code4rena

Initia

Smart Contract Security Assessment

Audit dates: Jan 07 — Jan 21, 2025

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Initia Rollup Modules smart contract system. The audit took place from January 07 to January 21, 2025.

This audit was judged by LSDan.

Final report assembled by Code4rena.

Summary

The C4 analysis yielded an aggregated total of 9 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 6 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 2 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Initia Rollup Modules repository, and is composed of 7 smart contracts written in the Golang programming language.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (3)

[H-01] Challenger misses discrepancy events, allowing executors to perform malicious actions

Submitted by 0xAlix2

https://github.com/initia-labs/opinit-bots/blob/640649b97cbfa5782925b7dc8c0b62b8fa5367f6/challenger/child/handler.go#L95

Finding Description and Impact

The Challenger is a critical process responsible for monitoring the Executor's output proposals and challenging any invalid submissions. It ensures the integrity of the rollup's state and prevents malicious actions by the Executor.

The Challenger operates by accumulating emitted events from both the L1 and L2 chains. At the end of every L2 block, a handler is triggered to validate the events and determine if any challenges need to be issued.

When a user deposits tokens into the L2 by sending an `InitiateTokenDeposit` transaction on the L1, the following event is emitted:

Event emission in L1:

```
sdk.UnwrapSDKContext(ctx).EventManager().EmitEvent(sdk.NewEvent(
        types.EventTypeInitiateTokenDeposit,
        sdk.NewAttribute(types.AttributeKeyBridgeId,
strconv.FormatUint(bridgeId, 10)),
        sdk.NewAttribute(types.AttributeKeyL1Sequence,
strconv.FormatUint(l1Sequence, 10)),
        sdk.NewAttribute(types.AttributeKeyFrom, req.Sender),
        sdk.NewAttribute(types.AttributeKeyTo, req.To),
        sdk.NewAttribute(types.AttributeKeyL1Denom, coin.Denom),
        sdk.NewAttribute(types.AttributeKeyL2Denom, l2Denom),
        sdk.NewAttribute(types.AttributeKeyAmount, coin.Amount.String()),
        sdk.NewAttribute(types.AttributeKeyData,
hex.EncodeToString(req.Data)),
))
```

This event is caught by the host component in the Challenger and saved to the child's database at the end of the L1 block:

Host event saving:

```
// save all pending events to child db
err =
eventhandler.SavePendingEvents(h.stage.WithPrefixedKey(h.child.DB().Prefi
xedKey), h.eventQueue)
```

When the Executor finalizes the deposit on the L2 by sending a `FinalizeTokenDeposit` message, the following event is emitted:

Event emission in L2:

```
event := sdk.NewEvent(
        types.EventTypeFinalizeTokenDeposit,
        sdk.NewAttribute(types.AttributeKeyL1Sequence,
strconv.FormatUint(req.Sequence, 10)),
        sdk.NewAttribute(types.AttributeKeySender, req.From),
        sdk.NewAttribute(types.AttributeKeyRecipient, req.To),
        sdk.NewAttribute(types.AttributeKeyDenom, coin.Denom),
        sdk.NewAttribute(types.AttributeKeyBaseDenom, req.BaseDenom),
        sdk.NewAttribute(types.AttributeKeyAmount, coin.Amount.String()),
        sdk.NewAttribute(types.AttributeKeyFinalizeHeight,
strconv.FormatUint(req.Height, 10)),
)
```

The child component of the Challenger catches this event and saves it in the child's database.

At the end of an L2 block, the `endBlockHandler` is called:

```
func (ch *Child) endBlockHandler(ctx types.Context, args
nodetypes.EndBlockArgs) error {
        // ...

        // check value for pending events
1@>     challenges, processedEvents, err :=
ch.eventHandler.CheckValue(ctx, ch.eventQueue)
        if err != nil {
                return err
        }
        pendingChallenges = append(pendingChallenges, challenges...)
        // check timeout for unprocessed pending events
        unprocessedEvents :=
ch.eventHandler.GetUnprocessedPendingEvents(processedEvents)
2@>     challenges, timeoutEvents :=
ch.eventHandler.CheckTimeout(args.Block.Header.Time, unprocessedEvents)
3@>     pendingChallenges = append(pendingChallenges, challenges...)

        // ...

        ch.eventHandler.DeletePendingEvents(processedEvents)
        ch.eventHandler.SetPendingEvents(timeoutEvents)
4@>     ch.challenger.SendPendingChallenges(challenges)
        return nil
}
```

The handler performs the following steps:

- Checks for discrepancies in the mapped events, such as `InitiateTokenDeposit` and `FinalizeTokenDeposit` (`1@`).
- Checks for timeout events among unprocessed pending events (`2@`).
- Groups both types of events into `pendingChallenges` for later handling/challenging/rollbacks (`3@`).

However, at `4@`, the handler sends only the `challenges` variable, which is overridden when fetching timeout events. This results in discrepancies being missed, as they are not included in `pendingChallenges`.

The `SendPendingChallenges` function sends these challenges to a channel (`c.challengeCh`) for asynchronous handling. The `challengeHandler` function, running in a separate goroutine, processes challenges from this channel, the discrepancy events won't be included in this.

This defeats the purpose of the Challenger bot, allowing malicious Executors to act without detection.

Recommended Mitigation Steps

```
func (ch *Child) endBlockHandler(ctx types.Context, args
nodetypes.EndBlockArgs) error {
        // ...

        ch.eventHandler.DeletePendingEvents(processedEvents)
        ch.eventHandler.SetPendingEvents(timeoutEvents)
-       ch.challenger.SendPendingChallenges(challenges)
+       ch.challenger.SendPendingChallenges(pendingChallenges)
        return nil
}
```

beer-1 (Initia) confirmed and commented:

Mitigation here.

`challenges => pendingChallengs`

https://github.com/initia-labs/opinit-bots/blame/90594098d4dc95f14bd409c69aee2fd772610041/challenger/child/handler.go#L95

berndartmueller (warden) commented:

I just want to mention that those challenges are currently not acted upon (e.g., not sent to an alerting system, or actions and counter-measures taken):

https://github.com/initia-labs/opinit-bots/blob/640649b97cbfa5782925b7dc8c0b62b8fa5367f6/challenger/handler.go#L72-L77

```
func (c *Challenger) handleChallenge(ctx types.Context, challenge
challengertypes.Challenge) error {
	// TODO: warning log or send to alerting system
	ctx.Logger().Error("challenge", zap.Any("challenge", challenge))

	return nil
}
```

Therefore, the challenger is currently not effective and it does not prevent malicious actions.

a_kalout (warden) commented:

@berndartmueller - The protocol already has those challenges errored out in the logs, which are, of course, caught somewhere else on their end. If you're referencing the `TODO`, it is an improvement for later. This doesn't change the fact that the discrepancy events are being thrown away.

Our responsibility here is to make sure it is caught alongside the timeout issues, which are not being caught. This is enough impact to show that this is a high-severity issue.

LSDan (judge) commented:

I agree. Leaving this in place as high.

[H-02] `L1->L2` token deposits can be DoS'ed by purposefully providing a large `data` field in `MsgInitiateTokenDeposit`

Submitted by RadiantLabs

On L2, `MsgFinalizeTokenDeposit` must be relayed in strict sequence, matching `finalizedL1Sequence, err := ms.GetNextL1Sequence(ctx)`

x/opchild/keeper/msg_server.go::FinalizeTokenDeposit(..)

```go
finalizedL1Sequence, err := ms.GetNextL1Sequence(ctx)
if err != nil {
    return nil, err
}

if req.Sequence < finalizedL1Sequence {
    // No op instead of returning an error
    return &types.MsgFinalizeTokenDepositResponse{Result: types.NOOP}, nil
} else if req.Sequence > finalizedL1Sequence {
    return nil, types.ErrInvalidSequence
}
```

As long as a specific L1 sequence is not successfully processed, subsequent `L1->L2` deposits have to wait and cannot be processed. If it were possible to purposefully create a `MsgFinalizeTokenDeposit` message that cannot be executed on L2, it will **DoS all subsequent deposits to L2**!

Due to strict validation on the L1 in `InitiateTokenDeposit(..)` it seems almost impossible to create such a message.

However, the `data` field in `MsgFinalizeTokenDeposit` is not restricted in size. This allows creating a message that is so large that it cannot be processed on L2. Specifically, the RPC message size limit (`rpc-max-body-bytes` and `max_body_bytes`) and the mempool/consensus max tx size limit (`maxTxBytes`) might be exceeded, preventing the message from being processed.

The RPC limits messages by default to **1MB**, while `maxTxBytes` is set to **2MB** (on `L1` and the `minimove L2`).

Note that the RPC limit is not 100% effective. An active validator is not limited by it and can include a message in a proposed block that is larger than the RPC limit, but smaller than the mempool limit.

Given that `data` is unrestricted, and `MsgInitiateTokenDeposit` on L1 contains less properties than `MsgFinalizeTokenDeposit` on L2, it is possible to create a `MsgInitiateTokenDeposit` message that is smaller than the enforced 2MB, but when relayed to L2, the `MsgFinalizeTokenDeposit` message is too large (e.g., due to the additional `BaseDenom` field and others) and thus rejected.

MsgFinalizeTokenDeposit

```go
type MsgFinalizeTokenDeposit struct {
    // the sender address
    Sender string `protobuf:"bytes,1,opt,name=sender,proto3" json:"sender,omitempty" yaml:"sender"`
    // from is l1 sender address
    From string `protobuf:"bytes,2,opt,name=from,proto3" json:"from,omitempty"`
    // to is l2 recipient address
    To string `protobuf:"bytes,3,opt,name=to,proto3" json:"to,omitempty"`
    // amount is the coin amount to deposit.
    Amount types1.Coin `protobuf:"bytes,4,opt,name=amount,proto3" json:"amount" yaml:"amount"`
    // sequence is the sequence number of l1 bridge
    Sequence uint64 `protobuf:"varint,5,opt,name=sequence,proto3" json:"sequence,omitempty"`
    // height is the height of l1 which is including the deposit message
    Height uint64 `protobuf:"varint,6,opt,name=height,proto3" json:"height,omitempty"`
    // base_denom is the l1 denomination of the sent coin.
    BaseDenom string `protobuf:"bytes,7,opt,name=base_denom,json=baseDenom,proto3" json:"base_denom,omitempty"`
    /// data is a extra bytes for hooks.
    Data []byte `protobuf:"bytes,8,opt,name=data,proto3" json:"data,omitempty"`
}
```

VS MsgInitiateTokenDeposit

```go
type MsgInitiateTokenDeposit struct {
    Sender   string     `protobuf:"bytes,1,opt,name=sender,proto3" json:"sender,omitempty" yaml:"sender"`
    BridgeId uint64 `protobuf:"varint,2,opt,name=bridge_id,json=bridgeId,proto3" json:"bridge_id,omitempty" yaml:"bridge_id"`
    To       string     `protobuf:"bytes,3,opt,name=to,proto3" json:"to,omitempty" yaml:"to"`
    Amount   types.Coin `protobuf:"bytes,4,opt,name=amount,proto3" json:"amount" yaml:"amount"`
    Data    []byte    `protobuf:"bytes,5,opt,name=data,proto3"
json:"data,omitempty" yaml:"data"`
}
```

Additionally, the executor batches messages into a single Cosmos tx without any check that the incremental payload exceeds the maximum length that can be accepted by the API, which might lead to this scenario occurring organically if enough messages of "normal" size accumulate.

Ultimately, `L1->L2` deposits can be DoS'ed by purposefully providing a large `data` field which results in the L2 `MsgFinalizeTokenDeposit` message being too large to be processed.

Proof of Concept

The following PoC demonstrates how a large `data` field in `MsgInitiateTokenDeposit` can lead to a DoS of `L1->L2` deposits.

Setup

Add the following changes. For simplicity, the L1 RPC limit is increased. Note that this does not affect the mempool limit. The RPC limit can be sidestepped by a block proposer anyway.

`opinit-bots/e2e/helper.go#L253`

```
api := make(testutil.Toml)
api["rpc-max-body-bytes"] = 5_000_000
c["api"] = api
```

`opinit-bots/e2e/helper.go#L260`

```
configToml := make(testutil.Toml)
rpc := make(testutil.Toml)
rpc["max_body_bytes"] = 5_000_000
configToml["rpc"] = rpc

err = testutil.ModifyTomlConfigFile(ctx, logger, client, t.Name(),
l1Chain.Validators[0].VolumeName, "config/config.toml", configToml)
require.NoError(t, err)
```

Test Case

Paste the test case into `opinit-bots/e2e/multiple_txs_test.go` and run with`go test -v . -timeout 30m -run TestMultipleDepositsAndWithdrawalsExceedingLimit`.

▶ Details

After a while, the logs will show that the Cosmos transaction containing the `MsgFinalizeTokenDeposit` message is rejected due to the message size limit.

```
2025-01-13T22:05:38.586Z    WARN    executor
broadcaster/process.go:119    retry to handle processed msgs
{"seconds": 8, "count": 2, "error": "simulation failed: error in json rpc
client, with http response metadata: (Status: 400 Bad Request, Protocol
HTTP/1.1). RPC error -32600 - Invalid Request: error reading request
body: http: request body too large"}
```

This is a proof that `L1->L2` deposits can be DoS'ed by purposefully providing a large `data` field that results in the L2 `MsgFinalizeTokenDeposit` message being too large to be processed on L2.

The size of `data` can be meticulously crafted to be just below the L1 RPC and mempool limit, but exceeding the limits on L2.

One way to do so is to alter the `NewInitiateTokenDeposit` command to accept data size instead of content:

```
            dataLen, err := strconv.ParseUint(args[3], 10, 64)
            if err != nil {
                return err
            }

            data := make([]byte, dataLen)
```

With this change, it is possible to bisect incrementally larger sizes to find the right limit of a signed transaction (below 749667 is found):

```
➜  initia git:(main) ✗ build/initiad tx ophost initiate-token-deposit 1
abc 1uinit 749667 --from=validator --chain-id=initia-1
[...]
raw_log: 'out of gas in location: txSize; gasWanted: 200000, gasUsed:
7500502: out
  of gas'
[...]

➜  initia git:(main) ✗ build/initiad tx ophost initiate-token-deposit 1
abc 1uinit 749668 --from=validator --chain-id=initia-1
[...]
```

```
error in json rpc client, with http response metadata: (Status: 400 Bad
Request, Protocol HTTP/1.1). RPC error -32600 - Invalid Request: error
reading request body: http: request body too large
```

Recommended mitigation steps

Consider restricting the size of the `data` field in `MsgInitiateTokenDeposit` to a reasonable limit, such as 100KB. Additionally, the executor should be modified to handle large messages in a more graceful manner, such as splitting them into multiple transactions if the simulation fails due to the message size limit.

beer-1 (Initia) confirmed, but disagreed with severity and commented:

Thanks for your report! It seems one can halt the relaying, but it can be solved by increasing the rpc limit or decreasing the block gas limit kind of way.

But it would good to have limit on data field. We will apply it.

Mitigation here.

LSDan (judge) commented:

Given the temporary nature of the impact (ease of mitigation) I agree that this is more appropriate as a medium.

berndartmueller (warden) commented:

While changing the RPC limit is done per RPC/validator, and thus can be done quickly on the RPC that is used by the off-chain executor bot, increasing the RPC limit is only half of the mitigation.

The RPC limit can be bypassed if the attacker is a validator who proposes a block. In this case, the relevant limit is the mempool's `maxTxBytes`. This parameter is a consensus-relevant parameter, meaning, the majority of the nodes must increase this limit. Otherwise, the network continues to reject a block that contains such a large tx. Therefore, the mitigation is not that easy and requires coordination among the validators, which might take a while.

If this happens during times when the market is volatile, and users must quickly bridge assets from `L1->L2` to e.g., improve the health of borrowing positions, this is especially severe and will result in lost funds. Therefore, we kindly request considering this submission as a High.

LSDan (judge) commented:

Thank you for the additional clarification. I agree with your reasoning and have upgraded it back to high.

[H-03] Storage root assignment missing in tree finalization

Submitted by OMEN

https://github.com/initia-labs/opinit-bots/blob/640649b97cbfa5782925b7dc8c0b62b8fa5367f6/challenger/child/withdraw.go#L117-L170

Summary

The `handleTree` function in the challenger's child chain processing fails to assign the merkle tree root hash to the storage root return value, causing the output handling logic to never execute.

Impact

Severity: High

The bug prevents proper state synchronization between L1 and L2 chains by:

1. Never triggering output handling.
2. Missing state root updates.
3. Potentially breaking cross-chain verification.
4. Disrupting the challenge mechanism.

Technical Details

Location:

`opinit-bots/challenger/child/withdraw.go`:

```
func (ch *Child) handleTree(ctx types.Context, blockHeight int64,
blockHeader cmtproto.Header) (storageRoot []byte, err error) {
    if ch.finalizingBlockHeight == blockHeight {
        finalizedTree, newNodes, treeRootHash, err :=
ch.Merkle().FinalizeWorkingTree(nil)
        // ... saving trees and nodes ...

        // BUG: treeRootHash is never assigned to storageRoot
        // Should be: storageRoot = treeRootHash
    }
    return storageRoot, nil  // Always returns nil
}
```

Call Site Impact:

`opinit-bots/challenger/child/handler.go`

```
func (ch *Child) endBlockHandler(ctx types.Context, args
nodetypes.EndBlockArgs) error {
    storageRoot, err := ch.handleTree(ctx, blockHeight,
args.Block.Header)
    if storageRoot != nil {  // This condition never evaluates to true
        // Output handling never executes
        err = ch.handleOutput(...)
    }
}
```

Proof of Concept

1. System processes blocks normally.
2. At finalization height:
    - Tree is finalized successfully.
    - Root hash is calculated.
    - But root hash is never returned.
3. Output handling is skipped.
4. State synchronization fails.

Root Cause: Missing assignment of the calculated `treeRootHash` to the return value `storageRoot` in the `handleTree` function.

Recommended mitigation steps

`opinit-bots/challenger/child/withdraw.go`

```
func (ch *Child) handleTree(ctx types.Context, blockHeight int64,
blockHeader cmtproto.Header) (storageRoot []byte, err error) {
    if ch.finalizingBlockHeight == blockHeight {
        finalizedTree, newNodes, treeRootHash, err :=
ch.Merkle().FinalizeWorkingTree(nil)
        if err != nil {
            return nil, errors.Wrap(err, "failed to finalize working
tree")
        }

        // Fix: Assign tree root hash
        storageRoot = treeRootHash

        // ... rest of the function ...
    }
    return storageRoot, nil
}
```

beer-1 (Initia) confirmed and commented:

Mitigation here.

Medium Risk Findings (6)

[M-01] Apply oracle update doesn't check for duplicate validator votes

Submitted by 0xAlix2

When an Oracle update is required, validators cast votes for the change. An `UpdateOracle` message is then sent from one of the bridge executors. This message invokes the `ApplyOracleUpdate` function, which calls `UpdateOracle`. In this function, the incoming votes are validated, and if deemed valid, the Oracle prices are updated.

The main vote validation occurs in the `ValidateVoteExtensions` function, which performs three critical steps:

1. Calculates the total voting power of all validators in the current validator set.
2. Sums the voting power of the validators who voted on the change.
3. Verifies that the voting power of the validators who voted meets or exceeds the threshold of `(2/3 + 1)` of the total voting power.

https://github.com/initia-labs/OPinit/blob/7da22a78f367cb25960ed9c536500c2754bd5d06/x/opchild/l2connect/utils.go#L20-L124

```
func ValidateVoteExtensions(
        ctx sdk.Context,
        valStore ValidatorStore,
        height int64,
        chainID string,
        extCommit cometabci.ExtendedCommitInfo,
) error {
        // ...

        var (
                // Total voting power of all validators in current
validator store.
                totalVP int64
                // Total voting power of all validators that submitted
valid vote extensions.
                sumVP int64
        )

        totalBondedTokens, err := valStore.TotalBondedTokens(ctx)
        if err != nil {
                return err
        }
        totalVP = totalBondedTokens.Int64()

        for _, vote := range extCommit.Votes {
                valConsAddr := sdk.ConsAddress(vote.Validator.Address)
                power, err := valStore.GetPowerByConsAddr(ctx,
valConsAddr)
                if err != nil {
                        // return fmt.Errorf("failed to get validator %X
power: %w", valConsAddr, err)

                        // use only current validator set, so ignore if
the validator of the vote is not in the set.
                        continue
                }

                // snip... (unrelated code)

                sumVP += power.Int64()
        }

        // This check is probably unnecessary, but better safe than
sorry.
        if totalVP <= 0 {
                return fmt.Errorf("total voting power must be positive,
got: %d", totalVP)
        }

        // If the sum of the voting power has not reached (2/3 + 1) we
need to error.
        if requiredVP := ((totalVP * 2) / 3) + 1; sumVP < requiredVP {
                return fmt.Errorf(
                        "insufficient cumulative voting power received to
verify vote extensions; got: %d, expected: >=%d",
                        sumVP, requiredVP,
                )
        }

        return nil
}
```

As shown above, it just loops over the votes and adds the voting power of each entry, without validating if that validator is a duplicate entry.

This allows a malicious validator to solely change the oracle prices without the need of other validators' votes.

Recommended mitigation steps

Refactor `ValidateVoteExtensions` to throw an error if a duplicate validator is encountered.

```
        func ValidateVoteExtensions(
                ctx sdk.Context,
                valStore ValidatorStore,
                height int64,
                chainID string,
                extCommit cometabci.ExtendedCommitInfo,
        ) error {
                // ...

                totalBondedTokens, err := valStore.TotalBondedTokens(ctx)
                if err != nil {
                        return err
                }
                totalVP = totalBondedTokens.Int64()

+               seenValidators := make(map[string]bool)

                for _, vote := range extCommit.Votes {
+                       if seenValidators[vote.Validator.String()] {
+                               return fmt.Errorf("duplicate vote for
validator %s", vote.Validator.String())
+                       }
+                       seenValidators[vote.Validator.String()] = true

                        // ...
                }

                // ...
        }
```

beer-1 (Initia) confirmed and commented:

Thanks for your report, I will add this fix.

a_kalout (warden) commented:

I could see this as a high-severity issue. One malicious validator could change the Oracle prices, and unlike malicious propers/executors, there's no way to "catch" this. The way I'm looking at this is anyone can become a validator, and the above report allows any validator to solely update the Oracle prices; which clearly breaks the "quorum" logic for updating price here. This leaves us with likelihood arguably medium/high and a clear high impact.

LSDan (judge) commented:

I would agree if there was a follow-through impact reported that showed the Oracle price causing a direct loss of funds. The reporting warden did not take it to that level, so as reported, the impact is that the Oracle price is wrong, full stop. The follow on impacts are handwavy hypotheticals, making this a medium.

[M-02] Users can DOS the token deposits for any future bridges

Submitted by 0xAlix2

Users can create a bridge on L1 that points to L2 chains, which allows them to transfer funds between the 2 chains. When creating a bridge an incremental ID is generated for that bridge, and saved. To deposit coins to an L2, users have to call `InitiateTokenDeposit` on the L1.

https://github.com/initia-labs/OPinit/blob/7da22a78f367cb25960ed9c536500c2754bd5d06/x/ophost/keeper/msg_server.go#L222-L274.

This generates an L1 sequence:

```
l1Sequence, err := ms.IncreaseNextL1Sequence(ctx, bridgeId)
if err != nil {
        return nil, err
}
```

This sequence gets attached to the L2 message, when `FinalizeTokenDeposit` gets called, and gets validated on the L2, to ensure that messages are in order.

https://github.com/initia-labs/OPinit/blob/7da22a78f367cb25960ed9c536500c2754bd5d06/x/opchild/keeper/msg_server.go#L372-L487

```
finalizedL1Sequence, err := ms.GetNextL1Sequence(ctx)
if err != nil {
        return nil, err
}

if req.Sequence < finalizedL1Sequence {
        // No op instead of returning an error
        return &types.MsgFinalizeTokenDepositResponse{Result:
types.NOOP}, nil
} else if req.Sequence > finalizedL1Sequence {
        return nil, types.ErrInvalidSequence
}
```

This is all good for now. However, the issue is that the `InitiateTokenDeposit` function doesn't check if the provided input bridge ID is created before processing the message.

This allows a malicious user to initial a deposit on a non-existent bridge ID, which increments the `l1Sequence` to 2, the deposit sequence will be 1. This message won't be processed and sent to the L2, because simply the bridge/L2 doesn't exist.

Later, when a L2 and a bridge are created for that bridge ID (remember it's incremental), any deposits initiated from the L1 will fail, because the sequence number will be `> 1`. On L2, `finalizedL1Sequence` will be 1, which forces the following validation to fail:

https://github.com/initia-labs/OPinit/blob/7da22a78f367cb25960ed9c536500c2754bd5d06/x/opchild/keeper/msg_server.go#L393-L395.

```go
if req.Sequence > finalizedL1Sequence {
	return nil, types.ErrInvalidSequence
}
```

This DOSes the whole deposit functionality for any future bridges.

Proof of Concept

Add the following test in `x/ophost/keeper/msg_server_test.go`:

```go
func Test_InitiateTokenDeposit_NonExistBridge(t *testing.T) {
	ctx, input := createDefaultTestInput(t)

	ms := keeper.NewMsgServerImpl(input.OPHostKeeper)
	config := types.BridgeConfig{
		Proposer:              addrsStr[0],
		Challenger:            addrsStr[1],
		SubmissionInterval:    time.Second * 10,
		FinalizationPeriod:    time.Second * 60,
		SubmissionStartHeight: 1,
		Metadata:              []byte{1, 2, 3},
		BatchInfo:             types.BatchInfo{Submitter:
addrsStr[0], ChainType: types.BatchInfo_INITIA},
	}
	createRes, err := ms.CreateBridge(ctx,
types.NewMsgCreateBridge(addrsStr[0], config))
	require.NoError(t, err)
```

```
        require.Equal(t, uint64(1), createRes.BridgeId) // Bridge with ID
1 is created

        amount := sdk.NewCoin(sdk.DefaultBondDenom, math.NewInt(100))
        input.Faucet.Fund(ctx, addrs[1], amount)
        _, err = ms.InitiateTokenDeposit(
                ctx,
                types.NewMsgInitiateTokenDeposit(addrsStr[1], 100,
"l2_addr", amount, []byte("messages")), // Deposit to non-exist bridge,
ID 100
        )
        require.NoError(t, err)
        require.True(t, input.BankKeeper.GetBalance(ctx, addrs[1],
sdk.DefaultBondDenom).IsZero())
        // Bridge with ID 100 that doesn't exist, holds some amount
        require.Equal(t, amount, input.BankKeeper.GetBalance(ctx,
types.BridgeAddress(100), sdk.DefaultBondDenom))
}
```

Recommended mitigation steps

```
        func (ms MsgServer) InitiateTokenDeposit(ctx context.Context, req
*types.MsgInitiateTokenDeposit) (*types.MsgInitiateTokenDepositResponse,
error) {
                // ...

                coin := req.Amount
                bridgeId := req.BridgeId
                l1Sequence, err := ms.IncreaseNextL1Sequence(ctx,
bridgeId)
                if err != nil {
                        return nil, err
                }

+               if nextBridgeId, err := ms.GetNextBridgeId(ctx); err !=
nil {
+                       return nil, err
+               } else if bridgeId >= nextBridgeId {
+                       return nil, types.ErrBridgeNotFound
+               }

                // transfer only positive amount
                if coin.IsPositive() {
                        // send the funds to bridge address
                        bridgeAddr := types.BridgeAddress(bridgeId)
                        if err := ms.bankKeeper.SendCoins(ctx, sender,
bridgeAddr, sdk.NewCoins(coin)); err != nil {
                                return nil, err
                        }
                }

                // ...
        }
```

[beer-1 (Initia) confirmed, but disagreed with severity and commented](#):

Thanks for your finding! I have looked your descriptions and seems the problem exists, but the effect is not that serious actually.

In your report,

any deposits initiated from the L1 will fail, because the sequence number will be `> 1`. On L2, `finalizedL1Sequence` will be 1, which forces the following validation to fail:

https://github.com/initia-labs/OPinit/blob/7da22a78f367cb25960ed9c536500c2754bd5d06/x/opchild/keeper/msg_server.go#L393-L395.

but, actually, all these deposits still can be relayed. L1 sequence is increased before create bridge, but still bridge bot can relay it in sequence order and on l2 the sequence will be increased in order without problem.

Mitigation here.

https://github.com/initia-labs/OPinit/blob/bb31a386b6c643efc6b3f79db2c2abc3a6ce4b7a/x/ophost/keeper/msg_server.go#L229-L233

a_kalout (warden) commented:

@beer-1 - the message would be relayed with the wrong L1 sequence. The following is a scenario on the host chain. This is where a malicious actor performs a deposit on a non-existent bridge, which increases the next L1 sequence of that bridge to 2, then when the bridge is created and a legitimate deposit is done, it will have an L2 sequence of 2.

▶ Details

The executor won't relay the malicious deposit (which makes sense, as the bridge didn't even exist at that point). However, it will relay the legitimate deposit, but while passing the L1 sequence as 2 (which is emitted in the legitimate deposit on the host) here.

On the child side, the following will be run and reverted because of this, where `finalizedL1Sequence = 1` and `req.Sequence = 2`:

```
func Test_MsgServer_FinalizeDeposit(t *testing.T) {
        ctx, input := createDefaultTestInput(t)
        ms := keeper.NewMsgServerImpl(&input.OPChildKeeper)

        info := types.BridgeInfo{
                BridgeId:   1,
                BridgeAddr: addrsStr[1],
                L1ChainId:  "test-chain-id",
                L1ClientId: "test-client-id",
                BridgeConfig: ophosttypes.BridgeConfig{
                        Challenger: addrsStr[2],
                        Proposer:   addrsStr[3],
                        BatchInfo: ophosttypes.BatchInfo{
                                Submitter: addrsStr[4],
                                ChainType: ophosttypes.BatchInfo_INITIA,
                        },
                        SubmissionInterval:    time.Minute,
                        FinalizationPeriod:    time.Hour,
                        SubmissionStartHeight: 1,
                        Metadata:              []byte("metadata"),
                },
        }

        _, err := ms.SetBridgeInfo(ctx,
types.NewMsgSetBridgeInfo(addrsStr[0], info))
        require.NoError(t, err)

        baseDenom := "test_token"
        denom := ophosttypes.L2Denom(1, baseDenom)

        // @audit => Finalize deposit is called while passing sequence as
2 => reverts
        _, err = ms.FinalizeTokenDeposit(ctx,
types.NewMsgFinalizeTokenDeposit(addrsStr[0], "anyformataddr",
addrsStr[1], sdk.NewCoin(denom, math.NewInt(100)), 2, 1, "test/token",
nil))
        require.Error(t, err)
}
```

3docSec (warden) commented:

We believe that this situation would be only temporary because the executor has `disable_auto_set_l1_height` and `l1_start_height` settings which can be set by the admin operating the executor (`executor.go#L207C1-L209C11`) to include early deposit messages if any:

```
File: executor.go
    if ex.host.Node().GetSyncedHeight() == 0 {
        // use l1 start height from the config if auto set is disabled
        if ex.cfg.DisableAutoSetL1Height {
            l1StartHeight = ex.cfg.L1StartHeight
        } else {
```

Once this is done, nothing prevents early messages from being processed, then unblocking the bridge.
a_kalout (warden) commented:

The above comment is invalid; we can see that the returned `l1StartHeight` from the attached function `getNodeStartHeights` will only be used when initializing the node. You can follow that variable drilling, and we'll see that it is only being used here and is set as the synced height, this has nothing to do with the L1 sequence.

For clearer evidence, you can simply look at `deposit` which "catches" deposit messages from the host; which builds the finalize token deposit message. It is using the sequence emitted from the host events. In other words, the only source of truth of L2's L1 sequence is the host's sequence, which is messed up because of what's mentioned in this report.

As a result, no this is not temporary, because setting the L1 start height for the executor has nothing to do with the L1 sequence.

3docSec (warden) commented:

This finding's impact (bridge dos) is based on the fact that a "L1 sequence = 2" deposit cannot be processed before the "L1 sequence = 1" deposit.

My comment above shows how the "L1 sequence = 1" can be correctly processed despite being submitted before bridge creation; hence, mitigating the impact from a "permanent" loss to a "temporary" loss that can be mitigated by configuration.

LSDan (judge) commented:

The impact, while real, can be mitigated reasonably easily, so high risk is a stretch.

[M-03] Events emitted by L2 bridge hooks are discarded

Submitted by RadiantLabs

https://github.com/initia-labs/OPinit/blob/8119243edd00729124410dcd3cde5dc15b8c2fb8/x/opchild/keeper/deposit.go#L61

Finding description and impact

The L1 deposit function allows a depositor to send a signed payload to be executed atomically with the deposited tokens, via bridge hooks.

These are executed in L2 by routing messages to the default message handler from the app's router:

```
File: deposit.go
    for _, msg := range tx.GetMsgs() {
        handler := k.router.Handler(msg)
        if handler == nil {
            reason = fmt.Sprintf("Unrecognized Msg type: %s", sdk.MsgTypeURL(msg))
            return
        }

        _, err = handler(cacheCtx, msg)
        if err != nil {
            reason = fmt.Sprintf("Failed to execute Msg: %s", err)
            return
        }
    }
```

From the above code, we can see that the `sdk.Result` returned by the `handler()` call is discarded. This is incorrect because the returned `sdk.Result` is where the events emitted by `handler` are stored, because in the `handler` call the `EventManager` passed with `cacheCtx` is immediately discarded:

```
File: ../../../../../go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.50.9/baseapp/msg_service_router.go
    msr.routes[requestTypeName] = func(ctx sdk.Context, msg sdk.Msg) (*sdk.Result, error) {
        ctx = ctx.WithEventManager(sdk.NewEventManager())
```

Also by comparison, we can see that another place that similarly handles messages directly (the `MsgExecuteMessages` handler), does copy events over to the parent context:

```
File: opchild/keeper/msg_server.go
        res, err = handler(cacheCtx, msg)
        if err != nil {
            return nil, err
        }

```

```
            events = append(events, res.GetEvents()...)
    }

    writeCache()

    // TODO - merge events of MsgExecuteMessages itself
    sdkCtx.EventManager().EmitEvents(events)
```

As a consequence, events emitted by L2 hooks are discarded and not propagated to the transactions, and cannot be picked up by apps, and most importantly, bots.

Proof of Concept

In the below PoC we show the worst case scenario, where a withdrawal is initiated through hooks, and because the `EventTypeInitiateTokenWithdrawal` event is what bridges tokens to L1, tokens are permanently locked in the bridge.

This is a reasonable scenario where a user could, for example, bridge from L1 gas tokens that are necessary to initiate in L2 a withdrawal of a different token.

The test can be run if added to the `OPinit/x/opchild/keeper/msg_server_test.go` test file.

```
func Test_MsgServer_Deposit_HookNoEvents(t *testing.T) {
        ctx, input := createDefaultTestInput(t)
        ms := keeper.NewMsgServerImpl(&input.OPChildKeeper)

        bz := sha3.Sum256([]byte("test_token"))
        denom := "l2/" + hex.EncodeToString(bz[:])

        require.Equal(t, math.ZeroInt(), input.BankKeeper.GetBalance(ctx,
addrs[1], denom).Amount)

        // empty deposit to create account
        priv, _, addr := keyPubAddr()
        msg := types.NewMsgFinalizeTokenDeposit(addrsStr[0], addrsStr[1],
addr.String(), sdk.NewCoin(denom, math.ZeroInt()), 1, 1, "test_token",
nil)
        _, err := ms.FinalizeTokenDeposit(ctx, msg)
        require.NoError(t, err)

        // create hook data
        acc := input.AccountKeeper.GetAccount(ctx, addr)
        require.NotNil(t, acc)
        privs, accNums, accSeqs := []cryptotypes.PrivKey{priv},
[]uint64{acc.GetAccountNumber()}, []uint64{0}
```

```
        from, _ := input.AccountKeeper.AddressCodec().BytesToString(addr)
        to, _ :=
input.AccountKeeper.AddressCodec().BytesToString(addrs[2])

        signedTxBz, err := input.EncodingConfig.TxConfig.TxEncoder()
(generateTestTx(
                t, input,
                []sdk.Msg{
                        types.NewMsgInitiateTokenWithdrawal(from, to,
sdk.NewCoin(denom, math.NewInt(50))),
                },
                privs, accNums, accSeqs,
sdk.UnwrapSDKContext(ctx).ChainID(),
        ))
        require.NoError(t, err)

        // valid deposit
        ctx =
sdk.UnwrapSDKContext(ctx).WithEventManager(sdk.NewEventManager())
        msg = types.NewMsgFinalizeTokenDeposit(addrsStr[0], addrsStr[1],
addr.String(), sdk.NewCoin(denom, math.NewInt(100)), 2, 1, "test_token",
signedTxBz)
        _, err = ms.FinalizeTokenDeposit(ctx, msg)
        require.NoError(t, err)

        withdrawalEventFound := false
        for _, event := range
sdk.UnwrapSDKContext(ctx).EventManager().Events() {
                if event.Type == types.EventTypeInitiateTokenWithdrawal {
                        withdrawalEventFound = true
                }
        }

        // but no event was emitted by hooks
        require.False(t, withdrawalEventFound)
}
```

Recommended Mitigation Steps

Consider adding manual emission of generated events, as done for `MsgExecuteMessages`.

beer-1 (Initia) confirmed, but disagreed with severity and commented:

Seems the problem exists and events are dropped. We will do patch to emit the event. However, would like to lower severity, as events are not that critical part (not consensus) and we can re-index the chain after fix.

Mitigation here.
LSDan (judge) commented:

Downgrading to medium, but given that the impact is limited to events not being emitted, I can also see this as a low and am open to reasoned arguments during Post Judging QA.

3docSec (warden) commented:

For this codebase, lost events mean lost funds, because L2 events, and in particular the one proven to be lost in the PoC, are the one vector that bridges funds to L1.

In other words, a lost event means tokens burned in L2 and not ported to L1 `->` permanently lost funds. We believe High is appropriate.

1. On L2, user initiates a withdrawal by sending the `MsgInitiateTokenWithdrawal` message whose handling logic is here
2. The handling logic burns the L2 tokens and emits an `InitiateTokenWithdrawal` event, and that's where the message handling stops
3. The events are picked up by a bot and update the L2 state with the withdrawal
4. Funds can be rescued on L1 via the `MsgFinalizeTokenWithdrawal` message that grant them after verifying the withdrawal from L2 state.

If the event emitted at point 2 is lost, point 3 is not executed, and point 4 can never be executed because the withdrawal is not found in the L2 state tree, so funds are lost.

LSDan (judge) commented:

I agree with this assessment, but these are side effects that require a bunch of things to line up. The attack vector is not direct. Per the supreme court decision, the report does show "broader level impacts". However, they are medium in nature and fairly unlikely in practice (which is why I said I could be potentially talked into low risk).

[M-04] Using a timestamp as a key for processed messages leads to key collisions in the database

Submitted by RadiantLabs

https://github.com/initia-labs/opinit-bots/blob/640649b97cbfa5782925b7dc8c0b62b8fa5367f6/executor/batchsubmitter/batch.go#L147

https://github.com/initia-labs/opinit-bots/blob/640649b97cbfa5782925b7dc8c0b62b8fa5367f6/executor/batchsubmitter/batch.go#L167

Finding description and impact

The executor's `MsgsToProcessedMsgs()` converts messages to processed msgs:

```
276: // MsgsToProcessedMsgs converts msgs to processed msgs.
277: // It splits msgs into chunks of 5 msgs and creates processed msgs
for each chunk.
278: func MsgsToProcessedMsgs(queues map[string][]sdk.Msg)
[]btypes.ProcessedMsgs {
279:     res := make([]btypes.ProcessedMsgs, 0)
280:     for sender := range queues {
281:         msgs := queues[sender]
282:         for i := 0; i < len(msgs); i += 5 {
283:             end := i + 5
284:             if end > len(msgs) {
285:                 end = len(msgs)
286:             }
287:
288:             res = append(res, btypes.ProcessedMsgs{
289:                 Sender:    sender,
290:                 Msgs:      slices.Clone(msgs[i:end]),
291:                 Timestamp: types.CurrentNanoTimestamp(),
292:                 Save:      true,
293:             })
294:         }
295:     }
296:     return res
297: }
```

Messages are put into batches which will later get broadcasted as a single Cosmos SDK transaction. Additionally, processed messages are persisted to storage to ensure the executor can resume where it left off in case it restarts.

The issue is that the `Timestamp` (in nanoseconds) in line `291` (and the other instances linked to the submission) is very likely shared between batches, which results in key collisions in the database, as the `Key()` is based on the `Timestamp`.

`opinit-bots/node/broadcaster/types/db.go::Key()`

```
70: func (p ProcessedMsgs) Key() []byte {
71:     return
prefixedProcessedMsgs(types.MustInt64ToUint64(p.Timestamp))
72: }
```

This leads to wrongly deleting processed messages in the database,

`opinit-bots/node/broadcaster/db.go::DeleteProcessedMsgs()`

```go
// DeleteProcessedMsgs deletes processed messages
func DeleteProcessedMsgs(db types.BasicDB, processedMsgs btypes.ProcessedMsgs) error {
    return db.Delete(processedMsgs.Key())
}
```

Or, overwriting processed messages:

[opinit-bots/node/broadcaster/db.go::SaveProcessedMsgsBatch()]

```go
// SaveProcessedMsgsBatch saves all processed messages in the batch
func SaveProcessedMsgsBatch(db types.BasicDB, cdc codec.Codec, processedMsgsBatch []btypes.ProcessedMsgs) error {
    for _, processedMsgs := range processedMsgsBatch {
        if !processedMsgs.Save {
            continue
        }

        data, err := processedMsgs.Value(cdc)
        if err != nil {
            return err
        }

        err = db.Set(processedMsgs.Key(), data)
        if err != nil {
            return err
        }
    }
    return nil
}
```

Similarly, when [broadcasting a Cosmos transaction to the host/child node], the timestamp is also used as the key for `PendingTxInfo` when [storing it in the db for eventual restarts].

Ultimately, this leads to scenarios where after an executor restart, it will not resume correctly and misses to broadcast critical Cosmos transactions (e.g., finalizing a `L1->L2` token deposit).

Proof of Concept

We prepared a quick PoC that demonstrates how `MsgsToProcessedMsgs()` constructs batches of messages (`ProcessedMsgs`) with the same timestamp.

```go
func TestEndBlockHandlerMultipleBatches(t *testing.T) {
    childCodec, _, _ := childprovider.GetCodec("init")
```

```
        child := NewMockChild(nil, childCodec, "sender0", "sender1", 1)
        db, err := db.NewMemDB()
        defer func() {
                require.NoError(t, db.Close())
        }()
        hostdb := db.WithPrefix([]byte("test_host"))
        require.NoError(t, err)
        hostNode := node.NewTestNode(nodetypes.NodeConfig{}, hostdb, nil,
nil, nil, nil)

        h := Host{
                BaseHost: hostprovider.NewTestBaseHost(0, hostNode,
ophosttypes.QueryBridgeResponse{}, nodetypes.NodeConfig{}, nil),
                child:    child,
                stage:    hostdb.NewStage(),
        }

        msgQueue := h.GetMsgQueue()
        require.Empty(t, msgQueue["sender0"])

        // append a bunch of messages to the msg queue
        var numMessages = 100

        for i := 0; i < numMessages; i++ {
                h.AppendMsgQueue(&opchildtypes.MsgUpdateOracle{},
"sender0")
        }

        msgQueue = h.GetMsgQueue()
        require.Len(t, msgQueue["sender0"], numMessages)


h.AppendProcessedMsgs(broadcaster.MsgsToProcessedMsgs(h.GetMsgQueue())...
)

        msgs := h.GetProcessedMsgs()
        require.Len(t, msgs, 20) // 100 messages / 5 messages per batch =
20 batches

        // Verify that at least one of the batches has the same timestamp
        for i := 0; i < len(msgs)-1; i++ {
                if msgs[i].Timestamp == msgs[i+1].Timestamp {
                        // assert that the db key is the same
                        require.Equal(t, msgs[i].Key(), msgs[i+1].Key(),
fmt.Sprintf("DB key do not match - %s == %s", msgs[i].Key(),
msgs[i+1].Key()))

                        require.Fail(t, fmt.Sprintf("timestamps of batch
%d and %d are the same. %d == %d", i, i+1, msgs[i].Timestamp,
msgs[i+1].Timestamp))
                }
        }
}
```

Recommended mitigation steps

Consider using a truly unique identifier instead of the timestamp to avoid key collisions in the database.

[hoon (Initia) commented]:

There is an issue on mac approximating time in microseconds. It is occasionally possible in such cases.

[beer-1 (Initia) confirmed and commented]:

Mitigation [here].

[M-05] L2 hooks don't execute `ValidateBasic` on provided messages

Submitted by [RadiantLabs]

The L1 deposit function allows a depositor to send a signed payload to be executed atomically with the deposited tokens, via bridge hooks.

Within this signed payload, Cosmos Messages are executed one by one via the following steps:

- Transaction decoder
- Ante handler
- Message handler for each of the messages

```
File: deposit.go
        for _, msg := range tx.GetMsgs() {
                handler := k.router.Handler(msg)
                if handler == nil {
                        reason = fmt.Sprintf("Unrecognized Msg type: %s",
sdk.MsgTypeURL(msg))
                        return
                }

                _, err = handler(cacheCtx, msg)
                if err != nil {
                        reason = fmt.Sprintf("Failed to execute Msg: %s", err)
                        return
                }
        }
```

In this `msg` loop, we miss a step that is commonly executed by `BaseApp` on transactions submitted via RPC:

```
File: baseapp.go
func (app *BaseApp) runTx(mode execMode, txBytes []byte) (gInfo sdk.GasInfo, result *sdk.Result, anteEvents []abci.Event, err error) {
---
        tx, err := app.txDecoder(txBytes)
        if err != nil {
                return sdk.GasInfo{}, nil, nil, err
        }

        msgs := tx.GetMsgs()
        if err := validateBasicTxMsgs(msgs); err != nil {
                return sdk.GasInfo{}, nil, nil, err
        }
---
        if app.anteHandler != nil {
---
        }
---
        // Attempt to execute all messages and only update state if all messages pass
        // and we're in DeliverTx. Note, runMsgs will never return a reference to a
        // Result if any single message fails or does not have a registered Handler.
        msgsV2, err := tx.GetMsgsV2()
        if err == nil {
                result, err = app.runMsgs(runMsgCtx, msgs, msgsV2, mode)
        }
---
}
```

This `validateBasicTxMsgs` calls `ValidateBasic` on all messages that implement it:

```
func validateBasicTxMsgs(msgs []sdk.Msg) error {
        if len(msgs) == 0 {
                return errorsmod.Wrap(sdkerrors.ErrInvalidRequest, "must
contain at least one message")
        }

        for _, msg := range msgs {
                m, ok := msg.(sdk.HasValidateBasic)
                if !ok {
                        continue
                }

                if err := m.ValidateBasic(); err != nil {
                        return err
                }
        }

        return nil
}
```

This logic is missing from L2 hooks, meaning that messages that do have a `ValidateBasic` method will have this called bypassed as opposed to the situation of the same message submitted through RPC call.

This call can be reasonably executed by the `ante` handler. However, if we take as example `minimove`, this isn't the case, because `NewValidateBasicDecorator` is included for the RPC ante handler (L90) and not the hook's (L111-115):

```
File: ante.go
040: func NewAnteHandler(options HandlerOptions) (sdk.AnteHandler, error)
{
---
085:    anteDecorators := []sdk.AnteDecorator{
086:
accnumante.NewAccountNumberDecorator(options.AccountKeeper),
087:            ante.NewSetUpContextDecorator(), // outermost
AnteDecorator. SetUpContext must be called first
088:
ante.NewExtensionOptionsDecorator(options.ExtensionOptionChecker),
089:            moveante.NewGasPricesDecorator(),
090:            ante.NewValidateBasicDecorator(),
091:            ante.NewTxTimeoutHeightDecorator(),
092:            ante.NewValidateMemoDecorator(options.AccountKeeper),
093:
ante.NewConsumeGasForTxSizeDecorator(options.AccountKeeper),
        ante.NewDeductFeeDecorator(options.AccountKeeper, options.BankKeeper, options.FeegrantKeeper, freeLaneFeeChecker),
        // SetPubKeyDecorator must be called before all signature verification decorators
        ante.NewSetPubKeyDecorator(options.AccountKeeper),
        ante.NewValidateSigCountDecorator(options.AccountKeeper),
        ante.NewSigGasConsumeDecorator(options.AccountKeeper, sigGasConsumer),
        sigverify.NewSigVerificationDecorator(options.AccountKeeper, options.SignModeHandler),
        ante.NewIncrementSequenceDecorator(options.AccountKeeper),
        ibcante.NewRedundantRelayDecorator(options.IBCkeeper),
        auctionante.NewAuctionDecorator(options.AuctionKeeper, options.TxEncoder, options.MevLane),
        opchildante.NewRedundantBridgeDecorator(options.OPChildKeeper),
    }

    return sdk.ChainAnteDecorators(anteDecorators...), nil
}

func CreateAnteHandlerForOPinit(ak ante.AccountKeeper, signModeHandler *txsigning.HandlerMap) sdk.AnteHandler {
    return sdk.ChainAnteDecorators(
        ante.NewSetPubKeyDecorator(ak),
        ante.NewValidateSigCountDecorator(ak),
        ante.NewSigGasConsumeDecorator(ak, ante.DefaultSigVerificationGasConsumer),
        ante.NewSigVerificationDecorator(ak, signModeHandler),
        ante.NewIncrementSequenceDecorator(ak),
    )
}
```

The consequence is that, when called through L2 hook transactions, `MsgServer` instances can execute messages that would not pass `ValidateBasic`. Depending on the specific message handling, this can lead to inconsistent operation or application panics.

Recommended Mitigation Steps

We recommend either:

- Implementing a similar `validateBasicTxMsgs()` function on the hook execution logic.
- Adding `ante.NewValidateBasicDecorator()` in the L2 app's hook `ante` handler

Links to affected code

- deposit.go#L46

beer-1 (Initia) confirmed and commented via Move audit, Issue F-7:

Mitigation here.

[M-06] Malicious proposer can DOS bridge withdrawals

Submitted by Ruhum

https://github.com/initia-labs/OPinit/blob/7da22a78f367cb25960ed9c536500c2754bd5d06/x/ophost/keeper/msg_server.go#L145

https://github.com/initia-labs/OPinit/blob/7da22a78f367cb25960ed9c536500c2754bd5d06/x/ophost/keeper/msg_server.go#L351-L365

Finding description and impact

Anybody can create an L2 and its bridge. They set their own proposer and challenger to handle the output proposals. If the proposer sets the L2 block number of an output proposal to the max value of a `uint64`, any subsequent output proposal will fail preventing users from withdrawing their funds.

Proof of Concept

In ophost's `ProposeOutput()` endpoint the module returns an error if the given L2 block number is smaller than the last proposed output:

```
                if l2BlockNumber <= lastOutputProposal.L2BlockNumber {
                        return nil,
types.ErrInvalidL2BlockNumber.Wrapf("last %d, got %d",
lastOutputProposal.L2BlockNumber, l2BlockNumber)
                }
```

By proposing an output with `l2BlockNumber = 2^64 - 1` any subsequent call will fail.

A bridge's admin, proposer, and challenger are not trusted users. Instead, the module allows the L1 governance account (`authority`), to take over the bridge in case the admin misbehaves, see `UpdateProposer()`. However, with this attack vector, the governance account cannot reverse the attack or rescue funds unless it deletes the malicious output before it is finalized, see `DeleteOutputProposal()`.

The governance account has to react in a matter of minutes or an hour (depending on the bridge's configuration).

Recommended mitigation steps

This will always be an issue as long as the L2 block number is an unverified input by the proposer. Maybe you can fully remove it and just depend on the output index that's tracked by the module.

beer-1 (Initia) acknowledged and commented:

We have logic to delete invalid output proposal during 7 days, and anyone can do challenge.

a_kalout (warden) commented:

This issue requires the proposer to be a malicious entity, which is very unlikely, but in case it is, any output can't be processed unless it "sits there" for some time; according to the sponsor, it'll be 7 days. In this period, the challenger should be able to catch and discard that, so the challenger is already there to solve such cases.

As a result, I believe this is a low-severity issue, specifically looking at the mitigation for such scenarios (malicious proposer) is already there.

LSDan (judge) commented:

I think that logic is sound, but if it was not, I would consider this high risk due to the impact and direct nature of the attack. Given the likelihood that the attack happens and is not caught, I agree with the wardens assessment of medium risk.

Low Risk and Non-Critical Issues

For this audit, 2 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **_karanel** received the top score from the judge.

The following wardens also submitted reports: Oxozovehe.

[01] `DeleteFutureFinalizedTrees` does not delete Nodes from the database

If a `WorkingTree` was initialized, then a few leaf nodes were inserted and saved, and the `WorkingTree` was also saved. If then the Working Tree was deleted, the Nodes would still exist in the database and take up space.

https://github.com/initia-labs/opinit-bots/blob/640649b97cbfa5782925b7dc8c0b62b8fa5367f6/merkle/db.go#L14-L23

```
// DeleteFutureFinalizedTrees deletes all finalized trees with sequence
number greater than or equal to fromSequence.
```

```
func DeleteFutureFinalizedTrees(db types.DB, fromSequence uint64) error {
	return
db.Iterate(dbtypes.AppendSplitter(merkletypes.FinalizedTreePrefix),
merkletypes.PrefixedFinalizedTreeKey(fromSequence), func(key, _ []byte)
(bool, error) {
		err := db.Delete(key)
		if err != nil {
			return true, err
		}
		return false, nil
	})
}
```

[02] `TestFillLeaves` test fails when it is expected to catch an error

In the range of test cases, each `tc` has an `expected` bool value which determines if the `fillLeaves()` would give an error or not. If `tc.expected` is set to `false`, `fillLeaves()` returns an error; however, before reaching the intended block where this error is caught, i.e., `require.Error(t, err)`, it encounters a `require.NoError(t, err)`. This would cause the test to fail when it was expected to pass since the testcase was pre-determined to fail.

https://github.com/initia-labs/opinit-bots/blob/640649b97cbfa5782925b7dc8c0b62b8fa5367f6/merkle/merkle_test.go#L108-L222

```
	for _, tc := range cases {
		t.Run(tc.title, func(t *testing.T) {
			err = m.InitializeWorkingTree(10, 1, 1)
			require.NoError(t, err)

			for i := uint64(0); i < tc.leaves; i++ {
				_, err := m.InsertLeaf([]byte("node"))
				require.NoError(t, err)
			}

@>			nodes, err := m.fillLeaves()
			// @@audit-qa unnecessary condition, test would
fail if err
@>			require.NoError(t, err)

@>			if tc.expected {
				require.NoError(t, err)
				require.Equal(t, tc.nodes, nodes)
			} else {
@>				require.Error(t, err)
```

```
                        }
                })
        }
```

[03] `TestInsertLeaf` has an incorrect assertion

In `TestInsertLeaf` when 4 nodes are inserted the last assertion is incorrect. It asserts the node with height `1`, `localNodeIndex` `1`, and data `hash34[:]`. It should have been height `0`, `localNodeIndex` `0`, and data `hash1234[:]` instead.

https://github.com/initia-labs/opinit-bots/blob/640649b97cbfa5782925b7dc8c0b62b8fa5367f6/merkle/merkle_test.go#L282-L309

```
        // 4 nodes
        hash34 := hashFn([]byte("node3"), []byte("node4"))
        hash1234 := hashFn(hash12[:], hash34[:])
        nodes, err = m.InsertLeaf([]byte("node4"))
        require.NoError(t, err)
        require.Len(t, m.workingTree.LastSiblings, 3)
        require.Equal(t, []byte("node4"), m.workingTree.LastSiblings[0])
        require.Equal(t, hash34[:], m.workingTree.LastSiblings[1])
        require.Equal(t, hash1234[:], m.workingTree.LastSiblings[2])
        require.Len(t, nodes, 3)
        require.Equal(t, merkletypes.Node{
                TreeIndex:      1,
                Height:         0,
                LocalNodeIndex: 3,
                Data:           []byte("node4"),
        }, nodes[0])
        require.Equal(t, merkletypes.Node{
                TreeIndex:      1,
                Height:         1,
                LocalNodeIndex: 1,
                Data:           hash34[:],
        }, nodes[1])
        // @@audit-qa incorrect node checked here
@>      require.Equal(t, merkletypes.Node{
                TreeIndex:      1,
                Height:         1,
                LocalNodeIndex: 1,
                Data:           hash34[:],
        }, nodes[1])
```

Disclosures

C4 is an open organization governed by participants in the community.

C4 audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit submissions are judged by a knowledgeable security researcher and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.