Jito BAM Validator

Security Assessment

August 8th, 2025 — Prepared by OtterSec

Nicola Vella	nick0ve@osec.io
Michael Debono	mixy1@osec.io
Jinwoo Lee	jin@osec.io
Robert Chen	r@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Jito Labs engaged OtterSec to assess the `bam-validator` program. This assessment was conducted between June 29th and August 5th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 9 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability concerning the lack of signature verification in BAM mode, allowing malicious nodes to inject invalid packets and risk validator DoS and consensus divergence (OS-JBM-ADV-00), and another signature replay issue, where the BAM auth challenge is signed without domain separation, rendering the signature re-utilizable in other contexts (OS-JBM-ADV-02). Also, if the transaction sending and confirmation fail, the payment transaction may still be processed later, risking duplicate fee payments if the same transaction is re-sent after the failure (OS-JBM-ADV-03).

Additionally, tips may be lost due to uncranked payment instructions, and there is a possibility of restricted accounts bypassing protections due to missing blocklist enforcement (OS-JBM-ADV-01), and the fee calculation logic while calculating the payment amount overlooks precompile signature instruction costs, potentially overestimating excess fees and overpaying the BAM node (OS-JBM-ADV-04).

Furthermore, while updating the builder commission value in the BAM manager module, there is a lack of an upper bound check on the value of the commission, allowing excessively high inputs that may result in an invalid state and disrupt validator reward distribution (OS-JBM-ADV-06).

We also recommended utilizing the actual index of the failing transaction when validating each transaction in a batch, to improve the accuracy of error reporting (OS-JBM-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/jito-labs/jito-solana-jds. This audit was performed against commit 1d83c0a.

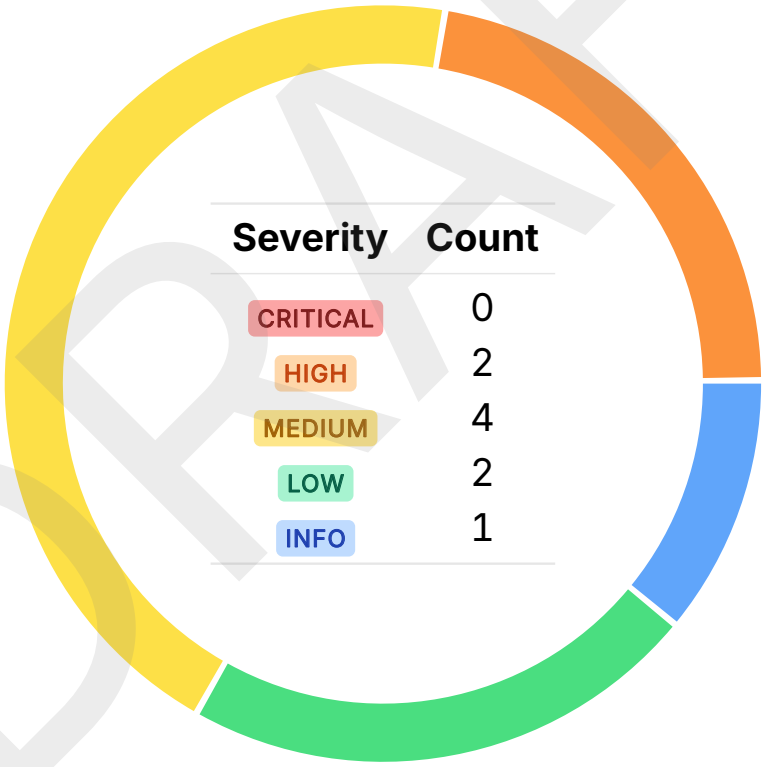
A brief description of the programs is as follows:

Name	Description
bam-validator	Extends the Jito-Solana client to allow validators to delegate transaction ordering to external schedulers via gRPC, enabling optimized block construction while preserving validator control and Solana's execution rules.

03 — Findings

Overall, we reported 9 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	2
MEDIUM	4
LOW	2
INFO	1

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-JBM-ADV-00	HIGH	RESOLVED	Signature verification check is not performed in BAM mode, allowing malicious nodes to inject invalid packets and risk validator DoS and consensus divergence.
OS-JBM-ADV-01	HIGH	RESOLVED	Tips may be lost due to uncranked payment instructions, and missing blocklist enforcement allows restricted accounts to bypass protections.
OS-JBM-ADV-02	MEDIUM	RESOLVED	The BAM auth challenge is signed without domain separation, rendering the signature reutilizable in other contexts.
OS-JBM-ADV-03	MEDIUM	RESOLVED	Currently, if `send_and_confirm_transaction` fails, the payment transaction may still be processed later, risking duplicate fee payments.
OS-JBM-ADV-04	MEDIUM	RESOLVED	The fee calculation in `calculate_payment_amount` overlooks precompile signature instruction costs, potentially overestimating excess fees and overpaying the BAM node.
OS-JBM-ADV-05	MEDIUM	RESOLVED	`insert_new_batch` may exceed the container's capacity without evicting entries, resulting in `get_vacant_map_entry` panicking and creating a denial-of-service scenario.

OS-JBM-ADV-06	LOW	RESOLVED	There is a lack of an upper bound check on the value of builder commission, allowing excessively high inputs that may result in an invalid state and disrupt validator reward distribution.
OS-JBM-ADV-07	LOW	RESOLVED	`parse_bundle` fails to reject vote transactions from BAM nodes, deviating from BAM protocol specifications.

Absence of Signature Verification in BAM Mode HIGH OS-JBM-ADV-00

Description

Currently in BAM mode, packets bypass the `sigverify` step, implying transactions are not validated for proper cryptographic signatures. Consequently, this allows a malicious BAM node to inject invalid transactions into a validator's pipeline, resulting in denial-of-service for validators and state divergence. Such divergence creates consensus forking, violating the invariant that BAM nodes should only stall transaction flow or fail, and not compromise correctness.

Remediation

Add `sigverify` checks for BAM-mode transactions.

Patch

Resolved in 8ce6993.

Improper Tip Handling and Missing Blocklist Enforcement HIGH OS-JBM-ADV-01

Description

The system fails to crank tip payment instructions before executing a tip transfer, causing pending tips to be skipped and resulting in lost validator rewards. This undermines the incentive mechanism by reducing earned fees. Additionally, the blocklist for the tip payment account is not enforced, allowing restricted or malicious accounts to interact with the system.

Remediation

Ensure the tip payment instructions aren't cranked before a tip transfer and add the blocklist for the tip payment account.

Patch

Resolved in ddf9e55 and 8370f94.

Missing Domain Separator MEDIUM OS-JBM-ADV-02

Description

`prepare_auth_proof` in `bam_connection` signs arbitrary challenge data from the BAM scheduler without applying domain separation. This results in a vulnerability because the signed message may be re-utilized in unintended contexts. Without a fixed prefix, the signature is not cryptographically bound to the BAM authentication domain. To prevent misuse, the signed message should include a domain separator.

>_ *core/src/bam_connection.rs* RUST

```rust
async fn prepare_auth_proof(
    validator_client: &mut BamNodeApiClient<tonic::transport::channel::Channel>,
    cluster_info: Arc<ClusterInfo>,
) -> Option<AuthProof> {
    let request = tonic::Request::new(AuthChallengeRequest {});
    let Ok(resp) = validator_client.get_auth_challenge(request).await else {
        error!("Failed to get auth challenge");
        return None;
    };
    let resp = resp.into_inner();
    let challenge_to_sign = resp.challenge_to_sign;
    let challenge_bytes = challenge_to_sign.as_bytes();

    let signature = Self::sign_message(cluster_info.keypair().as_ref(), challenge_bytes)?;
    [...]
}
```

Remediation

Pre-append a domain separator such as `\xffsolana offchain-jito bam` to the message in `bam_connection` so that it is only possible for it to be exclusively utilized for only BAM authentication.

Patch

Resolved in f55c5b5.

Risk of Transaction Replay MEDIUM OS-JBM-ADV-03

Description

In the current implementation, `bam_payment::payment_successful` assumes that a failure in `send_and_confirm_transaction` implies the transaction did not land on-chain. However, due to potential network or RPC issues, the transaction may have succeeded without local confirmation. Retrying the transaction in such cases may result in duplicate payments and overpayment.

>_ *core/src/bam_payment.rs* RUST

```rust
fn payment_successful(
    rpc_url: &str,
    txn: &VersionedTransaction,
    lowest_slot: Slot,
    highest_slot: Slot,
) -> bool {
    // Send it via RpcClient (loopback to the same node)
    let rpc_client = RpcClient::new_with_commitment(rpc_url, CommitmentConfig::confirmed());
    if let Err(err) = rpc_client.send_and_confirm_transaction(txn) {
        error!(
            "Failed to send payment transaction for slot range ({}, {}): {}",
            lowest_slot, highest_slot, err
        );
        false
    } [...]
}
```

Remediation

Utilize durable nonces to ensure the transaction is only valid for a single execution.

Patch

Bam payments have been disabled.

Lack of Precompile Fee Accounting MEDIUM OS-JBM-ADV-04

Description

The fee calculation logic in `calculate_payment_amount` in `bam_payment` incorrectly assumes that the base cost of a transaction is solely determined by the number of signatures. It subtracts a fixed `BASE_FEE_LAMPORT_PER_SIGNATURE` from the total fee to compute the payment base, but this neglects the costs of precompile signature instructions such as `ed25519`, `secp256k1`, and `secp256r1`, which incur significant compute unit costs. As a result, the function overestimates the fee and overpays the BAM node.

>_ core/src/bam_payment.rs RUST

```rust
pub fn calculate_payment_amount(blockstore: &Blockstore, slot: u64) -> Option<u64> {
    [...]
    const BASE_FEE_LAMPORT_PER_SIGNATURE: u64 = 5_000;
    Some(
        block
            .transactions
            .iter()
            .map(|tx| {
                let fee = tx.meta.fee;
                let base_fee = BASE_FEE_LAMPORT_PER_SIGNATURE
                    .saturating_mul(tx.transaction.signatures.len() as u64);
                fee.saturating_sub(base_fee)
            })
            .sum::<u64>()
            .saturating_mul(COMMISSION_PERCENTAGE)
            .saturating_div(100),
    )
}
```

Remediation

Ensure proper accounting, considering all protocol-imposed base costs, including precompile execution.

Patch

Resolved in ed0e01a.

Denial of Service on New Batch Insertion MEDIUM OS-JBM-ADV-05

Description

`get_vacant_map_entry` in `transaction_state_container` panics when the internal `Slab` exceeds its `capacity`, due to an assertion. This assertion panics when the number of elements in `id_to_transaction_state` reaches its maximum `capacity`. This is problematic because `insert_new_batch` does not check or preemptively manage capacity before calling it multiple times in a loop.

>_ *core/src/banking_stage/transaction_scheduler/transaction_state_container.rs* RUST

```rust
fn get_vacant_map_entry(&mut self) -> VacantEntry<BatchIdOrTransactionState<Tx>> {
    assert!(self.id_to_transaction_state.len() < self.id_to_transaction_state.capacity());
    self.id_to_transaction_state.vacant_entry()
}
```

Thus, since `insert_new_batch` does not perform any eviction similar to what `push_ids_into_queue` does, if multiple batches are inserted rapidly, the `Slab` may exceed `capacity`, triggering a panic on the next `get_vacant_map_entry`, creating a denial of service scenario.

Remediation

Replace the hard assertion with a graceful fallback.

Patch

Resolved in 18ea92b.

Failure to Cap Commission Value LOW OS-JBM-ADV-06

Description

`update_block_engine_key_and_commission` in `bam_manager` blindly accepts and applies the `builder_commission` value from the BAM config without enforcing an upper bound. This is risky because an abnormally high commission may create issues downstream in the validator, potentially resulting in an invalid state that affects validators' MEV earnings.

>_ *core/src/bam_manager.rs* RUST

```rust
fn update_block_engine_key_and_commission(
    config: Option<&ConfigResponse>,
    block_builder_fee_info: &Arc<Mutex<BlockBuilderFeeInfo>>,
) {
    [...]
    let commission = builder_info.builder_commission;
    let mut block_builder_fee_info = block_builder_fee_info.lock().unwrap();
    block_builder_fee_info.block_builder = pubkey;
    block_builder_fee_info.block_builder_commission = commission;
}
```

Remediation

Add a sanity check to reject unreasonable commission values to help maintain correctness.

Patch

Resolved in 75c2319.

Inconsistency in Vote Transaction Filtering Logic LOW OS-JBM-ADV-07

Description

`BamReceiveAndBuffer::parse_bundle` does not enforce BAM's specification that prohibits vote transactions in incoming bundles. Without this check, vote transactions may be accepted and scheduled, violating protocol rules. Add a check to reject vote transactions from BAM nodes.

Remediation

Reject such transactions in `parse_bundle`, ensuring adherence to BAM specifications.

Patch

Resolved in fc1bb8f.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-JBM-SUG-00	`parse_batch` incorrectly reports all deserialization errors at `index: 0`, even when failures occur later in the batch, resulting in inaccurate error attribution.

Incorrect Error Index Reporting OS-JBM-SUG-00

Description

`parse_batch` in `bam_receive_and_buffer` validates each transaction in a batch, but incorrectly hardcodes `index: 0` when reporting deserialization errors, even though it is processing multiple transactions in a loop and the failure occurs in a later transaction. This results in misleading diagnostics hindering the debugging process, as the hardcoded `index` may misrepresent which transaction failed.

>_ *core/src/banking_stage/transaction_scheduler/bam_receive_and_buffer.rs* RUST

```rust
fn parse_batch(
    batch: &AtomicTxnBatch,
    bank_forks: &Arc<RwLock<BankForks>>,
) -> Result<ParsedBatch, Reason> {
    [...]
    for (index, parsed_packet) in parsed_packets.drain(..).enumerate() {
        // Check 1
        let Some((tx, deactivation_slot)) = parsed_packet.build_sanitized_transaction(
            vote_only,
            root_bank.as_ref(),
            root_bank.get_reserved_account_keys(),
        ) else {
            return Err(Reason::DeserializationError(
                jito_protos::proto::bam_types::DeserializationError {
                    index: 0,
                    reason: DeserializationErrorReason::SanitizeError as i32,
                },
            ));
        };[...]
    }
    [...]
}
```

Remediation

Utilize the actual `index` of the failing transaction during error reporting to ensure accuracy.

Patch

Resolved in 3666184.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.