    if original_feerate >= new_feerate.to_sat_per_vb_ceil() as f64 { // [2]
        return Ok(None);
    }

    Ok(Some(Amount::from_sat(effective_feerate_sat_per_vb)))
}
```

First of all, at [1], the increment for the fee is calculated by *dividing* the cost per vbyte by the transaction size which makes no sense. The calculated value should represent the `feerate`, so the fee per vbyte which has nothing to do with the size of the transaction. Therefore the division should be removed alltogether.

Second of all, at [2], the check for whether the new `feerate` is larger than the original incorrectly compares the `original_feerate` with the `new_feerate` instead of the actually returned `effective_feerate_sat_per_vb`.

Recommendation: In order to ensure correct fee bumping, consider changing the two parts of the code:

1. Remove the incorrect division.
2. Change the check at [2] to check against `effective_feerate_sat_per_vb` instead of `new_feerate`.

3.2.2 Incorrect error handling in state machine

Submitted by n4nika

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Summary: Too broad error handling in `process_block_parallel` causes all state machines to permanently halt if a single one errors upon transitioning state.

Finding Description: Whenever a new state machine is added or a finalized block is encountered, `process_block_parallel` is called, executing all state transition of all state machines in that block.

```
if !all_errors.is_empty() {
    // revert state machines to the saved state as the content of the machines might be changed before the error
    ↪  occurred
    self.kickoff_machines = kickoff_machines_checkpoint;
    self.round_machines = round_machines_checkpoint;
    // Return first error or create a combined error
    return Err(eyre::eyre!(
        "Multiple errors occurred during state processing: {:?}",
        all_errors
    ));
}
```

Since there is no mechanism, removing persistently erroring state machines, one corrupted machine will completely halt any state transitions.

Impact: Since the integrity of the system relies on challengers sending disprovals on time, in the worst case this may allow for reimbursement of malicious kickoffs, draining the bridge.

Recommendation: Consider adding more nuanced error handling. For example logic to retry state transitions a certain amount of times and otherwise abort them completely, preventing all state machines from getting stuck permanently.

3.2.3 Operator `get_reimbursement_txs` uses zero `move_txid` when fetching LCP

Submitted by Jiri123, also found by TheStryke, 0xozovehe and Petrus

Severity: Medium Risk

Context: operator.rs#L1819-L1836

Summary: In `Operator::get_reimbursement_txs`, within the branch that sends the kickoff when it is not yet onchain, the code constructs `move_txid` as `Txid::all_zeros()` and then calls `get_payout_info_from_move_txid` with this zero txid. It should use the actual deposit's `move_to_vault` txid, not a zero txid placeholder.

Finding Description:

```
// core/src/operator.rs
let move_txid = Txid::all_zeros();

let (_, _, _, citrea_idx) = self
    .db
    .get_payout_info_from_move_txid(dbtx.as_deref_mut(), move_txid)
    .await?
```

```
    .ok_or_eyre("Couldn't find payout info from move txid")?;

let _ = self
    .citrea_client
    .fetch_validate_and_store_lcp(
        payout_block_height as u64,
        citrea_idx as u32,
        &self.db,
        dbtx.as_deref_mut(),
        self.config.protocol_paramset(),
    )
    .await?;
```

`get_payout_info_from_move_txid` expects a real `move_to_vault_txid` keyed in the DB; zero is not a valid identifier and will typically return None.

Impact Explanation: Medium. Prevents or delays reimbursement flow.

Likelihood Explanation: Medium. This branch runs when kickoff isn't yet on chain.

Recommendations: Replace `let move_txid = Txid::all_zeros();` with the actual `move_to_vault` txid for the deposit in context.