Botanix

Macbeth Review

Version: 2.0

May, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Botanix components. The review focused solely on the security aspects of the Solidity implementation of the contract, 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 Botanix 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 Botanix components in scope.

Overview

Botanix EVM is a Layer 2 EVM-compatible network that leverages Bitcoin as both its settlement layer and native currency. By modifying the Reth client, it achieves full EVM equivalence, enabling seamless compatibility with existing Ethereum smart contracts and developer tooling. To support this architecture, Botanix utilizes CometBFT as its consensus mechanism.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the botanix-labs/Macbeth repository and the ControlCplus-ControlV/stBTC repository.

The scope of this time-boxed review was strictly limited to files at commits 3a5bb70 and 3d22855 respectively. The fixes of the identified issues were assessed at commit 3868b40 for the `Macbeth` repository.

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

Approach

The security assessment covered components written in Solidity and Rust.

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

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

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

To support the Solidity components of the review, the testing team also utilised the following automated testing tools:

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

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

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

To support the Rust components of the review, the testing team also utilised 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 34 issues during this assessment. Categorised by their severity:

- Critical: 10 issues.
- High: 10 issues.
- Medium: 3 issues.
- Low: 5 issues.
- Informational: 6 issues.

Detailed Findings

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

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

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

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

Summary of Findings

ID	Description	Severity	Status
BTNX-01	Peg-In Amounts May Be Stolen By Manipulating `tx.gasprice`	Critical	Resolved
BTNX-02	Unnecessary `Unwrap()` In `PeginData::validate()`	Critical	Resolved
BTNX-03	Unchecked Allocation May Panic In `PeginMeta::deserialization()`	Critical	Resolved
BTNX-04	`transaction.value` Is Used To Refund Invalid Burns	Critical	Resolved
BTNX-05	No Duplication Checks For Peg-In Proofs	Critical	Resolved
BTNX-06	Invalid Peg-in Proofs May Drain The Minting Contract	Critical	Resolved
BTNX-07	Invalid Proof Causes Early Return For Multiple Calls To `mint()`	Critical	Resolved
BTNX-08	Receiver May Spend Minted Funds Before Peg-In Proof Is Verified	Critical	Resolved
BTNX-09	Not All State Changes Are Reverted For An Invalid Peg-In Proof	Critical	Resolved
BTNX-10	Peg-In Proofs Are Only Verified For Top-level Calls	Critical	Resolved
BTNX-11	Pending Peg-Out Validation Incorrectly Accounts For Limits	High	Resolved
BTNX-12	Insufficient PSBT validation	High	Resolved
BTNX-13	`change_output` Validation Can Be Bypassed	High	Resolved
BTNX-14	Lack Of Authentication Of Peers During DKG	High	Resolved
BTNX-15	`PeerMessageResponse::Signing` May Panic On Malformed `signing_session_id`	High	Resolved
BTNX-16	Peg-Out Amount Lacks Validation	High	Resolved
BTNX-17	`FrostProtoMessage::decode_message()` May Panic On Malformed Inputs	High	Resolved
BTNX-18	Partial Mints May Freeze Bridged BTC	High	Closed
BTNX-19	Inaccurate Gas Estimations In `mint()`	High	Resolved
BTNX-20	Base Fee Burned For Legacy and EIP-2930 Transactions	High	Resolved
BTNX-21	Use Of `expect()` In `apply_chunks()` Causes Node To Crash	Medium	Resolved
BTNX-22	Increased Resource Usage By Not Disconnecting Non Federation Peers	Medium	Resolved
BTNX-23	Arithmetic Overflow In `coin_selection()`	Medium	Resolved
BTNX-24	JWT Signatures Shared Without Encryption	Low	Closed
BTNX-25	Irrecoverable State Loss In BTC Server On Checkpoint Failures	Low	Resolved

BTNX-26	Channel Receiver May Never Resolve	Low	**Resolved**
BTNX-27	Non-Determinism Of `bitcoin_checkpoint_block_hash` In ABCI `process_proposal()`	Low	**Closed**
BTNX-28	Outdated Dependencies	Low	**Closed**
BTNX-29	Lack Of Blob Data Storage And Validation For EIP-4844 Transactions	Informational	**Closed**
BTNX-30	`stBTC` Is Vulnerable To An Inflation Attack	Informational	**Closed**
BTNX-31	`finalize_block()` Lacks Aggregate Public Key Check	Informational	**Closed**
BTNX-32	Small Peg-In Amounts Cannot Be Processed	Informational	**Closed**
BTNX-33	Potential Block Rejection Due To Unchecked Transaction Size Limits	Informational	**Resolved**
BTNX-34	Entire Transaction Reverts On Invalid Calls	Informational	**Closed**

BTNX-01	Peg-In Amounts May Be Stolen By Manipulating `tx.gasprice`		
Asset	`contracts/src/Minting.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

While minting synthetic BTC in the L2 system, the refund value calculation uses `tx.gasprice`. The refund value is taken from the user's peg-in `amount` and given to the `refundAddress`. An attacker may manipulate this to reduce the funds distributed to the `destination`.

Minting.sol

```
uint256 txCost =
            (gasStart - gasleft()
                + GAS_INTERNAL_TRANSFER
                + GAS_INTERNAL_TRANSFER
                + GAS_AMOUNT_UPDATE
                + GAS_REVERT_TRUE
                + BASE_GAS_MINT_EVENT
                + metadata.length / 4 - 1)
            * tx.gasprice; // @audit tx.gasprice can be manipulated by sender

// 3 gas for comparison if true
require(txCost <= amount, "Tx cost exceeds pegin amount");

// snipped ...

(bool successRefund, ) = payable(refundAddress).call{value: txCost}("");
require(successRefund, "Refund to refundAddress failed");
```

After EIP-1559, the effective gas price is calculated as `base fee + priority fee (tip)`. In this case, a malicious block builder can use an unrealistically high tip to increase the gas price and inflate the refund amount. If the block builder is the one who submits the mint transaction, they would get the entire peg-in `amount` as a refund. Additionally, they would receive the priority fee back as block rewards. This way, the attacker could steal the user's entire peg-in amount.

Furthermore, if the attacker is not a builder but a regular user who submits the mint transaction, they may execute the same attack. In this case, the attacker would get the entire transaction fee refunded while the user loses all their peg-in `amount`, effectively creating a DoS attack on the peg-in process.

Recommendations

A resolution is to use `block.basefee` instead of `tx.gasprice` while calculating the refund amount.

Resolution

The recommendation has been implemented in PR #687.

Using `block.basefee` ensures that manipulating gas price in no longer feasible.

BTNX-02	Unnecessary `Unwrap()` In `PeginData::validate()`		
Asset	`crates/primitives/src/botanix/peg_contract.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A reachable panic occurs in the `PeginData::validate()` function, when a peg-in transaction is sent with a malformed proof.

In the `PeginData::validate()` function, the merkle root for the block headers is calculated as:

```
let root = merkle.extract_matches(&mut txids, &mut idxs).unwrap();
```

The `extract_matches()` function returns an error in multiple scenarios. One such scenario is decoding a `PeginMeta` object from the `Mint` event which creates a valid `merkle_proof: PartialMerkleTree` object of the form:

```
PartialMerkleTree {
    num_transactions: 0,
    bits: vec![],
    hashes: vec![]
}
```

Note that even though the `PartialMerkleTree` object doesn't allow creating a struct with `num_transactions == 0` from its constructors, it is possible to create an empty object using `PartialMerkleTree::consensus_decode(&mut bytes)`. This is how the `PeginMeta::deserialize()` creates the `merkle_proof`. The result will be a panic due to the `unwrap()` on `merkle.extract_matches()`.

Any user may trigger this function execution via the `Minting` contract, `mint()` function. Therefore, may cause arbitrary node crashes, giving the issue a high likelihood and impact.

Recommendations

Return the error returned by the `PartialMerkleTree::extract_matches()` instead of unwrapping. This would ensure that no invalid `merkle_proof` is accepted.

Resolution

The recommendation has been implementated in PR #677.

BTNX-03	Unchecked Allocation May Panic In `PeginMeta::deserialization()`		
Asset	`crates/primitives/src/botanix/peg_contract.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A memory exhaustion vulnerability exists when deserialising peg-in proofs. The memory exhaustion may crash the node, causing a chain halt.

The function `PeginMetaV0::deserialize()` allocates a vector using `Vec::with_capacity()` based on a decoded value. As such, a malicious user may set the value large enough to exhaust all memory in the machine.

crates/primitives/src/botanix/peg_contract.rs

```
pub fn deserialize(mut bytes: &[u8]) -> Result<(Self, usize), PeginDataError> {

          // ... snipped

           block_headers: {
               let len = btcencode::VarInt::consensus_decode(&mut bytes)?.0; // @audit untrusted input
               let mut ret = Vec::with_capacity(len as usize); // @audit panic if len is larger than available memory
               for _ in 0..len {
                   ret.push(Decodable::consensus_decode(&mut bytes)?);
               }

               ret
           },

          // ... snipped

}
```

The `len` is decoded based on untrusted input that anyone can send to the `Minting.sol` contract in the `metadata` bytes. An attacker may send maliciously crafted `metadata` bytes to the `Minting.sol` contract with the `len` set to a very high number (e.g. `u64::MAX`). The result would be allocating a vector of `u64::MAX` which causes an out of memory panic and crashes the node.

The impact and likelihood are rated as high as arbitrary users may send a transaction to the `Minting` contract, `mint()` function to crash all nodes on the network.

Recommendations

Check that the decoded length from the metadata bytes doesn't exceed a maximum value and fail the decoding if it exceeds the value.

Resolution

The recommendation has been implemented in PR #629.

BTNX-04	`transaction.value` Is Used To Refund Invalid Burns		
Asset	`crates/ethereum/evm/src/execute.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A user may set a transaction value different to the amount attached to `burn()` to refund excess BTC tokens during a peg-out.

If a `burn()` operation is deemed invalid, for example if `metadata` has the wrong format, the user is refunded their burned amount. However, this amount is set to `transaction.value` which is not always equal to the amount that was burned. An attacker can call a smart contract and attach a large value to the transaction to inflate the transaction value. If the contract then calls `burn()` with only a small value and invalid data such that the proof fails, the user will be refunded the entire transaction value without having to burn those funds.

The impact and likelihood are rated as high as arbitrary attackers may exploit this vulnerability to mint an unbounded amount of tokens.

crates/ethereum/evm/src/execute.rs

```
let pegout_amount = transaction.value(); // @audit set `pegout_amount` to transaction.value' rather than burnt amount

// ...

MintContractError::InvalidPegoutData(_) => {
    Self::increment_balance_by_address(
        *sender,
        EthersU256::from_little_endian(pegout_amount.as_le_slice()), // @audit refunds `pegout_amount`'
        &mut state,
    );
}
```

Recommendations

To mitigate the issue refund the actual amount that was burned instead of the transaction value.

Resolution

The issue has been resolved in PR #664 by reverting all state transitions when Botanix validation fails.

The updated code no longer refunds any amount to the user when a burn operation is deemed invalid, instead it reverts the transaction's state changes, ensuring the issue is properly fixed.

BTNX-05	No Duplication Checks For Peg-In Proofs		
Asset	`crates/ethereum/evm/src/execute.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A single call to `mint()` can contain multiple peg-in proofs, this is required when a user performs peg-in transactions within a single bitcoin block. However, no duplication checks occur for these proofs. As such, an attacker can duplicate a valid proof and submit them to `mint()` to multiply their received amount.

The impact and likelihood are high as it allows any user to replay a peg-in an arbitrary amount of times to mint BTC to their Botanix address.

Recommendations

Implement duplication checks or more granular replay protection for peg-in validation.

Resolution

The issue has been resolved in PR #674.

The fix ensures that only unique `outpoints` are accepted, preventing duplicate proofs from being provided.

BTNX-06	Invalid Peg-in Proofs May Drain The Minting Contract		
Asset	`crates/ethereum/evm/src/execute.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

In `execute_state_transitions()`, if the validation of a peg-in proof fails, the balance of the receiver is decremented with the amount they originally received. However, this does not take into account that this amount is then effectively removed from the minting contract and taken out of the total supply. This allows for an attack where a user calls `mint()` with an invalid proof and `amount` set to the minting contract's entire balance. The minting contract then transfers its entire balance to the user, afterwards, the proof is checked and found invalid such that the users balance is decremented again. However, the minting contract is not refunded this balance.

The impact and likelihood of the issue are rated as high as any attacker may cause a permanent denial of service of the peg-in mechanism, since the contract no longer has any balance to pay out to other minters.

The inverse of this issue occurs in `burn()`, where if a `burn()` is invalid the burner is refunded the value without it being taken out of the minting contract's balance.

Recommendations

A mitigation to the issue is to only mint tokens to the user after the proof has been verified. This may be implemented by performing balance modifications during peg-in validation in `execute.rs`. As a result, if the proof is invalid, the minting contract does not lose any balance.

For the `burn()` function, user tokens must be consumed during the smart contract call to avoid double spending. Therefore, the resolution would be to update the `Minting` contract balance if a user is refunded during peg-out validation.

Resolution

The issue has been resolved in PR #664 by reverting state transitions when Botanix validation fails, ensuring the minting contract's balance remains unchanged.

BTNX-07	Invalid Proof Causes Early Return For Multiple Calls To `mint()`		
Asset	`crates/ethereum/evm/src/execute.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

When multiple calls to `mint()` occur in a single transaction only one error is processed.

A user may call `mint()` multiple times in the same transaction, for example by using a multicall smart contract. When these calls are verified in `botanix_mint_contract_checks()`, every mint operation will have their proofs verified iteratively. However, if a proof is deemed invalid an early return may occur, meaning that subsequent proofs are not verified.

crates/ethereum/evm/src/execute.rs

```
/// Performs additional checks on mint contract transactions.
fn botanix_mint_contract_checks(
    &self,
    result: &ExecutionResult,
    botanix_consensus_pkg: &BotanixConsensusPackage,
    tx_hash: TxHash,
    provider: ProviderFactory<RethDB>,
) -> Result<(Vec<PeginData>, Vec<PegoutWithId>), MintContractError> {
    let consensus_pkg = botanix_consensus_pkg;
    let btc_network = consensus_pkg.btc_network;

    // Check pegins.
    let mut pegins = vec![];
    let mut pegouts = vec![];
    for log in result.logs() {
        let pegin_data = match try_parse_mint_event(log)? {
            None => continue,
            Some(p) => p,
        };

        // ... snipped

        // the pegin height must be equal or less than the required block depth (checkpoint)
        if pegin_data.bitcoin_block_height > bitcoin_checkpoint.1 {
            return Err(MintContractError::InvalidPeginData { // @audit example of early return
                error: format!(
                    "pegin height {} greater than checkpoint of {}",
                    pegin_data.bitcoin_block_height, bitcoin_checkpoint.1,
                ),
                revert_address: pegin_data.account,
                revert_amount: pegin_data.amount,
            });
        }
```

The impact is that, the receiver's balance is only decremented for the proofs that were checked and failed, any balances from proofs that were not checked are not decremented. An attacker may exploit this by calling `mint()` twice or more in a single transaction with invalid proofs. Only the first proof will be marked as invalid and the attacker can mint free BTC during the subsequent calls.

Recommendations

All events from a single transaction should be validated and balances adjusted accordingly.

Resolution

The issue has been resolved in PR #664 by reverting state transitions when Botanix validation fails. This ensures the entire transaction is atomically reverted if any proof provided in the transaction is invalid.

BTNX-08	Receiver May Spend Minted Funds Before Peg-In Proof Is Verified		
Asset	`crates/ethereum/evm/src/execute.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Receiver funds cannot be retracted for an invalid proof when they have already been spent.

In `mint()` the receiver is called with the minted amount on line 80. During this call the receiver may perform arbitrary calls, including spending the balance that they just received. If the peg-in proof is later deemed invalid the protocol attempts to decrement the receivers balance. However, seeing as that balance was already spent, a panic occurs. As such, an attacker can crash every node on the network at the same time by spending the received balance during this call.

The funds are distributed during the call to `mint()` as seen in the following code snipept.

Minting.sol

```
function mint(
    address destination,
    uint256 amount,
    uint32 bitcoinBlockHeight,
    bytes calldata metadata,
    address refundAddress
) public {

    // ...

    (bool succeesMint, ) = payable(destination).call{value: amount}(""); //@audit receiver gains control of execution flow here
    require(succeesMint, "Mint to destination failed");
```

After a `mint()` transaction is complete, the core node's state transition will extract `Mint` events and verify the proof and decrement the user balance if the proof is invalid. The balance decrementing panics due to the `expect()` on an overflow.

crates/ethereum/evm/src/execute.rs

```
fn decrement_balance_by_address(address: Address, amount: EthersU256, state: &mut EvmState) {
    let mut account = state.get(&address).expect("Account to exist").clone();
    // print balance before decrement
    info!("Balance before decrement: {:?}", account.info.balance);
    // decrement balance by amount
    info!("Decrementing address: {:?} by {:?}", address, amount);
    account.info.balance = account
        .info
        .balance
        .checked_sub(U256::from_be_bytes(amount.into()))
        .expect("No overflow for checked_sub");
    // update state with new balance
    state.insert(address, account);
}
```

The impact and likelihood is rated as high as any user may create a transaction which calls the `mint()` function to crash the nodes and stall the network.

Recommendations

A solution is to modify the `Minting.sol` contract such that the `receiver` and `refundAddress` are not immediately minted funds. Instead increment the receivers balance directly during the core node state transition function after verifying the peg-in proof.

The solution is similar to the one proposed in BTNX-06.

Resolution

The issue has been resolved in PR #664 by reverting all state transitions when Botanix validation fails.

The updated code no longer subtracts the amount if validation fails, as the entire state of the transaction is reverted, confirming that the issue no longer exists.

BTNX-09	Not All State Changes Are Reverted For An Invalid Peg-In Proof		
Asset	`crates/ethereum/evm/src/execute.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

A security vulnerability exists where an invalid proof may lead to a permanently adjusted `peginBitcoinBlockHeight` in the core node state.

If a peg-in proof is deemed invalid after a `mint()` operation, the receivers balance is decremented to its original value. However, the receiver's `peginBitcoinBlockHeight` is not reverted. Therefore, any user can change `peginBitcoinBlockHeight` for any receiver without needing a valid peg-in proof. As such, an attacker may increment `peginBitcoinBlockHeight` to `u32::MAX` for any receiver. Any subsequent calls to `mint()` will revert for that receiver. This may be used to frontrun any incoming mints and permanently freeze their bridged BTC.

Furthermore, the relayer's balance, that is the `refundAddress` in `mint()`, is also not reverted correctly after an invalid proof is sent. This means that a relayer has their transaction costs refunded even though the proof was not valid. An attacker may abuse this with invalid calls to `mint()`, which fill up the block's gas limit since the transactions are effectively free.

The impact is rated as high as it will permanently prevent valid calls to `mint()` and will result in refunds being paid arbitrarily to addresses.

Recommendations

Revert all state changes that occur in the call to `mint()` if the peg-in proof is deemed invalid.

Resolution

The issue has been resolved in PR #664 by reverting all state transitions when Botanix validation fails.

This means that changes to `peginBitcoinBlockHeight` and `refundAddress` are also reverted if the proof is invalid, thereby resolving the issue.

BTNX-10	Peg-In Proofs Are Only Verified For Top-level Calls		
Asset	`crates/ethereum/evm/src/execute.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

In the `execute_state_transitions()` function `botanix_mint_contract_checks()` is called to ensure any call that was made to the minting contract has a valid peg-in proof. If a peg-in proof is deemed invalid the balance of the receiver is decremented again. However, `botanix_mint_contract_checks()` is only called if `transaction.to()` is equal to the minting contract. This means that if the call to `mint()` is not the top-level call, for example if a smart contract calls `mint()`, the peg-in proof is never verified.

The impact and likelihood is rated a high as it allows an arbitrary user to mint BTC without a valid peg-in proof.

crates/ethereum/evm/src/execute.rs

```
for (sender, transaction) in block.transactions_with_sender() {

    // ...

    if result.is_success() && transaction.to() == Some(*MINT_CONTRACT_ADDRESS) {
        match self.botanix_mint_contract_checks(
            &result,
            &botanix_consensus_pkg,
            transaction.hash,
            provider.clone(),
        ) {

    // ...
```

In a similar vein, calls to `burn()` that are not the top-level call will not be picked up by `botanix_mint_contract_checks()`. The result is users will have their BTC stuck in the minting contract without tokens being released on Bitcoin L1.

Recommendations

To resolve the issue, check each `Mint()` and `Burn()` event emitted by the minting contract regardless of the original destination of the transaction.

Resolution

The recommendation has been implemented in PR #638.

The fix removes the top-level call check that matched against the minting contract address, ensuring that validations are performed for every call that emits events from the minting contract.

BTNX-11	Pending Peg-Out Validation Incorrectly Accounts For Limits		
Asset	`bin/btc-server/src/util.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

When validating pegouts in `validate_outputs()`, the function checks that all `pending_pegout_ids` that are stored in the database are also present in `psbt_pegout_ids`.

bin/btc-server/src/util.rs

```
for pegout_id in pending_pegout_ids.iter() { // @audit validates all pending pegouts in the DB
    if !psbt_pegout_ids.contains(pegout_id) {
        return Err(ValidateOutputsError::MissingPsbtPegout(*pegout_id));
    }
}
```

However, when the `psbt` signing is initiated by the coordinator, the number of pegouts to be included in the `psbt` is limited by `UPPER_PEGOUT_BOUND`.

bin/btc-server/src/bin/main.rs

```
async fn get_psbt(
    &self,
    req: tonic::Request,
) -> Result, tonic::Status> {
    //...snipped

    // Select up to `UPPER_PEGOUT_BOUND` pegouts, sorted by age in ascending
    // order. Respectively, the oldest pegouts come first.
    let pending_pegouts = self.db.coord_pending_pegouts(UPPER_PEGOUT_BOUND).to_status()?; // @audit only fetches UPPER_PEGOUT_BOUND
        ↪  peg-outs
    //...snipped
```

This allows for a scenario where an attacker could create many small peg-out requests in the minting contracts over multiple blocks in the Botanix chain such that the `UPPER_PEGOUT_BOUND` limit is reached. As a result, not all pending pegouts will be included in the `psbt`. As a result, `validate_outputs()` will fail for any newly generated `psbt`, causing a permanent DoS to the peg-out mechanism.

The likelihood is as high as it would allow any user to stall the peg-out process.

Recommendations

Modify `validate_outputs()` such that it accounts for the `UPPER_PEGOUT_BOUND` limit.

Resolution

The issue is no longer applicable, as the related code was removed during redesign, which is detailed in this PR #652.

BTNX-12	Insufficient PSBT validation		
Asset	`bin/btc-server/src/util.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `validate_psbt()` function is executed when a signing package is received from a peer during a FROST signing round. It ensures the `psbt` is correctly formatted and contains the correct inputs and outputs before it is signed. However, there are two places where the outputs of the `psbt` are not sufficiently validated:

- In `validate_psbt_by_ids()` it is checked that for each peg-out in the `psbt` there is a corresponding event on the chain. However, it only checks peg-outs that are in the `psbt.outputs` vector. The actual transaction outputs `psbt.unsigned_tx.output` are not checked. This allows a malicious coordinator to add an arbitrary output to `psbt.unsigned_tx.output` without the validation failing.
- `validate_psbt()` ensures that all pending peg-outs are included in the `psbt`. However, it does not check that all outputs of the `psbt` are pending peg-outs. As such a malicious coordinator may include outputs in `psbt` that are not pending peg-outs.

The issue is rated as high as it allows a malicious user to include invalid outputs which will be signed by the Federation multisig, causing Bitcoin to be released on L1. The issue is restricted to the coordinator as they are the only actor who may initiate a signing round to create a `psbt` and send it to the other signers. Therefore, the likelihood is rated as medium as the coordinator is required to perform the attack.

Recommendations

Ensure the `psbt` is sufficiently validated such that it only contains valid peg-outs.

Resolution

The issue has been resolved in PR #725. The fix ensures that all pegouts in `psbt.outputs` are aligned with those in `psbt.unsigned_tx.output` by checking the lengths of both arrays and verifying that the `script_pubkey` and `amount` at each corresponding index match. This prevents the addition of malicious pegouts.

The latter occurance of the issue is no longer applicable, as the relevant code was removed during the redesign detailed in PR #652.

BTNX-13	`change_output` Validation Can Be Bypassed		
Asset	`bin/btc-server/src/util.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A malicious coordinator can bypass the validation checks on the `change_output` to send the change to their own address.

When a `psbt` is received by a peer during the signing process `validate_outputs()` is called to ensure the `psbt` only contains legitimate peg-out transactions. Additionally, the `change_output` is also validated to ensure that any change from the inputs is sent to the aggregated public key. However, this validation only checks if one of the outputs is sent to the aggregated public key. It does not check that this output is the change output. As such, a malicious coordinator could create a peg-out that is sent to the aggregated public key and include it in the `psbt`. This will result in the validation passing regardless of the actual destination of the `change_output`, allowing the coordinator to steal the change.

bin/btc-server/src/util.rs

```
if !change_outputs.is_empty() {
    // TxOut scriptpubkey should be scriptpubkey derived from aggregated public key
    let agg_pk = public_key_package.verifying_key().to_secp_pk().expect("valid secp pk");
    let expected_script_pubkey = generate_taproot_change_scriptpubkey(&agg_pk);
    // TODO remove the clone here
    let tx = psbt.clone().extract_tx_unchecked_fee_rate();
    let has_correct_change =
        tx.output.iter().any(|o| o.script_pubkey == expected_script_pubkey); // @audit only checks a single output is a expected
            ↪  address
    if !has_correct_change {
        return Err(ValidateOutputsError::InvalidChangeOutput);
    }
}
```

The impact is rated as high as the coordinator may drain the multisig account balance by extracting change from each UTXO. The likelihood is rated as medium as this attack vector is restricted to the coordinator.

Recommendations

It is recommended to validate the specific `change_output` such that the `script_pubkey` is the aggregated public key.

Resolution

The issue has been resolved in PR #679.

The fix involves validating the `script_pubkey` of the `change_output` entry in `psbt.unsigned_tx.output`.

Subsequently, PR #725 ensures that all pegouts in `psbt.outputs` are aligned with those in `psbt.unsigned_tx.output`. This further adds protection against reordering pegouts to perform the attack.

BTNX-14	Lack Of Authentication Of Peers During DKG		
Asset	`crates/consensus/authority/src/frost_task.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

When receiving a message during the distributed key generation (DKG) no authentication of the sender occurs. Rather, the value of `frost_identifier` is assumed to be the sender's identifier. However, `frost_identifier` is part of the message and can be set arbitrarily by the sender. As such, a malicious peer may send dkg round packages while impersonating multiple other peers. This could eventually lead to the malicious peer obtaining enough shares of the key such that they own the entire multisig.

crates/consensus/authority/src/frost_task.rs

```
match peer_message {
    // ...
    PeerMessageResponse::Dkg(dkg_response) => {
        let DkgResponse { response_type, identifier, data } = dkg_response;
        let frost_identifier = match deserialize_frost_peer_id(identifier) {
            Ok(frost_identifier) => frost_identifier,
            Err(e) => {
                error!(target: "consensus::authority::frost_task::start_task", "Error deserializing frost identifier in DKG payload
                    ↪ {:?}", e);
                continue;
            }
```

Recommendations

Add an authentication mechanism using a peer's public key to ensure messages are matched to a peer.

Resolution

The issue is no longer applicable, as the related code was removed during the DKG redesign, which is detailed in this PR #663.

BTNX-15	`PeerMessageResponse::Signing` May Panic On Malformed `signing_session_id`		
Asset	`crates/consensus/authority/src/frost_task.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Peers in the network can craft a malformed `signing_response` and call any event in the `PeerMessageResponse::Signing` block to trigger a panic and crash the node.

crates/consensus/authority/src/frost_task.rs

```
PeerMessageResponse::Signing(signing_response) => {
   let SigningResponse { response_type, signing_session_id, psbt } =
       signing_response;
   let signing_session_id = FixedBytes::from_slice(&signing_session_id); // @audit signing_session_id is not being validated
..
```

The `signing_session_id` is not validated properly before using `FixedBytes::from_slice()`. This function expects the input slice to be exactly 32 bytes long. However, `signing_session_id` is declared as `Vec` in the `SigningResponse` struct, which means it can be of any length. Thus, an attacker can use an arbitrary length of `signing_session_id` to cause the `from_slice()` function to panic.

The impact is rated as high as it allows any network packages to crash a node. The likelihood is rated as medium as this attack vector is only reachable from authorised peers.

Recommendations

Use `FixedBytes::<32>::try_from()` instead of `FixedBytes::from_slice()` and handle the error instead of panicking.

Resolution

The recommendation has been implemented in PR #666.

BTNX-16	Peg-Out Amount Lacks Validation		
Asset	`crates/consensus/authority/src/utils.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Lack of validation of `amount` in the `btc-server` peg-out signing process allows the coordinator to drain the multisig wallet holding the locked Bitcoin.

In the `validate_psbt_by_output()` function, the `amount` parameter from the `PegoutData` is not being validated.

crates/consensus/authority/src/utils.rs

```
/// Validate psbt contains the correct output
pub fn validate_psbt_by_output(
   psbt: &Psbt,
   destination: &Address,
   _amount: Amount,   //@audit amount is not being validated
)
```

A malicious coordinator can exploit this vulnerability to drain locked Bitcoin in L1. The attack vector works as follows:

1. The coordinator performs a `burn()` action for a very small amount in the `minting` contract, triggering the peg-out flow.
2. Later, they craft a malicious `PSBT` (Partially Signed Bitcoin Transaction) which includes the `pegoutId` corresponding to the previous `burn()` action but modifies the peg-out amount to an unrealistically large value.
3. Since the validation function does not check the amount parameter, the validation will be successful and the signing process will be completed.
4. This allows the coordinator to drain the multisig wallet holding the locked Bitcoin.

The impact is rated as high as it allows the coordinator to drain the BTC tokens. The likelihood is rated as medium as the coordinator is required to perform the attack.

Recommendations

Ensure the `PegoutData.Amount` is properly validated.

Resolution

The issue has been resolved in PR #667, which introduces validation of `psbt.unsigned_tx.output` against the expected destination address and amount for each pegout.

BTNX-17	`FrostProtoMessage::decode_message()` May Panic On Malformed Inputs		
Asset	`crates/net/network/src/frost/messages.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `FrostProtoMessage::decode_message()` function decodes messages sent by other trusted federation peers. The function assumes that the bytes sent by peers cannot be an invalid `FrostProtoMessage` message. The malicious peer may send invalid bytes that can crash all other nodes.

For instance, receiving the input `0x00` can immediately crash the node due to an out of bounds access.

crates/net/network/src/frost/messages.rs

```
pub fn decode_message(buf: &mut &[u8]) -> Option {
    if buf.is_empty() {
        return None;
    }
    let id = buf[0];
    buf.advance(1);
    let message_type = match id {
        0x00 => FrostProtoMessageId::Round1Dkg,
        // ... snipped
    };
    let message = match message_type {
        // Other cases remain unchanged
        FrostProtoMessageId::Round1Dkg => {
            let id_len = buf[0] as usize; // @audit index out of bounds panic if buf.len() == 0
```

The panic occurs as `buf` is empty after being advanced and therefore has length zero.

There are numerous issues that may call a panic in this function, consider the following list:

- Unsafe indexing of the slice `buf`.
- Usage of `unwrap()` on UTF-8 string conversion and `PeerID` decoding.
- Calling the function `PeerId::from_slice()` with invalid length input.

The issue is rated as high severity as any panics will cause the node to crash. The likelihood is rated as medium as peer connected nodes may trigger this function with arbitrary data by sending messages on the Frost subnet.

Recommendations

Remove all panics and instead return `None`. Consider the following recommendations.

- Instead of using unsafe indexing, use safe indexing methods on slices like `get()` that return an `Option`.
- Avoid using `unwrap()` and handle error cases.

- Avoid using functions which may panic on malformed input such as `PeerId::from_slice()` and use `try_from()` variants.

Resolution

The issue has been resolved in the PR #683.

The fix adds buffer length checks before indexing, replaces `unwrap()` with safe `try_into()` conversions that handle errors gracefully, and ensures proper handling of UTF-8 conversion errors.

BTNX-18	Partial Mints May Freeze Bridged BTC		
Asset	`contracts/src/Minting.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

A single call to `mint()` can contain multiple peg-in proofs. This is supported in case that a user bridges BTC multiple times in a single BTC block. In such cases the user must provide all their peg-in proofs from that block in one call to `mint()`, if they do not do so, `peginBitcoinBlockHeight` is incremented and they will forever lose access to the other peg-in proofs for that block.

The issue opens up a griefing attack where a malicious front runner can extract a single proof and submit it before the user. The user's `peginBitcoinBlockHeight` will be incremented, invalidating their other proofs, and permanently freezing their bridged BTC.

Recommendations

Implement a more granular replay-protection mechanism than `peginBitcoinBlockHeight`.

Resolution

The issue has been acknowledged, and the team is aware of the risk. A fix is planned for a future version. Full details of the discussion can be found here.

BTNX-19	Inaccurate Gas Estimations In `mint()`		
Asset	`contracts/src/Minting.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

During the execution of `mint()` the execution costs are estimated in `txCost`, at the end of execution this cost is then subtracted from the receiver of the mint and given to the `refundAddress`. Which incentives relayers to call `mint()` in the name of minters, who may not have funds yet to pay for execution costs themselves. However, the calculations for the `txCost` are inaccurate.

Several gas costs are unaccounted for such as execution of the function selector, ABI decoding and potential memory expansion for emitting the event. Additionally, the call on line [80] is assumed to use 2300 gas, however a value transfer to a cold and empty account will cost at least 34300 gas. Furthermore, if the `destination` address is a smart contract its execution cost may be arbitrarily large as it executes bytecode.

As such, the `destination` may spend an arbitrary amount of gas during the `call`. This gas cost would not be subtracted from the `amount` minted to the user and the relayer would absorb the fees.

Recommendations

It is recommended to update the gas cost estimations to be more accurate by using testing tools such as Forge. Additionally, consider limiting the gas stipend for the call to `destination` on line [80] to limit 'gas stealing' attacks.

Resolution

A solution has been implemented in PR #687. The implementation adds a fixed amount of gas, 150, 000, to be refunded.

Additionally, a comment has been added to the contract warning about gas-stealing attacks, as it is the responsibility of the `mint()` caller.

BTNX-20	Base Fee Burned For Legacy and EIP-2930 Transactions		
Asset	`crates/primitives/src/transaction/mod.rs`		
Status	**Resolved:** See Recommendations		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

When calculating the miner tip, it is adjusted to avoid burning the base fee. However, for Legacy and EIP-2930 transactions, the base fee will still be burned.

src/gas/fee.rs

```
pub fn effective_tip_per_gas(&self, base_fee: Option<u64>) -> Option<u128> {
    //...snipped

    let fee = max_fee_per_gas - base_fee;

    if let Some(priority_fee) = self.max_priority_fee_per_gas() {
        Some(fee.min(priority_fee) + base_fee)
    } else {
        Some(fee) // @audit Not adding back base_fee which will be burnt
    }
}
```

For Legacy and EIP-2930 transactions, `self.max_priority_fee_per_gas()` returns `None`. As a result, the else branch simply returns `Some(fee)`, meaning the base fee is not added back to the miner tip calculation. Therefore, the `base_fee` ends up being burned.

Recommendations

Add `base_fee` when calculating the tip for Legacy and EIP-2930 transactions.

Resolution

The recommendation has been implemented in commit e0368b3.

BTNX-21	Use Of `expect()` In `apply_chunks()` Causes Node To Crash		
Asset	`crates/consensus/authority/src/comet_bft/abci.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `apply_snapshot_chunk()` method in `abci.rs` contains a reachable `expect()` when decoding `request.chunk`, allowing peers to cause the node to crash.

Specifically, the following line is problematic:

crates/consensus/authority/src/comet_bft/abci.rs

```
let blocks_with_senders: Vec = compressor
    .decode(request.chunk.as_ref())
    .await
    .expect("Failed to deserialize and decompress block with context"); // @audit reachable by malicious peer
```

If a peer sends a malformed or malicious chunk, the `decode()` operation could result in an `Error`, causing the program to panic due to the use of `expect()`. This results in a denial-of-service (DoS) vulnerability, as a malicious peer could send a malformed chunk causing a client to crash.

The likelihood is rated as low as the issue will only occur during syncing. Syncing will occur when a node is either initially starting up or falls behind the head. The impact is rated as high as the panic is unrecoverable and the node will crash.

Recommendations

Replace the use of `expect()` with proper error handling to gracefully handle decoding failures.

Resolution

The issue has been properly addressed in PR #645 by gracefully handling decoding failures instead of panicking.

BTNX-22	Increased Resource Usage By Not Disconnecting Non Federation Peers		
Asset	`crates/net/network/src/frost/manager.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

In the frost networking subprotocol, when a connection is received from a non-federation peer, the request is ignored in `on_network_event::FrostProtocolEvent::ConnectionEstablished`.

crates/net/network/src/frost/manager.rs

```
if !self.is_authority_peer(&peer_id) {
    return;
}
```

The `on_network_event()` function is triggered from polling the `FrostProtoConnection` stream. After sending the `FrostProtocolEvent::ConnectionEstablished` to the frost manager, the stream goes into `RegistrationState::Pending` state and awaits for a message on `callback_rx` receiver from the `FrostManager`.

However, since the manager returns immediately and does not explicitly disconnect from a non-authority peer, an unauthorised peer will be stuck in the `Pending` registration state and never resolve. An attacker may generate numerous of peer IDs and initiate connections to a federation peer. Each connection attempt consumes additional resources posing a minor DoS risk.

Recommendations

Explicitly disconnect from non authorised peers by dropping the `FrostConnection`.

Resolution

The recommendation has been implementated in PR #684.

BTNX-23	Arithmetic Overflow In `coin_selection()`		
Asset	`bin/btc-server/src/wallet/coin_selection.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

In the `coin_selection()` function, the value of each output in the transaction is subtracted by a fee. This ensures the transaction costs are covered by the peg-outs. However, no check is performed to ensure that the value of the output is larger than the fee. Hence, an overflow may occur on line [**126**]. For this case, the output will have an excessively high value which would result in the transaction being invalid due to insufficient balance.

The impact is a block in the peg-out mechanism which may result in a denial of service.

bin/btc-server/src/wallet/coin_selection.rs

```
for (output, _pegout_id) in pegouts.iter_mut() {
    output.value -= fee_per_output; // @audit potential overflow
}
```

Note there is some limitation in the Solidity contract `burn()` function, which sets a minimum balance of a peg-out. However, this value may not be sufficient when there are a large number of input UTXOs required to craft the transaction.

Recommendations

It is recommended to implement logic for handling outputs with a small value.

Resolution

The issue has been resolved by handling underflow when the output value was too small, as addressed in PR #668.

Additional changes were introduced in PR #717 to ensure that small pegouts cannot cause a denial of service during PSBT generation.

BTNX-24	JWT Signatures Shared Without Encryption		
Asset	`bin/btc-server/src/bin/main.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The Botanix system relies on JWT (JSON Web Token) validation for authenticating requests to the BTC Server. However, the current implementation does not enforce TLS (Transport Layer Security) encryption for the communication channel. This exposes sensitive JWT tokens to potential interception and replay attacks by malicious actors.

Without TLS, JWT tokens are transmitted in plaintext, making them vulnerable to interception by attackers using man-in-the-middle (MITM) or eavesdropping attacks.

An intercepted JWT token can be reused by an attacker to impersonate a legitimate user or system, leading to unauthorised access to critical operations, such as `new_consensus_checkpoint()` and signing messages.

The development team have stated that the BTC server and Botanix Reth binary are intended to be run on the same machine. Therefore, for an attacker to exploit this issue they would need access to the machine. As such, this issue is rated as low likelihood.

Recommendations

Ensure users are aware of the limitation between the BTC Server and Botanix Reth connection. If these are intended to be used on separate machines, consider the use of a reverse proxy.

Resolution

The design is intentional, as confirmed by the development team. Their response is as follows:

> *Will let node operators know about this limitation and recommend using a reverse proxy if the processes are on separate machine.*

BTNX-25	Irrecoverable State Loss In BTC Server On Checkpoint Failures		
Asset	`crates/consensus/authority/src/frost_task.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

While handling canon state notifications, `new_consensus_checkpoint()` is called to sync the peg-ins and pending peg-outs of the finalised block to the `btc_server`.

However, the `new_consensus_checkpoint()` gRPC call can fail for several reasons, such as network issues or validation errors. If the call fails, there is currently no way to re-sync the peg-ins and `pending_pegouts` data of the current block to the `btc_server`.

If a validator consistently misses checkpoints, it will have a different view of `utxos` and `pending_pegouts` compared to other validators. In the worst-case scenario, if the peg-outs of a block are not synced by enough validators, those peg-outs could be lost forever.

Recommendations

Allow the `new_consensus_checkpoint()` call to be retried on an error, or enable the peg-in and `pending_pegouts` data to be resynced if an error occurs.

Resolution

The recommendation has been implemented in PR #669.

BTNX-26	Channel Receiver May Never Resolve		
Asset	`crates/authority/src/dkg.rs`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

In the `get_all_peers_handle()` function, a `FrostCommand` is sent to the manager. An error is logged if the channel send errors. However, it does not return early.

Therefore, in the case of an error, the `await` will never resolve.

crates/authority/src/dkg.rs

```
if let Err(e) = self
    .frost_handle
    .send_command(FrostCommand::GetAllConnectedPeers(peers_connections_sender))
{
    error!(target: "consensus::authority::dkg::get_all_peers_handle", "Failed to send GetAllConnectedPeers frost command {}", e);
}
// @audit This will never resolve if the send was an error
match peers_connections_receiver.await { ... }
```

The issue poses a minor denial of service risk as each indefinitely pending thread will increase resource consumption.

Recommendations

Return an error if the `send_command()` fails.

Resolution

The issue has been resolved in PR #715.

The fix involves adding continue statements when `send_command()` encounters an error, ensuring that the subsequent `await` statement is not executed.

BTNX-27	Non-Determinism Of `bitcoin_checkpoint_block_hash` In ABCI `process_proposal()`		
Asset	`crates/consensus/authority/src/comet_bft/abci.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

When an incoming block is validated in `process_proposal()` the bitcoin block hash included in the block is compared to the local `bitcoin_checkpoint`. If they are not equal the block is rejected. The local `bitcoin_checkpoint` is dependent on the highest block seen by the local bitcoin node and as such may not be equal on all nodes. This introduces non-determinism which may result in liveness issues.

crates/consensus/authority/src/comet_bft/abci.rs

```
if bitcoin_checkpoint_block_hash != non_deterministic_data.bitcoin_block_hash {
    warn!("Bitcoin block hash mismatch");
    return ResponseProcessProposal { status: VERIFY_REJECT };
}
```

The `bitcoin_checkpoint_block_hash` is set in `bin/reth/src/commands/poa/mod.rs` as the current bitcoin height, less the configured confirmation depth. It is possible for different nodes to have a different Bitcoin block height and therefore the checkpoint hash will be different between nodes.

The issue is rated as low severity as the nodes will quickly align on the current head of the bitcoin chain, due to the infrequency of Bitcoin blocks. Non-determinism is undesirable in CometBFT and may slow block or halt production. However, when nodes converge on the bitcoin checkpoint hash the chain will resume producing blocks.

Recommendations

A potential solution here is to base the local `bitcoin_checkpoint` on time rather than number of confirmations. For example, given the consensus request time, go back 1 hour and pick the first block before then as a checkpoint. This will ensure each node selects the same bitcoin block as a checkpoint.

Resolution

The development team have acknowledged the issue and have decided to address this in a future release.

BTNX-28	Outdated Dependencies		
Asset	`*.rs`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The following dependencies are outdated and have security issues:

- `crossbeam-channel`
- `fast-float`
- `idna`
- `protobuf`
- `ring`

The full details can be found by running `cargo audit`.

Recommendations

Consider updating the dependencies with security issues. Additionally, consider setting up monitoring infrastructure to ensure any security advisories for dependencies are noticed and can be fixed.

Resolution

The development team has decided to address this in a future release. Their response is as follows:

> *We will address this in the future but not now.*

BTNX-29	Lack Of Blob Data Storage And Validation For EIP-4844 Transactions
Asset	`crates/consensus/authority/src/comet_bft/abci.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `process_proposal()` function in `abci.rs` accepts EIP-4844 transactions with blob commitments. However, there is no mechanism in place to validate or store the associated blob data. This creates a scenario where transactions with blob commitments can be processed without the actual blob data being provided or persisted.

EIP-4844 transactions are processed successfully without including the blob sidecar. The blob sidecar contains the underlying data associated with the blob transaction, which is not stored or validated. As opposed to the EIP-4844 transaction, which only contains the commit hash of the blob data.

Users will be charged gas for including blob transactions, yet the blob data itself is unavailable for retrieval. Users may incorrectly expect blob data to be retrievable after inclusion of an EIP-4844 transaction.

The issue is rated as informational severity as the development team are aware of this issue and do not intend to support blob storage or retrieval, yet EIP-4844 transaction types are accepted for compatibility.

Recommendations

Ensure the issue is understood and users are aware that EIP-4844 transaction may cost additional gas without storing blob data.

Resolution

The development team has decided to address this in a future release. Their response is as follows:

> *We do not plan to address this at this time.*

BTNX-30	`stBTC` Is Vulnerable To An Inflation Attack
Asset	`stBTC/src/stBTC.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `stBTC` contract is based on OpenZeppelin's implementation of an ERC4626 vault. It protects against inflation attacks by using virtual shares. However this only makes the attack less profitable, it doesn't fully prevent it. For example, if there are multiple victim deposits, instead of just one, the attack may still be profitable.

An additional protection is implemented in `stBTC` by overriding `totalAssets()`. Instead of using `asset.balanceOf(vault)`, the balance is calculated internally. This makes the donation part of the inflation attack much harder, since a simple donation to the vault is not registered and thus has no effect on the share price. However, this can still be circumvented by calling `notifyRewardAmount()` after the donation and waiting 7 days for the full reward period to end. After this, the donation balance is entirely registered.

A step-by-step of the attack could look like this:

1. The attacker deposits 1 asset.
2. The attacker donates 10,000 assets to the vault.
3. The attacker calls `notifyRewardAmount()` and waits 7 days for the reward period to end. Crucially, no large deposits may occur during this time.
4. Some victims deposit multiple times into the vault, where each deposit must be smaller or equal to 5,000 assets. Due to the inflated share price each deposit will receive 0 shares. The attacker's profit increases with the amount of victim deposits, at least 10,000 assets must be deposited in total for the attack to be profitable.

Given the very low likelihood of this attack being successful, the testing team rates this as an informational issue.

Recommendations

Ensure the above comments are understood, and consider increasing the vault's `_decimalsOffset()`. This increases the accuracy of the share price and provides additional protection against inflation attacks. Alternatively, the vault can be seeded with initial funds after deployment to avoid inflation attacks.

Resolution

The design is intentional, as confirmed by the development team. Their response is as follows:

> *We are not addressing this issue as the risk is deemed acceptable and very low likelihood of happening.*

BTNX-31	`finalize_block()` Lacks Aggregate Public Key Check
Asset	`crates/consensus/authority/src/comet_bft/abci.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `finalize_block()` function in `abci.rs` does not include a check to ensure the aggregated public key is correct. This may lead to a block with the wrong aggregated public key being finalised.

The check is present in `process_proposal()` and therefore, this is only an issue if a super majority of CometBFT nodes confirm a block with an invalid aggregate public key such that `finalize_block()` is called.

crates/consensus/authority/src/comet_bft/abci.rs

```
fn process_proposal(&self, request: RequestProcessProposal) -> ResponseProcessProposal {

  //...

  if agg_pk != non_deterministic_data.aggregated_public_key {
      warn!("Aggregate public key mismatch");
      return ResponseProcessProposal { status: VERIFY_REJECT };
  }
}
```

Recommendations

Consider adding this check to `finalize_block()`.

Resolution

The issue is a know design choice by the development team. The following response was provided by the development team:

> *Since `finalize_block` is called during block sync and the aggregate public key is non-deterministic (it may change), we cannot determine what the agg pub key should have been. We are relying on the super majority to have confirmed the agg pub key correctly during live consensus.*

BTNX-32	Small Peg-In Amounts Cannot Be Processed
Asset	`contracts/src/Minting.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

When processing a peg-in during `mint()`, if the `amount` of the peg-in is less than the gas cost the transaction is reverted.

Minting.sol

```
function mint(
    address destination,
    uint256 amount,
    uint32 bitcoinBlockHeight,
    bytes calldata metadata,
    address refundAddress
) public {

    // ...

    uint256 txCost =
        (gasStart - gasleft()
            + GAS_INTERNAL_TRANSFER
            + GAS_INTERNAL_TRANSFER
            + GAS_AMOUNT_UPDATE
            + GAS_REVERT_TRUE
            + BASE_GAS_MINT_EVENT
            + metadata.length / 4 - 1)
        * tx.gasprice;

    // 3 gas for comparison if true
    require(txCost <= amount, "Tx cost exceeds pegin amount"); // @audit small amounts cannot be minted

    // 3 gas for subtraction and 2000 to update the local variable
    amount -= txCost;
```

The issue is raised as informational severity as the current solution will prevent dust amounts for peg-in UTXOs being processed, which may be considered a benefit. Thus, it may be desirable to leave the code as is.

Recommendations

Ensure the above comments are understood and consider changes if desired.

Resolution

The design is intentional, as confirmed by the development team. Their response is as follows:

> *This is desirable and leaving as is.*

BTNX-33	Potential Block Rejection Due To Unchecked Transaction Size Limits
Asset	`crates/consensus/authority/src/comet_bft/abci.rs`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Reth prepare payload is used to create a block proposal without guarantees each transaction fits within the CometBFT transaction size limits.

The `prepare_proposal()` function gathers transactions from the Reth mempool and adds the non-deterministic transaction to make a block. Noting that any transactions that are sent in `request.txs` from CometBFT are explicitly ignored.

crates/consensus/authority/src/comet_bft/abci.rs

```
// These are bytes of [SignedTransaction]
let mut txs: Vec<_> = block
    .raw_transactions()
    .iter()
    .map(|tx| prost::bytes::Bytes::copy_from_slice(tx))
    .collect::<_>();
info!("prepare_proposal number of txs: {:?}", txs.len());

// insert non-deterministic data tx at index 0 so historical sync will pass
// verification
txs.insert(0, non_deterministic_data_bytes);
```

Seeing as the transactions directly handled by Reth have different limits compared to CometBFT's transactions, their size may exceed `request.max_tx_bytes`. This could lead to the block being rejected by CometBFT, resulting in a failed block proposal. This transaction size requirement can be seen in Requirement 2 of CometBFT's documentation.

The issue is rated as informational severity the current default limit for CometBFT's `max_tx_bytes` is around 1MB, whereas the default limit for Reth transactions is 128KB.

Recommendations

Ensure transactions are only inserted as long as their size remains under `max_tx_bytes`.

Resolution

The recommendation has been implemented in PR #647.

BTNX-34	Entire Transaction Reverts On Invalid Calls
Asset	`crates/ethereum/evm/src/execute.rs`
Status	**Closed:** See Resolution
Rating	Informational

Description

When executing a transaction, if any of the Botanix Peg-in or Peg-out validation fail, the entire transaction is being reverted atomically. It allows anyone to call the minting contract's `mint()` or `burn()` functions with invalid parameters, causing the entire transaction to revert, as opposed to the current call context.

On Ethereum, when interacting with untrusted smart contracts, developers often use a fixed gas stipend. If the called contract fails and consumes all the gas provided, the caller still retains some gas and can recover control to handle the error. This safety mechanism no longer applies if a call to `mint()` or `burn()` with invalid parameters reverts the entire transaction.

This behaviour impacts systems that rely on partial execution or gas isolation, such as account abstraction frameworks, where a relayer might deduct ERC20 tokens from a user before executing their intended call. Also affects bridges like CCIP that invoke external calls with fixed gas stipends (e.g., `ccipReceive()`), and batching mechanisms that process multiple user operations in a single transaction while allowing others to succeed even if one fails

Recommendations

Refactor the execution logic to ensure that only the state changes resulting from the mint or burn actions are reverted when the Botanix checks fail.

Resolution

The development team have acknowledged the issue and may enact modifications to address this.

Appendix A Vulnerability Severity Classification

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

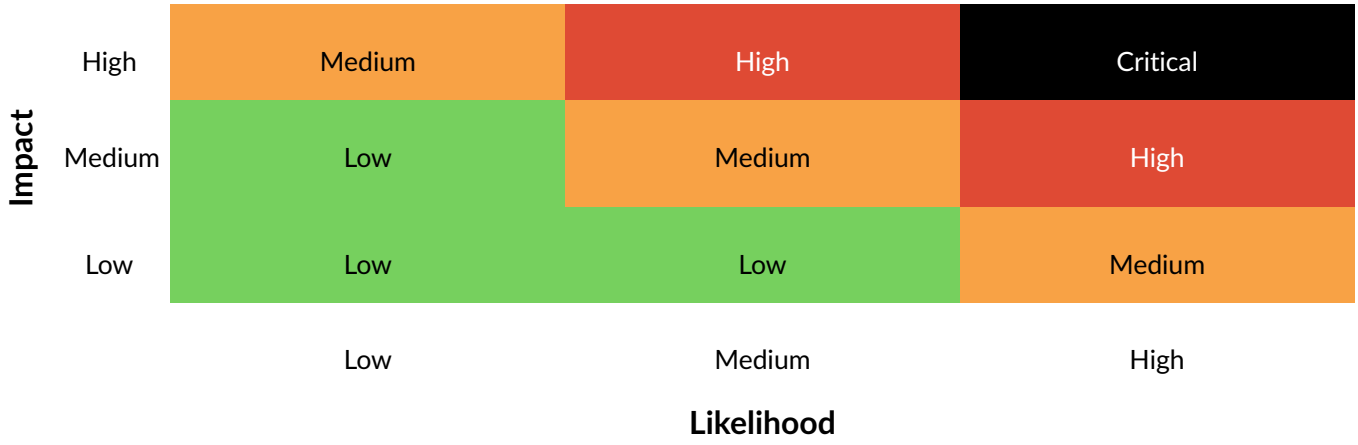

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

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

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