Skip Block SDK

Audit

Presented by:

OtterSec contact@osec.io

James Wang james.wang@osec.io

Robert Chen r@osec.io

Contents

01 Executive Summary **2**

Overview 2

Key Findings 2

02 Scope **3**

03 Findings **4**

04 Vulnerabilities **5**

OS-SBLK-ADV-00 [high] | Unchecked Block Gas Limit 6

OS-SBLK-ADV-01 [high] | Insufficient Block Size Check 7

OS-SBLK-ADV-02 [med] | Auction Transaction Unbundling 8

OS-SBLK-ADV-03 [low] | Insufficient Bid Fund Assertion 9

05 General Findings **10**

OS-SBLK-SUG-00 | Unchecked Error 11

06 Invariant Fuzzing **12**

Appendices

A Vulnerability Rating Scale **16**

B Procedure **17**

01 | Executive Summary

Overview

Skip Protocol engaged OtterSec to perform an assessment of the `block-sdk` program. This assessment was conducted between September 9th and October 10th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 5 findings in total.

In particular, we discussed insufficient checks against block limits which may cause validators to construct illegal blocks (OS-SBLK-ADV-00, OS-SBLK-ADV-01), suggested possible solutions to prevent unbundling of auctioned txs which may lead to loss of MEV searchers (OS-SBLK-ADV-02), and discovered incorrect balance checks that might allowing specific accounts to evade paying auction fees (OS-SBLK-ADV-03).

We also made recommendations on properly checking errors returned by critical functions to improve code quality (OS-SBLK-SUG-00).

To validate critical invariants such as `PrepareProposal` and `ProcessProposal` equivalence, we also produced a proof-of-concept fuzzer, described in more detail in chapter 06.

02 | Scope

The source code was delivered to us in a git repository at github.com/skip-mev/block-sdk. This audit was performed against commit 3c6f319.

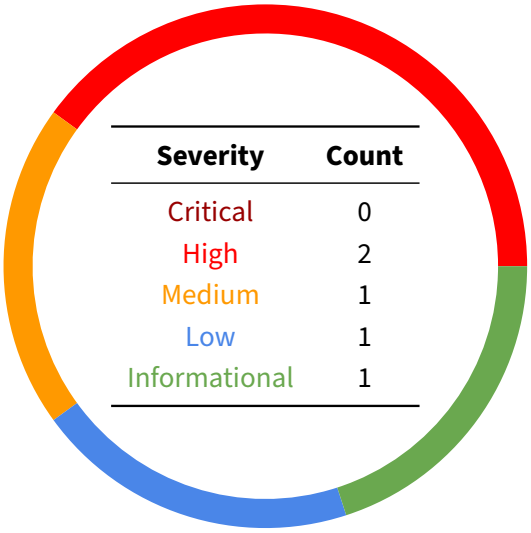
A brief description of the programs is as follows:

Name	Description
block-sdk	A general-purpose block constructing module that allows adopters of Cosmos chains to have finer control over block construction. This module splits the overall `mempool` into several `lanes`, allowing each `lane` to adhere to different tx sorting priorities. The Skip team also provided an implementation of the `MEV lane`, allowing MEV searchers to bid for the top-of-block position to place their transaction bundle.

03 | Findings

Overall, we reported 5 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	1
Low	1
Informational	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-SBLK-ADV-00	High	Resolved	Block gas limits are not checked during both `prepareProposal` and `processProposal` stages, allowing the creation of extra costly blocks.
OS-SBLK-ADV-01	High	Resolved	The summation of transaction sizes for the MEV lane is done incorrectly and may allow the construction of blocks that exceed the maximum block size.
OS-SBLK-ADV-02	Medium	Resolved	Transactions within an auction may be unbundled and submitted as standalone ones, potentially losing MEV searchers.
OS-SBLK-ADV-03	Low	Resolved	Assertion against the balance of bidding accounts is insufficient to guarantee the ability to pay auction fees.

OS-SBLK-ADV-00 [high] | Unchecked Block Gas Limit

Description

Cosmos-sdk allows chains to configure both the maximum block size and gas limit. The current implementation only checks the total block size against the maximum block size and does not validate gas limits.

block-sdk/block/base/handlers.go GO

```go
func (l *BaseLane) DefaultPrepareLaneHandler() PrepareLaneHandler {
    return func(
        ctx sdk.Context,
        proposal block.BlockProposal,
        maxTxBytes int64
    ) ([][]byte, []sdk.Tx, error) {
        [...]
        for iterator := l.Select(ctx, nil); iterator != nil; iterator =
            ↪  iterator.Next() {
            tx := iterator.Tx()

            txBytes, hash, err := utils.GetTxHashStr(l.TxEncoder(), tx)
            [...]
            txSize := int64(len(txBytes))
            if updatedSize := totalSize + txSize; updatedSize > maxTxBytes {
                [...]
                break
            }
            [...]
            totalSize += txSize
            txs = append(txs, txBytes)
        }

        return txs, txsToRemove, nil
    }
}
```

This would allow validators to propose blocks that exceed gas limits and, in the worst-case scenario, result in partial execution of MEV bundled transactions.

Remediation

Check accumulated gas limit against max block gas.

Patch

Resolved in b9d6761.

OS-SBLK-ADV-01 [high] | Insufficient Block Size Check

Description

When checking transaction sizes against block size limits, the `MEV` lane does not consider that bundled transactions will be unpacked in `prepareProposal` and included twice in the block proposal. Thus, it underestimates the total block size and may result in the construction of blocks that exceed block size limits.

block-sdk/lanes/mev/abci.go GO

```
func (l *MEVLane) PrepareLaneHandler() base.PrepareLaneHandler {
    return func(
        ctx sdk.Context,
        proposal block.BlockProposal,
        maxTxBytes int64
    ) ([][]byte, []sdk.Tx, error) {
        [...]
        for ; bidTxIterator != nil; bidTxIterator = bidTxIterator.Next() {
            cacheCtx, write := ctx.CacheContext()
            tmpBidTx := bidTxIterator.Tx()

            bidTxBz, hash, err := utils.GetTxHashStr(l.TxEncoder(), tmpBidTx)
            [...]
            bidTxSize := int64(len(bidTxBz))
            if bidTxSize <= maxTxBytes {
                [...]
                for index, rawRefTx := range bidInfo.Transactions {
                    // Does not count bundled tx size againt to account for
                        ↪ unbundling in block
                    [...]
                }
                [...]
            }
            [...]
        }
        [...]
    }
}
```

Remediation

Track bundled transaction sizes while building proposals and break when they exceed block size limits.

Patch

Resolved in d495b38.

OS-SBLK-ADV-02 [med]| Auction Transaction Unbundling

Description

The original implementation of auction transactions requires each sub-transaction in the bundle to be signed by its sender. However, since there are no assertions requiring those bundled transactions to be included within an auction, it is possible for malicious actors to actively search for submitted auction bundles, extract transactions from within them, and send those as standalone transactions.

The code below shows that each bundled transaction is passed through `antehandlers`, which contains the same signature checks applied to standalone transactions. Submitting bundled transactions as standalone ones would also pass the signature check and be accepted.

block-sdk/lanes/mev/check_tx.go GO

```
func (handler *CheckTxHandler) ValidateBidTx(
    [...]
) (sdk.GasInfo, error) {
    [...]
    for _, tx := range bidInfo.Transactions {
        bundledTx, err := handler.mevLane.WrapBundleTransaction(tx)
        [...]

        if ctx, err = handler.anteHandler(ctx, bundledTx, false); err != nil {
            return gasInfo, fmt.Errorf("invalid bid tx; failed to execute bundled
                ↪  transaction: %w", err)
        }
    }
    [...]
}
```

Remediation

Enforce that all bundled transactions are signed by the bidder, and set a timeout for `auctionTx` to the next block. This ensures that

1. Bundled transactions cannot be resubmitted in later blocks.
2. Since all bundled transactions will have sequence numbers following after `auctionTx`, if it is not included in a block, neither will the bundled transactions.

One notable instance that must be made clear is that bidders are expected not to submit another transaction with the same sequence number as `auctionTx` into other lanes. Doing so would break assertion (2) listed above. This is the bidders' responsibility and should be noted in `block-sdk` documents.

Patch

Resolved in 339b927.

OS-SBLK-ADV-03 [low] | Insufficient Bid Fund Assertion

Description

One of the most crucial aspects in verifying the validity of an auction bid transaction is to ensure that the bidder has a sufficient balance to cover the bid. In the original implementation, this is done by comparing the bidder's balance with the specified bid.

block-sdk/x/auction/keeper/auction.go GO

```
func (k Keeper) ValidateAuctionBid(
    ctx sdk.Context,
    bidder sdk.AccAddress,
    bid, highestBid sdk.Coin
) error {
    [...]

    // ensure the bidder has enough funds to cover all the inclusion fees
    balances := k.bankKeeper.GetBalance(ctx, bidder, bid.Denom)
    if !balances.IsGTE(bid) {
        return fmt.Errorf("insufficient funds to bid %s with balance %s", bid,
            ↪  balances)
    }

    return nil
}
```

However, when the bidder is a vesting account, part of the balance may be locked and not spendable, rendering this comparison ineffective. In the worst-case scenario, this would allow vesting accounts to submit high bids without paying the fee.

Remediation

Perform an actual coin send instead of only checking the balance. This would ensure that the specified bid is spendable.

Patch

Resolved in 3374203.

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-SBLK-SUG-00	The error returned by `GetAuctionBidInfo` is ignored in `ValidateBidTx`.

OS-SBLK-SUG-00 | Unchecked Error

Description

The `err` returned by `GetAuctionBidInfo` is ignored in `ValidateBidTx`. Since `bindInfo != nil` here may imply that either the transaction is not a `auctionTx`, or the `auctionTx` is malformed, checking it without also validating `err` is imprecise. Although the program will catch a malformed `auctionTx` later in the `anteHandler`, it is recommended to add checks to reinforce the validation logic here.

block-sdk/lanes/mev/check_tx.go GO

```
func (handler *CheckTxHandler) ValidateBidTx(ctx sdk.Context, bidTx sdk.Tx, bidInfo
    ↪  *types.BidInfo) (sdk.GasInfo, error) {
    [...]
    // Verify all of the bundled transactions.
    for _, tx := range bidInfo.Transactions {
        [...]
        // bid txs cannot be included in bundled txs
        bidInfo, _ := handler.mevLane.GetAuctionBidInfo(bundledTx)
        if bidInfo != nil {
                return gasInfo, fmt.Errorf("invalid bid tx; bundled tx cannot be a
                    ↪  bid tx")
        }

        if ctx, err = handler.anteHandler(ctx, bundledTx, false); err != nil {
                return gasInfo, fmt.Errorf("invalid bid tx; failed to execute
                    ↪  bundled transaction: %w", err)
        }
    }

return gasInfo, nil
}
```

Remediation

Assert `err == nil` for `GetAuctionBidInfo`.

06 | Invariant Fuzzing

Here, we present a discussion about the fuzzing of chain implementations. We include example implementations, recommendations, and general ideas to test critical invariant.

The accuracy of the block-sdk relies on the `priority_nonce` upholding an important invariant. Due to the standalone nature of this component, with minimal external dependencies, and the relatively complex logic involved in transaction selection from the `priority_nonce mempools`, it is advised to adopt dynamic testing techniques to verify its correctness.

Here, we showcase two fuzzing tests as examples that demonstrate our effort to ensure the correctness of `priority_nonce`.

To begin, we test the determinism of the `priority_nonce` sorting algorithm. Transactions within the `mempool` should be sorted in a strictly decreasing order after considering all tie-breakers. This means that for any two `mempools` containing identical transactions, `Proposals` constructed should always be the same, regardless of the order in which transactions are inserted into the `mempools`.

To validate this invariant, we randomly generate `TX_CNT` valid transactions and insert those transactions into several `mempools` in various orders. We then verify that the `Proposals` constructed by each `mempool` are identical.

block-sdk-fuzzer/fuz/main$_t$est.go GO

```
f.Fuzz(func(t *testing.T, data []byte) {
    [...]
    acct := make([]DummyTx, TX_CNT)
    for i := 0; i < TX_CNT; i++ {
        acct[i].accountIdx, _ = fuzzConsumer.GetUint16()
        acct[i].nonce, _ = fuzzConsumer.GetUint16()
        [...]
    }
    [...]
    txs := make([]authsigning.Tx, TX_CNT)

    for i := 0; i < TX_CNT; i++ {
        tx, err := testutils.CreateRandomTx(
            encodingConfig.TxConfig,
            accounts[acct[i].accountIdx % ACCT_CNT],
            uint64(acct[i].nonce),
            [...]
        )
        [...]
        txs[i] = tx
    }

    resps := make([]*cometabci.ResponsePrepareProposal, COPIES)
```

```
    for i := 0; i < COPIES; i++ {
        defaultLane := defaultlane.NewDefaultLane(cfg)
        mempool := block.NewLanedMempool(logger, false, defaultLane)
        rand.Shuffle(TX_CNT, func(j, k int) {
            txs[j], txs[k] = txs[k], txs[j]
        })
        for j := 0; j < TX_CNT; j++ {
            if err := defaultLane.Insert(sdk.Context{}, txs[j]); err != nil {
                t.Errorf("unexpected insertion error")
            }
        }
        [...]
        resp, err := prepareProposalHandler(ctx,
            ↪  &cometabci.RequestPrepareProposal{Height: 2})
        [...]
        resps[i] = resp
    }

    for i := 1; i < COPIES; i++ {
        if !reflect.DeepEqual(resps[0].Txs, resps[i].Txs) {
            [...]
            t.Errorf("proposal mismatched")
        }
    }
})
```

It is worth noting that a special scenario arises when a newer transaction is received from the same `sender` with the same `nonce`. In such cases, the newer transaction replaces older ones in the `mempool`. To avoid false positives during testing, we have implemented additional code to adjust the `nonce` whenever a collision occurs during the generation of random transactions.

block-sdk-fuzzer/fuz/main$_{test}$.go GO

```
    var seen map[uint16]map[uint16]bool = make(map[uint16]map[uint16]bool)
    for i := 0; i < TX_CNT; i++ {
        acct[i].accountIdx %= ACCT_CNT
        idxMap, ok := seen[acct[i].accountIdx]
        if !ok {
            seen[acct[i].accountIdx] = make(map[uint16]bool)
            idxMap = seen[acct[i].accountIdx]
        }
        if _, ok = idxMap[acct[i].nonce]; ok {
            for k := 0; k <= math.MaxUint16; k++ {
                if _, ok = idxMap[uint16(k)]; !ok {
                    acct[i].nonce = uint16(k)
                    break
                }
            }
        }
        idxMap[acct[i].nonce] = true
    }
```

Our second test concerns the utilization of priorities which may vary between the time transactions are inserted into `mempool`, and selection from the `mempool`.

For instance, if `sender balance` is used as priority, transactions that stay in the `mempool` for more than one block may witness changes in its `sender balance` due to execution of other transactions.

While it is unlikely to pose an issue since `priority_nonce` captures the priority during insertion and uses it after without trying to re-fetch the priority, we conducted tests to validate our assumptions.

For this purpose, we implemented a custom priority function that reads from a caller controllable mutable `DynamicContext` structure. This allows us to easily modify `priorities` without the need for a complex setup. By minimizing the use of excessive `cosmos-sdk` APIs, our test solely focuses on the key invariant under scrutiny.

block-sdk-fuzzer/fuz/main$_{test}$.go GO

```go
type DynamicContext struct {
        DynamicPriority map[string]map[uint64]uint16
}

func DynamicTxPriority(ctx DynamicContext, t *testing.T) base.TxPriority[uint16] {
        return base.TxPriority[uint16]{
                GetTxPriority: func(goCtx context.Context, tx sdk.Tx) uint16 {
                        sender, sequence, err := getTxSenderInfo(tx)

                        if err != nil {
                                t.Errorf("unexpected getTxSenderInfo error")
                                return 0
                        }

                        senderPriorities, ok := ctx.DynamicPriority[sender]

                        if !ok {
                                t.Errorf("unexpected get dynamicPriority error
                                    ↪  (sender)")
                                return 0
                        }

                        priority, ok := senderPriorities[sequence]

                        if !ok {
                                t.Errorf("unexpected get dynamicPriority error
                                    ↪  (sequence)")
                                return 0
                        }

                        return priority
                },
                [...]
        }
}
```

Once the `DynamicContext` is implemented, we proceed to introduce random priority mutations for each transaction immediately upon insertion. Subsequently, we apply the same `Proposal` comparison to verify the fulfillment of the invariant

block-sdk-fuzzer/fuz/main$_{t}est.go$ GO

```go
for i := 0; i < TX_CNT; i++ {
    acct[i].accountIdx, _ = fuzzConsumer.GetUint16()
    acct[i].nonce, _ = fuzzConsumer.GetUint16()
    [...]
    for j := 0; j < VARIATIONS; j++ {
        priorities[i][j].priority, _ = fuzzConsumer.GetUint16()
    }
}
[...]
for i := 0; i < VARIATIONS; i++ {
    [...]
    for j := 0; j < TX_CNT; j++ {
        sender, sequence, err := getTxSenderInfo(txs[j])
        if err != nil {
            t.Errorf("unexpected tx sender info error")
        }
        senderPriority, ok := dctx.DynamicPriority[sender]
        if !ok {
            dctx.DynamicPriority[sender] = make(map[uint64]uint16)
            senderPriority, _ = dctx.DynamicPriority[sender]
        }
        senderPriority[sequence] = priorities[j][0].priority
        if dynamicLane.Insert(ctx, txs[j]) != nil {
            t.Errorf("unexpected insertion error")
        }
        //change priority between Insert and Select (this is no-op first iter)
        dctx.DynamicPriority[sender][sequence] = priorities[j][i].priority
    }
    resp, err := prepareProposalHandler(ctx,
        ↪  &cometabci.RequestPrepareProposal{Height: 2})
    [...]
    resps[i] = resp
}
```

The complete code for those fuzzing tests can be found at github.com/otter-sec/block-sdk-fuzzer.

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 section.

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.
Informational	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 the other 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.