SHERLOCK

Security Review For Gurufin

Collaborative Audit Prepared For: Gurufin
Lead Security Expert(s): Bauchibred
iammxuse
Date Audited: September 22 - October 8, 2025

Introduction

Gurufin is building the next generation global payment infrastructure bridging fiat and Web3.

Scope

Repository: gurufinglobal/guru

Audited Commit: d0ea61d7eac0130afbca3a50fc71d900bb1ef4e0

Final Commit: c80a52d3b677550e7074b347c8dc2bb0d87ba087

Files:

- cmd/gurud/cmd/root.go
- cmd/gurud/config/chain_id.go
- cmd/gurud/config/config.go
- cmd/gurud/config/constants.go
- cmd/gurud/config/opendb.go
- cmd/gurud/main.go
- cmd/oracled/main.go
- gurud/activators.go
- gurud/ante/ante.go
- gurud/ante/cosmos_fees.go
- gurud/ante/cosmos_handler.go
- gurud/ante/evm_handler.go
- gurud/ante/handler_options.go
- gurud/ante/interfaces.go
- gurud/ante/validator_tx_fee.go
- gurud/app.go
- gurud/config.go
- gurud/eips/eips.go
- gurud/export.go
- gurud/genesis.go
- gurud/interfaces.go
- gurud/precompiles.go

- gurud/upgrades.go
- oralce/config/config.go
- oralce/daemon/daemon.go
- oralce/submiter/submitter.go
- oralce/subscriber/subscriber.go
- oralce/types/types.go
- oralce/worker/client.go
- oralce/worker/pool.go
- x/feemarket/client/cli/query.go
- x/feemarket/genesis.go
- x/feemarket/keeper/abci.go
- x/feemarket/keeper/eip1559.go
- x/feemarket/keeper/grpc_query.go
- x/feemarket/keeper/hooks.go
- x/feemarket/keeper/keeper.go
- x/feemarket/keeper/msg_server.go
- x/feemarket/keeper/params.go
- x/feemarket/module.go
- x/feemarket/types/codec.go
- x/feemarket/types/events.go
- x/feemarket/types/genesis.go
- x/feemarket/types/keys.go
- x/feemarket/types/msg.go
- x/feemarket/types/params.go
- x/feepolicy/client/cli/cli.go
- x/feepolicy/client/cli/query.go
- x/feepolicy/client/cli/tx.go
- x/feepolicy/genesis.go
- x/feepolicy/keeper/grpc_query.go
- x/feepolicy/keeper/keeper.go
- x/feepolicy/keeper/msg_server.go

- x/feepolicy/module.go
- x/feepolicy/types/codec.go
- x/feepolicy/types/discount.go
- x/feepolicy/types/errors.go
- x/feepolicy/types/events.go
- x/feepolicy/types/genesis.go
- x/feepolicy/types/interfaces.go
- x/feepolicy/types/keys.go
- x/feepolicy/types/msgs.go
- x/oracle/client/cli/query.go
- x/oracle/client/cli/tx.go
- x/oracle/genesis.go
- x/oracle/handler.go
- x/oracle/keeper/abci.go
- x/oracle/keeper/aggregate.go
- x/oracle/keeper/hooks.go
- x/oracle/keeper/keeper.go
- x/oracle/keeper/msg_server.go
- x/oracle/keeper/query_server.go
- x/oracle/module.go
- x/oracle/types/codec.go
- x/oracle/types/errors.go
- x/oracle/types/events.go
- x/oracle/types/expected_keepers.go
- x/oracle/types/genesis.go
- x/oracle/types/interfaces.go
- x/oracle/types/keys.go
- x/oracle/types/msgs.go
- x/oracle/types/oracle.go
- x/oracle/types/params.go

Repository: gurufinglobal/cosmos-sdk

Audited Commit: d5a86dea4e894544079a6ba35196dea8e09972ba

Final Commit: bffe0c02403156c78c92e137b7cf8cfa480bd6c5

Files:

- client/flags/flags.go
- store/types/gas.go
- x/distribution/abci.go
- x/distribution/autocli.go
- x/distribution/client/cli/query.go
- x/distribution/client/cli/tx.go
- x/distribution/client/common/common.go
- x/distribution/doc.go
- x/distribution/exported/exported.go
- x/distribution/keeper/abci.go
- x/distribution/keeper/alias_functions.go
- x/distribution/keeper/allocation.go
- x/distribution/keeper/delegation.go
- x/distribution/keeper/fee_pool.go
- x/distribution/keeper/genesis.go
- x/distribution/keeper/grpc_query.go
- x/distribution/keeper/hooks.go
- x/distribution/keeper/keeper.go
- x/distribution/keeper/migrations.go
- x/distribution/keeper/msg_server.go
- x/distribution/keeper/params.go
- x/distribution/keeper/store.go
- x/distribution/keeper/validator.go
- x/distribution/migrations/v1/types.go
- x/distribution/migrations/v2/helpers.go
- x/distribution/migrations/v2/store.go
- x/distribution/migrations/v3/json.go

- x/distribution/migrations/v3/migrate.go
- x/distribution/module.go
- x/distribution/simulation/decoder.go
- x/distribution/simulation/genesis.go
- x/distribution/simulation/msg_factory.go
- x/distribution/simulation/operations.go
- x/distribution/simulation/proposals.go
- x/distribution/types/codec.go
- x/distribution/types/delegator.go
- x/distribution/types/errors.go
- x/distribution/types/events.go
- x/distribution/types/expected_keepers.go
- x/distribution/types/fee_pool.go
- x/distribution/types/genesis.go
- x/distribution/types/keys.go
- x/distribution/types/msg.go
- x/distribution/types/params.go
- x/distribution/types/proposal.go
- x/distribution/types/querier.go
- x/distribution/types/query.go
- x/distribution/types/ratio.go
- x/distribution/types/validator.go

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
5	13	12

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue H-1: Distribution module's begin blocker can crash the node [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/144

Summary

Inside `AllocateTokens`, if `BurnCoins` or `SendCoinsFromModuleToAccount` fails, the entire ABCI process panics and the node halts instead of returning a failed transaction/block.

Vulnerability Details

First see `distribution/keeper/allocation.go::AllocateTokens()`:

```
func (k Keeper) AllocateTokens(ctx context.Context, totalPreviousPower int64,
↪  bondedVotes []abci.VoteInfo) error {
// ..snip

    // transfer collected fees to the distribution module account
    err := k.bankKeeper.SendCoinsFromModuleToModule(ctx, k.feeCollectorName,
    ↪  types.ModuleName, feesCollectedInt)
    if err != nil {
        return err
    }
    if len(feesCollectedInt) > 0 {
        ratio, err := k.GetRatio(sdk.UnwrapSDKContext(ctx))
        if err != nil {
            return err
        }


// ..snip

burnFee := k.CalculatePercentage(feesCollectedInt, ratio.Burn)
err = k.bankKeeper.BurnCoins(ctx, types.ModuleName, burnFee)
if err != nil {
    panic(err)   // ← upstream: return err
}

// ...snip

baseFee = k.CalculatePercentage(feesCollectedInt, ratio.Base)
err = k.bankKeeper.SendCoinsFromModuleToAccount(
         ctx, types.ModuleName, baseAddr, baseFee)
if err != nil {
    panic(err)   // ← upstream: return err
}
```

As seen, the error handling cases are inconsistent, making us "return" the errors in some cases and "panic" in the newly implemented guru addition of burning fees and also sending base fees to the base address.

Now the call is routed to `AllocateTokens()` from the `BeginBlocker()`, and the begin blocker expects `AllocateTokens` to signal failure via `return err` as is done in classic code so it can propagate the ABCI error up to CometBFT:

https://github.com/sherlock-audit/2025-09-gurufin-chain/blob/d63b56ad665dc01eeb38a7c5cf17b42219ca8cc1/cosmos-sdk/x/distribution/keeper/abci.go#L11-L41

```
func (k Keeper) BeginBlocker(ctx sdk.Context) error {
    start := telemetry.Now()
    defer telemetry.ModuleMeasureSince(types.ModuleName, start,
     ↪  telemetry.MetricKeyBeginBlocker)

    // determine the total power signing the block
    var previousTotalPower int64
    // determine the total power signing the block
    for _, voteInfo := range ctx.VoteInfos() {
        previousTotalPower += voteInfo.Validator.Power
    }

    // TODO this is Tendermint-dependent
    // ref https://github.com/cosmos/cosmos-sdk/issues/3095
    height := ctx.BlockHeight()
    if height > 1 {
|>      if err := k.AllocateTokens(ctx, previousTotalPower, ctx.VoteInfos()); err
↪   != nil {
            return err
        }

        // send whole coins from community pool to x/protocolpool if enabled
        if k.HasExternalCommunityPool() {
            if err := k.sendCommunityPoolToExternalPool(ctx); err != nil {
                return err
            }
        }
    }

    // record the proposer for when we pay out on the next block
    consAddr := sdk.ConsAddress(ctx.BlockHeader().ProposerAddress)
    return k.SetPreviousProposerConsAddr(ctx, consAddr)
}
```

```
if err := k.AllocateTokens(ctx, totalPower, ctx.VoteInfos()); err != nil {
    return err   // exported to DeliverBlock
}
```

However, because the Guru code panics instead, any runtime issue–insufficient balance, malformed coin, or anti-spam safeguard–breaks the invariant and tears down the consensus process.

Impact

Chain halt, breaking liveness as using `panic` as a error handler in a begin Blocker path would crash the node, which also goes against the standard as we are to return errors and let the application handle them.

Recommendation

Replace the two `panic(err)` statements with `return err`, mirroring the upstream SDK:

```
if err := k.bankKeeper.BurnCoins(ctx, types.ModuleName, burnFee); err != nil {
    return err
}

if err := k.bankKeeper.SendCoinsFromModuleToAccount(
        ctx, types.ModuleName, baseAddr, baseFee); err != nil {
    return err
}
```

This restores graceful error propagation: the block fails, the transaction is rejected, and the node continues operating instead of crashing.

Issue H-2: Fee granter is over charged for fees [RE-SOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/146

Summary

`DeductFeeDecorator` currently always looks up the fee-policy discount by the fee-payer's address. Now when a transaction uses fee-grant–i.e. another account actually pays the fees–the decorator still evaluates the discount table against the payer, not the granter who would pay for the tx.
So if the granter enjoys a lower fee policy, that benefit is ignored, forcing the granter to pay the full, undiscounted fee.

Vulnerability Details

First notice cosmos_fees.go::AnteHandle:

```
func (dfd DeductFeeDecorator) AnteHandle(ctx sdk.Context, tx sdk.Tx, simulate bool,
 ↪  next sdk.AnteHandler) (sdk.Context, error) {
    feeTx, ok := tx.(sdk.FeeTx)
    if !ok {
        return ctx, errorsmod.Wrap(sdkerrors.ErrTxDecode, "Tx must be a FeeTx")
    }

    if !simulate && ctx.BlockHeight() > 0 && feeTx.GetGas() == 0 {
        return ctx, errorsmod.Wrap(sdkerrors.ErrInvalidGasLimit, "must provide
         ↪  positive gas")
    }

    var (
        priority int64
        err      error
    )

|>  fee := feeTx.GetFee()
    if !simulate {
        fee, priority, err = dfd.txFeeChecker(ctx, tx)
        if err != nil {
            return ctx, err
        }
    }

    addrCodec := address.Bech32Codec{
        Bech32Prefix: sdk.GetConfig().GetBech32AccountAddrPrefix(),
    }
|>  feePayer, err := addrCodec.BytesToString(feeTx.FeePayer())
    if err != nil {
        return ctx, err
    }

|>  discount := dfd.feepolicyKeeper.GetDiscount(ctx, string(feePayer),
↪   tx.GetMsgs())//@audit

    // apply discounts
    var deductedFee sdk.Coins

    if discount.DiscountType == feepolicytypes.FeeDiscountTypePercent {
        for _, f := range fee {
            // type: "percent"
            // fee calculation: (100 - amount) % => if discount is 30%, then 70% of
            ↪  the fee is deducted
            deductedFee = deductedFee.Add(sdk.NewCoin(f.Denom,
            ↪  f.Amount.MulRaw(math.LegacyNewDec(100).Sub(discount.Amount).Truncat⌋
            ↪  eInt64()).QuoRaw(100)))
        }
    } else if discount.DiscountType == feepolicytypes.FeeDiscountTypeFixed {
        for _, f := range fee {
            // type: "fixed"
            // fee calculation: fixed amount
            deductedFee = deductedFee.Add(sdk.NewCoin(f.Denom,
            ↪  discount.Amount.TruncateInt()))
        }
    } else {
        // if no discount, deduct full fee
        deductedFee = fee
    }

    if err = dfd.checkDeductFee(ctx, tx, deductedFee); err != nil {
        return ctx, err
    }

    newCtx := ctx.WithPriority(priority)

    return next(newCtx, tx, simulate)

}
```

As seen the intention is that the discount should be tagged to the person paying the fee. Now later, in checkDeductFee, the coins are deducted from the granter when present:

https://github.com/sherlock-audit/2025-09-gurufin-chain/blob/d63b56ad665dc01eeb38a7c5cf17b42219ca8cc1/guru-v2/gurud/ante/cosmos_fees.go#L107-L161

```
func (dfd DeductFeeDecorator) checkDeductFee(ctx sdk.Context, sdkTx sdk.Tx, fee
↪   sdk.Coins) error {
```

```go
// ..snip

    feePayer := feeTx.FeePayer()
    feeGranter := feeTx.FeeGranter()
    deductFeesFrom := feePayer

    // if feegranter set deduct fee from feegranter account.
    // this works with only when feegrant enabled.
    if feeGranter != nil {
        feeGranterAddr := sdk.AccAddress(feeGranter)

        if dfd.feegrantKeeper == nil {
            return sdkerrors.ErrInvalidRequest.Wrap("fee grants are not enabled")
        } else if !bytes.Equal(feeGranterAddr, feePayer) {
            err := dfd.feegrantKeeper.UseGrantedFees(ctx, feeGranterAddr, feePayer,
            ↪  fee, sdkTx.GetMsgs())
            if err != nil {
                return errorsmod.Wrapf(err, "%s does not allow to pay fees for %s",
                ↪  feeGranter, feePayer)
            }
        }

|>      deductFeesFrom = feeGranterAddr
    }

    deductFeesFromAcc := dfd.accountKeeper.GetAccount(ctx, deductFeesFrom)
    if deductFeesFromAcc == nil {
        return sdkerrors.ErrUnknownAddress.Wrapf("fee payer address: %s does not
        ↪  exist", deductFeesFrom)
    }

    // deduct the fees
    if !fee.IsZero() {
        err := authante.DeductFees(dfd.bankKeeper, ctx, deductFeesFromAcc, fee)
        if err != nil {
            return err
        }
    }

    events := sdk.Events{
        sdk.NewEvent(
            sdk.EventTypeTx,
            sdk.NewAttribute(sdk.AttributeKeyFee, fee.String()),
            sdk.NewAttribute(sdk.AttributeKeyFeePayer,
            ↪  sdk.AccAddress(deductFeesFrom).String()),
        ),
    }
    ctx.EventManager().EmitEvents(events)

    return nil
```

```
}
```

So the algorithm is:

1. lookup discount by `feePayer`
2. compute fee
3. if `feeGranter != nil`, charge that amount to `feeGranterAddr`

This however can be unfair in some instances, for e.g

- Account A (payer) has no discount
- Account B (granter) has a 10 % discount

A broadcasts a tx, B pays fees via grant, the Decorator in this case fetches “no discount” (because keyed to A) and charges B the full amount, ignoring B’s 10 % entitlement.

Impact

Loss of funds for the fee granter, since the economic intent of fee-grant is broken: granters with favourable fee tiers over-pay every granted transaction.
Large relayer or dApp accounts that subsidise users lose funds unnecessarily and may end up refusing to grant.

Recommendation

When `feeGranter` is non-nil and different from `feePayer`, perform the discount lookup with the granter’s address.

Issue H-3: `fetchRawData` mis-uses max() and would stall data fetching indefinitely [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/148

Summary

fetchRawData tries to implement a back-off logic, but it instead sleeps for `max(retryDelay, config.RetryMaxDelaySec())`, since `max` returns the larger value, the delay is never smaller than RetryMaxDelaySec and grows without an upper bound.

Vulnerability Details

client.go::fetchRawData()

```
// fetchRawData retrieves bytes from an external endpoint with bounded retries.
func (hc *httpClient) fetchRawData(url string) ([]byte, error) {
    maxAttempts := max(1, config.RetryMaxAttempts())
    for attempt := 0; attempt < maxAttempts; attempt++ {
        if 0 < attempt {
            retryDelay := time.Duration(1<<(attempt-1)) * time.Second
            time.Sleep(max(retryDelay, config.RetryMaxDelaySec()))
        }
        // ..snip
    }
    //..snip
}
```

1. `retryDelay` doubles each attempt with `(1<<(attempt-1)`...
2. `max(a, b)` picks the larger of `retryDelay` and `RetryMaxDelaySec()`; if the config is, say, 30 s, every sleep is currently at least 30 s instead of at max 30 s.
3. There is no ceiling and with every retry the delay is doubled.

Impact

The intention of having bounded entries with `fetchRawData()` to an external endpoint is broken. Oracle now stalls for long periods when endpoints are flaky, missing price slots and breaking SLA.

Multiple workers would also be stuck in long sleep consume goroutines and delay shutdown.

Recommendation

Use `min` instead of `max` this way we correctly implement the exponential backoff.

Issue H-4: `UpdateModeratorAddress` is broken causing permanent moderator lock [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/152

Summary

keeper::UpdateModeratorAddress validates that a new moderator address differs from the current one but mistakenly stores `msg.ModeratorAddress` (the old address) instead of `msg.NewModeratorAddress` making it impossible to update the moderator address.

Vulnerability Details

First see the oracle module's UpdateModeratorAddress():

```
// UpdateModeratorAddress defines a method for updating the moderator address
func (k Keeper) UpdateModeratorAddress(c context.Context, msg
↪   *types.MsgUpdateModeratorAddress) (*types.MsgUpdateModeratorAddressResponse,
↪   error) {
    ctx := sdk.UnwrapSDKContext(c)

    currentModeratorAddress := k.GetModeratorAddress(ctx)

    if currentModeratorAddress != msg.ModeratorAddress {
        return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, "from address is
        ↪   different from current moderator address")
    }
    if currentModeratorAddress == "" {
        return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, "moderator address
        ↪   is not set")
    }
    if currentModeratorAddress == msg.NewModeratorAddress {
        return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, "new moderator
        ↪   address is same as current moderator address")
    }

|>  k.SetModeratorAddress(ctx, msg.ModeratorAddress)//@audit here we wrongly set
↪   the old address again

    ctx.EventManager().EmitEvent(
        sdk.NewEvent(
            types.EventTypeUpdateModeratorAddress,
|>          sdk.NewAttribute(types.AttributeKeyModeratorAddress,
↪   msg.ModeratorAddress),
        ),
    )
    return &types.MsgUpdateModeratorAddressResponse{}, nil
}
```

As seen, the transaction succeeds and emits events claiming success, yet state never changes, making the moderator role forever immutable.

Looking at the `MsgUpdateModeratorAddress` schema, we can see the below:

https://github.com/sherlock-audit/2025-09-gurufin-chain/blob/d63b56ad665dc01eeb38a7c5cf17b42219ca8cc1/guru-v2/x/oracle/types/tx.pb.go#L286-L290

```
// MsgUpdateModeratorAddress represents a message to update the moderator address
type MsgUpdateModeratorAddress struct {
    ModeratorAddress    string
    ↪  `protobuf:"bytes,1,opt,name=moderator_address,json=moderatorAddress,proto3"
    ↪  json:"moderator_address,omitempty"`
    NewModeratorAddress string `protobuf:"bytes,2,opt,name=new_moderator_address,js⌋
    ↪  on=newModeratorAddress,proto3" json:"new_moderator_address,omitempty"`
}
```

Whereas the current validation proves intent to switch to `msg.NewModeratorAddress`, the subsequent state write re-sets the old address.

Impact

Moderator rotation becomes impossible after genesis. Any attempted update silently fails, forever locking oracle administration to a single address.

Recommendation

Store and emit the **new** address:

```
// UpdateModeratorAddress defines a method for updating the moderator address
func (k Keeper) UpdateModeratorAddress(c context.Context, msg
↪  *types.MsgUpdateModeratorAddress) (*types.MsgUpdateModeratorAddressResponse,
↪  error) {
    ctx := sdk.UnwrapSDKContext(c)

    currentModeratorAddress := k.GetModeratorAddress(ctx)

    if currentModeratorAddress != msg.ModeratorAddress {
        return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, "from address is
        ↪  different from current moderator address")
    }
    if currentModeratorAddress == "" {
        return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, "moderator address
         ↪  is not set")
    }
    if currentModeratorAddress == msg.NewModeratorAddress {
        return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, "new moderator
         ↪  address is same as current moderator address")
    }

-   k.SetModeratorAddress(ctx, msg.ModeratorAddress)wrongly set the old address
 ↪  again
+   k.SetModeratorAddress(ctx, msg.NewModeratorAddress)

    ctx.EventManager().EmitEvent(
        sdk.NewEvent(
            types.EventTypeUpdateModeratorAddress,
-           sdk.NewAttribute(types.AttributeKeyModeratorAddress,
 ↪  msg.ModeratorAddress),
+           sdk.NewAttribute(types.AttributeKeyModeratorAddress,
 ↪  msg.NewModeratorAddress),
        ),
    )

    return &types.MsgUpdateModeratorAddressResponse{}, nil
}
```

Issue H-5: Oracle codec does not register `MsgSubmitOracleData` and `MsgUpdateModeratorAddress` [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/166

Summary

The oracle module defines multiple messages including `MsgSubmitOracleData` and `MsgUpdateModeratorAddress` together with full handlers, CLI wiring and tests.
However these specific two messages are never registered in x/oracle/types/codec.go, now since the Cosmos SDK decodes inbound transactions by looking up the concrete type URL or Amino name in this registry, any transaction containing either message fails with "unknown message type". As a result, submitting oracle data or rotating the moderator address is impossible on-chain.

Vulnerability Details

oracle::RegisterLegacyAminoCodec and oracle::RegisterInterfaces register only two message types:

```
// RegisterLegacyAminoCodec registers the necessary x/oracle interfaces and
 ↪  concrete types
// on the provided LegacyAmino codec. These types are used for Amino JSON
 ↪  serialization.
func RegisterLegacyAminoCodec(cdc *codec.LegacyAmino) {
    cdc.RegisterConcrete(&MsgRegisterOracleRequestDoc{},
     ↪  "oracle/RegisterOracleRequestDoc", nil)
    cdc.RegisterConcrete(&MsgUpdateOracleRequestDoc{},
     ↪  "oracle/UpdateOracleRequestDoc", nil)
}

// RegisterInterfaces registers the x/oracle interfaces types with the interface
 ↪  registry
func RegisterInterfaces(registry cdctypes.InterfaceRegistry) {
    registry.RegisterImplementations((*sdk.Msg)(nil),
        &MsgRegisterOracleRequestDoc{},
        &MsgUpdateOracleRequestDoc{},
    )

    msgservice.RegisterMsgServiceDesc(registry, &_Msg_serviceDesc)
}
```

Yet `oracle::MsgSubmitOracleData` and `oracle::MsgUpdateModeratorAddress` are fully implemented:

x/oracle/types/tx.pb.go#L209-L215

```
// ..snip
// MsgSubmitOracleData represents a message to submit oracle data
type MsgSubmitOracleData struct {
    AuthorityAddress string
     ↪  `protobuf:"bytes,1,opt,name=authority_address,json=authorityAddress,proto3"
     ↪  json:"authority_address,omitempty"`
    // The oracle data set to be submitted, containing the raw data and metadata
    DataSet *SubmitDataSet
     ↪  `protobuf:"bytes,2,opt,name=data_set,json=dataSet,proto3"
     ↪  json:"data_set,omitempty"`
}
    // ..snip

// MsgUpdateModeratorAddress represents a message to update the moderator address
type MsgUpdateModeratorAddress struct {
    ModeratorAddress    string
     ↪  `protobuf:"bytes,1,opt,name=moderator_address,json=moderatorAddress,proto3"
     ↪  json:"moderator_address,omitempty"`
    NewModeratorAddress string `protobuf:"bytes,2,opt,name=new_moderator_address,js⌋
     ↪  on=newModeratorAddress,proto3" json:"new_moderator_address,omitempty"`
}
```

On the server-side, the message handlers are also wired and expect to be called:

```
// x/oracle/keeper/msg_server.go
func (k Keeper) SubmitOracleData(c context.Context, msg *types.MsgSubmitOracleData)
↪  (*types.MsgSubmitOracleDataResponse, error) {...}
// ..snip
func (k Keeper) UpdateModeratorAddress(c context.Context, msg
↪  *types.MsgUpdateModeratorAddress) (*types.MsgUpdateModeratorAddressResponse,
↪  error) {...}
// ..snip
```

Without registration:

1. Amino JSON fails: `unmarshal error: unknown concrete type name`.
2. Protobuf path lookup fails: `can't find type for URL /guru.oracle.MsgSubmitOracleData`.

Therefore any signed transaction containing these messages is rejected before it reaches the module's message server.

Impact

REST, or transaction clients can't use these messages, so transactions that deliver live data to the oracle cannot be processed, completely breaking external data feeds, also administrative rotation of the moderator address cannot be executed through normal governance or CLI flows.

Recommendation

Add both messages to Amino and interface registries:

```
// x/oracle/types/codec.go
cdc.RegisterConcrete(&MsgSubmitOracleData{}, "oracle/SubmitOracleData", nil)
cdc.RegisterConcrete(&MsgUpdateModeratorAddress{}, "oracle/UpdateModeratorAddress",
 ↪  nil)

registry.RegisterImplementations((*sdk.Msg)(nil),
    &MsgSubmitOracleData{},
    &MsgUpdateModeratorAddress{},
)
```

Discussion

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/31

Bauchibred

Verified, everything looks good!

Issue M-1: Applying burn/base split before community-tax under-funds the community-pool [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/143

Summary

`AllocateTokens` subtracts the burn and base portions from the fee pot *first*, then applies `communityTax` to the shrunken remainder, this silently reduces the coins that reach the community pool compared with the canonical distribution formula.

Vulnerability Details

`keeper/AllocateTokens`

```
func (k Keeper) AllocateTokens(ctx context.Context, totalPreviousPower int64,
 ↪  bondedVotes []abci.VoteInfo) error {
// ..snip
// burn & base slices are removed
feesCollectedInt = feesCollectedInt.Sub(burnFee...).Sub(baseFee...)
feesCollected    := sdk.NewDecCoinsFromCoins(feesCollectedInt...)
// ..snip

// @audit community-tax applied after the removal
communityTax   := k.GetCommunityTax(ctx)          // e.g. 2 %
voteMultiplier := 1 - communityTax
feeMultiplier  := feesCollected * voteMultiplier
```

Since `feesCollectedInt` already excludes `burnFee` and `baseFee`, the tax base is smaller:

So community pool via current implementation = `(fees - burn - base) × communityTax`.

However community pool via canonical logic = `fees × communityTax`.

Impact

Loss of funds for the community pool, considering governance funds to accumulate slower than expected, altering economic assumptions.

Recommendation

Apply the community-tax before carving out burn and base portions, then the `feesCollected` post the application of the community tax should be where the burn/base ratios are applied against.

Issue M-2: Ignoring `UnsubscribeAll` error in subscriber leads to silent subscription leak and resource exhaustion [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/147

Summary

The oracle daemon's Subscriber calls `subsClient.UnsubscribeAll` when its context is cancelled, but discards the returned error.

If the WebSocket client is not running or the RPC layer is unhealthy/ doesnt have a ”unsubscribe_all” route defined, the unsubscribe fails silently and the daemon leaves dangling subscriptions on the node, gradually exhausting the CometBFT event system and leaking local resources.

Vulnerability Details

subscriber.go::subscribeToEvents:

```
func (s *Subscriber) subscribeToEvents(ctx context.Context, subsClient *http.HTTP)
↪   (<-chan coretypes.ResultEvent, <-chan coretypes.ResultEvent, <-chan
↪   coretypes.ResultEvent) {
    go func() {
        <-ctx.Done()
        subsClient.UnsubscribeAll(ctx, "")
        s.logger.Info("unsubscribed all")
    }()

```

Which calls the cometbft's `WSEvents.UnsubscribeAll`

```
func (w *WSEvents) UnsubscribeAll(ctx context.Context, _ string) error {
    if !w.IsRunning() {           // ← returns errNotRunning
        return errNotRunning
    }
    if err := w.ws.UnsubscribeAll(ctx); err != nil {
        return err                // ← network or server error
    }
    w.mtx.Lock()
    w.subscriptions = make(map[string]chan ctypes.ResultEvent)
    w.mtx.Unlock()
    return nil
}
```

But because the caller ignores the returned error, two failure modes go unnoticed:

1. The WebSocket client has already stopped (`IsRunning()==false`) → `errNotRunning`.
2. Network I/O or Tendermint node error while sending the unsubscribe request.

In both cases the goroutine believes it cleaned up and then `"unsubscribed all"` is logged while the subscription remains on the node and in the client's internal map.

Impact

Each restart of the subscriber leaks a new set of event subscriptions; after enough cycles the node could hit its per-peer subscription limit and starts rejecting new connections, breaking all downstream services.

Locally, `WSEvents.subscriptions` map retains channels that are never closed, holding memory and goroutines which also make operators to receive misleading `"unsubscribed all"` logs, masking the real cause of resource exhaustion and complicating incident response.

Recommendation

Handle and log the error explicitly; use a non-cancelled context so the request can complete even after `ctx.Done()`.

Discussion

PamLa-gurufin

Thanks, I fixed the initial issue and the additional issue.
https://github.com/gurufinglobal/guru/pull/16

Bauchibred

Fixes look good!

Issue M-3: Unmarshal errors remain unhandled possibly causing invalid endpoints to be registered [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/153

Description

`Marshal()` is called in many instances, in two instances however, the return value is not handled:

`UpdateOracleRequestDoc`:

```
// Marshal the endpoints to a JSON string
endpointsJson, _ := json.Marshal(doc.RequestDoc.Endpoints)
```

`RegisterOracleRequestDoc`:

```
endpointsJson, _ := json.Marshal(oracleRequestDoc.Endpoints)
```

Properly marshalling the endpoints is important as they serve as the source of data fetching.

As previously mentioned, `Marshal()` error value is unhandled and this is problematic as it can return an error:

```
func Marshal(v any) ([]byte, error) {
    e := newEncodeState()
    defer encodeStatePool.Put(e)

    err := e.marshal(v, encOpts{escapeHTML: true})
    if err != nil {
        return nil, err
    }
    buf := append([]byte(nil), e.Bytes()...)

    return buf, nil
}
```

Note that this does not require malicious intent. An endpoint could simply be invalid or failed to marshal. As of now it silently fails and uses the invalid endpoint. When it is then later unmarshalled it won't work as it has not been properly marshalled in the first place.

Recommendation

We would recommend checking for the returned `err` value. Just like it is done in other places of the codebase, such as:

```go
value, err := json.Marshal(log)
if err != nil {
    return nil, errorsmod.Wrap(err, "failed to encode log")
```

Issue M-4: Test moderator address is hardcoded into default genesis state [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/155

Summary

DefaultGenesisState embeds a fixed Bech32 string as the initial ModeratorAddress.

Vulnerability Details

types::DefaultGenesisState()

```
return &GenesisState{
    Params:             DefaultParams(),
    OracleRequestDocCount: 0,
    OracleRequestDocs:      []OracleRequestDoc{},
    ModeratorAddress:       "guru10jmp6sgh4cc6zt3e8gw05wavvejgr5pwggsdaj", // test
     ↪  address
}
```

Impact

A wrong address would be passed in as the moderator, blocking all operations to be made by the moderator in the module due to the `!=moderatorAddress` check.

Recommendation

Return an empty moderator field and force explicit configuration via a validation step:

```
if gs.ModeratorAddress == "" {
    return fmt.Errorf("moderator address must be set in genesis")
}
```

Discussion

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/23

Bauchibred

Acknowledged, we now force an explicit configuration of the moderator address.

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/43

We would kindly request that you review the final commit (https://github.com/gurufinglobal/guru/pull/43/commits/1e742b638c8fbef46378ceac741a663500873c1a) of the PR.

lpetroulakis

Reviewed - fixes look good.

Issue M-5: `OracleRequestDoc` is never validated in the update handler [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/158

Summary

OracleRequestDoc's validate() exists in order to perform validation on `OracleRequestDoc`, `MsgUpdateOracleRequestDoc` however skips calling OracleRequestDoc.Validate(), cause the code is commented out, this then causes the keeper to merge un-checked fields into the stored document and writes it back without any final validation.

Vulnerability Details

First see MsgUpdateOracleRequestDoc.ValidateBasic

```
// ValidateBasic implements the sdk.Msg interface
func (msg MsgUpdateOracleRequestDoc) ValidateBasic() error {
    if _, err := sdk.AccAddressFromBech32(msg.ModeratorAddress); err != nil {
        return errorsmod.Wrapf(errortypes.ErrInvalidAddress, "invalid from
         ↪  address(Moderator) (%s)", err)
    }
    // if err := msg.RequestDoc.Validate(); err != nil {
    //  return errorsmod.Wrap(errortypes.ErrInvalidRequest, err.Error())
    // }
    return nil
}
```

Above, we notice that the call to OracleRequestDoc.Validate() is commented out, which is unlike the implementation when we are registering the oracle request doc:

https://github.com/sherlock-audit/2025-09-gurufin-chain/blob/d63b56ad665dc01eeb38a7c5cf17b42219ca8cc1/guru-v2/x/oracle/types/msgs.go#L45-L54

```
// ValidateBasic implements the sdk.Msg interface
func (msg MsgRegisterOracleRequestDoc) ValidateBasic() error {
    if _, err := sdk.AccAddressFromBech32(msg.ModeratorAddress); err != nil {
        return errorsmod.Wrapf(errortypes.ErrInvalidAddress, "invalid from
         ↪  address(Moderator) (%s)", err)
    }
    if err := msg.RequestDoc.Validate(); err != nil {
        return errorsmod.Wrap(errortypes.ErrInvalidRequest, err.Error())
    }
    return nil
}
```

The implementation in MsgUpdateOracleRequestDoc.ValidateBasic then causes the keeper to merge un-checked fields into the stored document bypassing all the validation checks.

Impact

All validation checks present in OracleRequestDoc.Validate() and not in the process of the keeper updating the oracle docs are bypassed by the current implementation which can lead to:

- Quorum being higher than `len(AccountList)` → oracle submissions can never reach quorum, halting data updates.
- Empty or nil `Endpoints` → worker pool panics on index access, daemon stops processing the request.
- Malformed Bech32 addresses → authorization checks fail, but only after resources are spent fetching and submitting data.

> Overall, the oracle module can be placed in a self-inflicted DoS state by a single malicious or accidental update tx.

Recommendation

Re-enable full message validation when updating the oracle docs:

```
// ValidateBasic implements the sdk.Msg interface
func (msg MsgUpdateOracleRequestDoc) ValidateBasic() error {
    if _, err := sdk.AccAddressFromBech32(msg.ModeratorAddress); err != nil {
        return errorsmod.Wrapf(errortypes.ErrInvalidAddress, "invalid from
         ↪  address(Moderator) (%s)", err)
    }
-   // if err := msg.RequestDoc.Validate(); err != nil {
-   //  return errorsmod.Wrap(errortypes.ErrInvalidRequest, err.Error())
-   // }
+    if err := msg.RequestDoc.Validate(); err != nil {
+       return errorsmod.Wrap(errortypes.ErrInvalidRequest, err.Error())
+    }
    return nil
}
```

Discussion

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/25

Bauchibred

Fixes look good!

Issue M-6: Amino name mismatch in `MsgUpdateParams` blocks governance txs in legacy sign-mode [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/159

Summary

The x/feemarket codec registers `"os/feemarket/MsgUpdateParams"` while the proto file declares `"cosmos/evm/x/feemarket/MsgUpdateParams"`. So any transaction signed with `SIGN_MODE_LEGACY_AMINO_JSON` fails to decode on-chain.

Vulnerability Details

x/feemarket/types/codec.go#L22-L25

```go
const (
    // Amino names
    updateParamsName = "os/feemarket/MsgUpdateParams"
)

// ..snip

// RegisterLegacyAminoCodec required for EIP-712
func RegisterLegacyAminoCodec(cdc *codec.LegacyAmino) {
    cdc.RegisterConcrete(&MsgUpdateParams{}, updateParamsName, nil)
}
```

As seen we register the amino name as `os/feemarket/MsgUpdateParams` however the proto file declares a different amino name `cosmos/evm/x/feemarket/MsgUpdateParams`.

feemarket/v1/tx.proto

```
option (amino.name) = "cosmos/evm/x/feemarket/MsgUpdateParams";
```

Amino JSON clients use the proto tag when encoding.

Now since Gurufin still advertises SIGN_MODE_LEGACY_AMINO_JSON (see cosmos-sdk/client/flags/flags.go), any client is free to use that mode, so the mismatch becomes a hard failure path the first time the message reaches the network.

Assume wallet CLI signs governance proposal with `--sign-mode amino-json`, node receives tx, Amino `UnmarshalJSON` looks up the type string in its registry.

Lookup fails → `ErrUnknownRequest`, tx rejected.

Impact

Legacy Amino users cannot submit `MsgUpdateParams`, causing friction for governance parameter changes.

Recommendation

Align the strings, alternatively advise operators to prefer protobuf sign-mode; Amino remains supported for backward compatibility.

Discussion

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/27

Bauchibred

Fix verified.

Issue M-7: `fetchRawData` accepts unbounded response read and full-body error relay allowing DOS [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/161

Summary

The oracle daemon's fetchRawData downloads external JSON using `io.ReadAll` without imposing a size limit, and when the HTTP status is non-retryable it embeds the entire body in the returned error string:

```
body, _ := io.ReadAll(res.Body)          // ← allocates full body
...
return nil, fmt.Errorf("HTTP %d: %s",    // ← copies body again
                       res.StatusCode, string(body))
```

Vulnerability Details

As hinted under summary, here is a fuller implementation of client.go::fetchRawData()

```

func (hc *httpClient) fetchRawData(url string) ([]byte, error) {
    maxAttempts := max(1, config.RetryMaxAttempts())
    for attempt := 0; attempt < maxAttempts; attempt++ {
        if 0 < attempt {
            retryDelay := time.Duration(1<<(attempt-1)) * time.Second
            time.Sleep(max(retryDelay, config.RetryMaxDelaySec()))
        }

        req, err := http.NewRequest(http.MethodGet, url, nil)
        if err != nil {
            return nil, err
        }

        req.Header.Set("User-Agent", "Guru-V2-Oracle/1.0")
        req.Header.Set("Accept", "application/json")

        res, err := hc.client.Do(req)
        if err != nil {
            continue
        }

        body, err := io.ReadAll(res.Body)//@audit  no size cap
        res.Body.Close()
        if err != nil {
            return nil, err
        }

        switch {
        case res.StatusCode == http.StatusOK:
            return body, nil

        case 500 <= res.StatusCode:
            continue

        case res.StatusCode == http.StatusRequestTimeout ||
            res.StatusCode == http.StatusTooManyRequests ||
            res.StatusCode == http.StatusConflict:
            continue

        default:  //@audit any 4xx, 501 …
            return nil, fmt.Errorf("HTTP %d: %s", res.StatusCode, string(body))
        }
    }

    return nil, fmt.Errorf("failed to fetch raw data after %d attempts",
    ↪   maxAttempts)
}
```

So an endpoint that returns a huge or binary payload can therefore:

1. Force the daemon to allocate arbitrary amounts of RAM, and
2. Flood logs / telemetry pipelines with megabytes of error text.

Neither condition is bounded by configuration; a malicious or simply mis-configured endpoint can exhaust memory, disk, or log quotas.

Impact

Memory exhaustion / DOS, considering several concurrent jobs with large bodies can push the node past container limits.

Recommendation

Impose a hard limit on the body size and truncate on error.

Discussion

PamLa-gurufin

Here is the PR! https://github.com/gurufinglobal/guru/pull/35

Bauchibred

Fixes look good, the bug window has been closed.

Issue M-8: Quadratic median sort is done in Begin-Block risking a chain-wide DoS [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/162

Summary

The oracle's calculateMedian uses a classic O(n^2) bubble-sort on every aggregation risking a chain wide DOS when the data set to bubble sort is large.

Vulnerability Details

calculateMedian uses a classic O(n^2) bubble-sort on every aggregation:

```
for i := 0; i < len(values)-1; i++ {
    for j := i + 1; j < len(values); j++ {
        if values[i].Cmp(values[j]) > 0 {
            values[i], values[j] = values[j], values[i]
        }
    }
}
```

Now, ProcessOracleDataSetAggregation is invoked in BeginBlock when the oracle is enabled for each validator. With a large `AccountList` the sort in `calculateMedian()` performs ~$n^2/2$ big-number comparisons per block having a very high block-processing budget and stalling the chain.

A somewhat extreme example would be n = 1 000 submissions for a very active oracle price point, this makes us have □ 500 000 `big.Float.Cmp` calls × every validator × every block the oracle is enabled, saturating the CPU and leading to liveness issues.

Impact

Liveness issues considering this is directly called in the Begin blocker.

Recommendation

Replace the classic bubble sort with adifferent algorithm or implement a defensive limit on the number of submissions, then enforce `len(AccountList)` and `doc.Quorum` □ safe threshold (e.g. □ 100) during `doc.Validate()`.

Discussion

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/28

Bauchibred

Fix verified, everything looks good!

Issue M-9: Oracle module parameters are hardcoded post genesis [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/163

Summary

The oracle module defines parameters (`submit_window`, `min_submit_per_window` `slash_fraction_downtime`) together with a `Params.Validate()` function, but exposes no governance proposal, message, or keeper method to modify them after genesis.

Vulnerability Details

First note params.go#L17-L35

```
}

// Validate performs basic validation on oracle parameters
func (p Params) Validate() error {
    if p.SubmitWindow == 0 {
        return fmt.Errorf("submit window cannot be zero")
    }

    if p.MinSubmitPerWindow.IsNegative() {
        return fmt.Errorf("min submit per window cannot be negative")
    }

    if p.SlashFractionDowntime.IsNegative() {
        return fmt.Errorf("slash fraction downtime cannot be negative")
    }

    return nil
}
```

Note that the only call-sites are inside genesis processing:

`module::AppModule.ValidateGenesis()` → `GenesisState.Validate()` → `Params.Validate()`

So after InitGenesis stores the params via k.SetParams, there is no functionality like a `MsgUpdateParams`.

Impact

Parameter immutability prevents the network from tightening or relaxing submission windows in response to usage or liveness issues, adjusting `min_submit_per_window` and slashing fractions to align incentives over time.

This rigidity can lead to stalled oracle feeds, considering some feeds could be more active than others and would need to be updated more often than others.

Recommendation

Introduce an authority-controlled `MsgUpdateParams`.

Discussion

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/29

Bauchibred

Fixes look good since we can now update the oracle params, but the current CLI is stale as it does not allow us to update the new `MaxAccountlistSize` param.

Eddy-gurufin

I have updated it. Please take another look!

Bauchibred

> I have updated it. Please take another look!

Fix verified!

Issue M-10: Missing raw_data numeric validation can break oracle feed [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/165

Summary

During validation `msg.DataSet.RawData` is only checked for non-emptiness. If the string is not a valid decimal, `big.Float.SetString` fails silently and the value remains zero. Down-stream aggregation (calculateMin, calculateAverage, calculateMedian, calculateMax) therefore treats the submission as 0, skewing oracle prices.

Vulnerability Details

First see MsgSubmitOracleData.ValidateBasic

```
// ValidateBasic implements the sdk.Msg interface
func (msg MsgSubmitOracleData) ValidateBasic() error {
    if _, err := sdk.AccAddressFromBech32(msg.DataSet.Provider); err != nil {
        return errorsmod.Wrapf(errortypes.ErrInvalidAddress, "invalid provider
         ↪  address (%s)", err)
    }
    if msg.DataSet.RequestId == 0 {
        return errorsmod.Wrap(errortypes.ErrInvalidRequest, "request ID cannot be
         ↪  empty")
    }
    if msg.DataSet.RawData == "" {//@audit only cehcked that it is non-empty
        return errorsmod.Wrap(errortypes.ErrInvalidRequest, "raw data cannot be
         ↪  empty")
    }
    if msg.DataSet.Signature == "" {
        return errorsmod.Wrap(errortypes.ErrInvalidRequest, "signature cannot be
         ↪  empty")
    }
    return nil
}
```

It being non-empty however does not validate that the string is a valid decimal.

Using calculateAverage as an example

```
func (k Keeper) calculateAverage(submitDatas []*types.SubmitDataSet) (string,
 ↪  error) {
    if len(submitDatas) == 0 {
        return "", fmt.Errorf("no data to average")
    }
    sum := new(big.Float)
    for _, data := range submitDatas {
|>      value := new(big.Float)
|>      value.SetString(data.RawData)
|?      sum.Add(sum, value)
    }

    avg := new(big.Float).Quo(sum, new(big.Float).SetInt64(int64(len(submitDatas))))
    return avg.Text('f', -1), nil
}
```

While looping, we can see that the value defaults to `0` and then we query `SetString` to set the value, `SetString` however could return false if the string is not a valid decimal:

```
// SetString sets z to the value of s and returns z and a boolean indicating
// success. s must be a floating-point number of the same format as accepted
// by [Float.Parse], with base argument 0. The entire string (not just a prefix)
 ↪  must
// be valid for success. If the operation failed, the value of z is undefined
// but the returned value is nil.
func (z *Float) SetString(s string) (*Float, bool) {
    if f, _, err := z.Parse(s, 0); err == nil {
        return f, true
    }
    return nil, false
}
```

So since the boolean result of `SetString` is ignored, an invalid string such as `"abc"` or `"10, 5"` leaves `value` at 0 and then zero gets silently added via `sum.Add(sum, value)`.

Impact

Malformed data passes current validation yet is interpreted as zero on-chain. This can heavily skew aggregated prices, assume we have only three submissions having one defaulting to 0 drops the computed average by ~ 33 %, if the prices are close to each other, e.g:

- Price point A = 1000
- Price point B = 1001
- Price point C defaults to 0

The average would be (1000 + 1001 + 0) / 3 = `667` heavily deflating the price.

Asides the above, having one of the datapoint default to `0` also distorts median or min, triggering incorrect actions on all Guru's logic that relies on accurate price feeds.

Recommendation

Add numeric parsing in ValidateBasic, we can do this by checking that `setString()` does not return false:

```
// ValidateBasic implements the sdk.Msg interface
func (msg MsgSubmitOracleData) ValidateBasic() error {
    if _, err := sdk.AccAddressFromBech32(msg.DataSet.Provider); err != nil {
        return errorsmod.Wrapf(errortypes.ErrInvalidAddress, "invalid provider
         ↪  address (%s)", err)
    }
    if msg.DataSet.RequestId == 0 {
        return errorsmod.Wrap(errortypes.ErrInvalidRequest, "request ID cannot be
         ↪  empty")
    }
    if msg.DataSet.RawData == "" {
        return errorsmod.Wrap(errortypes.ErrInvalidRequest, "raw data cannot be
         ↪  empty")
    }
+   if _, ok := new(big.Float).SetString(msg.DataSet.RawData); !ok {
+       return errorsmod.Wrapf(errortypes.ErrInvalidRequest,
+           "raw data must be a valid decimal number: %q", msg.DataSet.RawData)
+   }
    if msg.DataSet.Signature == "" {
        return errorsmod.Wrap(errortypes.ErrInvalidRequest, "signature cannot be
         ↪  empty")
    }
    return nil
}
```

Discussion

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/30

Bauchibred

Fix verified, we now check the bool value returned from `setString` not to be false.

Issue M-11: Job execution prematurely increments the nonce causing on-chain rejections of valid data sets [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/167

Summary

worker::executeJob increments and stores the job's nonce before it knows whether fetching, parsing and extracting external data will succeed. If that external work fails, the in-memory store now holds a nonce `n` that was never submitted on-chain (chain still expects `n`).

Now when the daemon later reschedules the same request, it adds `+1`, creating nonce `n + 1` and finally submits that, yet the on-chain check when submitting data requires `n`, so the transaction is rejected.
If the daemon never re-schedules, the feed stalls permanently at `n - 1`.

Vulnerability Details

pool.go::executeJob is used to schedule a single job from `processcomplete()` or `processrequestdoc()`.

```
func (wp *WorkerPool) executeJob(ctx context.Context, job *types.OracleJob) {
    task := job

    wp.workerFunc(func() error {
        if 0 < task.Nonce && 0 < task.Delay {
            select {
            case <-time.After(task.Delay):
            case <-ctx.Done():
                return nil
            }
        }

        reqID := strconv.FormatUint(task.ID, 10)

        // @audit as seen, we have a premature increment & persistence
        if stored, ok := wp.jobStore.Get(reqID); ok {
        task.Nonce = stored.Nonce + 1   // n  (or n+1 on retries)
        } else {
        task.Nonce++                    // first run  n
        }
        wp.jobStore.Set(reqID, task)        // @audit we store nonce BEFORE doing
         ↪  work
        // @audit external IO that may fail
        rawData, err := wp.client.fetchRawData(task.URL)
        if err != nil {
            wp.logger.Error("failed to fetch raw data", "error", err)
            wp.resultCh <- nil
            return err
        }
        wp.logger.Debug("fetched raw data", "id", task.ID, "url", task.URL)

        jsonData, err := wp.client.parseRawData(rawData)//@audit parsing may also
        ↪  fail
        if err != nil {
            wp.logger.Error("failed to parse raw data", "error", err)
            return err
        }

        result, err := wp.client.extractDataByPath(jsonData, task.Path)//@audit
        ↪  extraction may also fail
        if err != nil {
            wp.logger.Error("failed to extract data by path", "error", err)
            return err
        }

        wp.resultCh <- &types.OracleJobResult{
            ID:    task.ID,
            Data:  result,
            Nonce: task.Nonce,
        }
        wp.logger.Debug("sent result to channel", "id", task.ID, "data", result)

        return nil
    })
}
```

Now in all cases, if failure occurs, the goroutine exits without any rollback of the nonce, now the next run picks the value for e.g in `processrequestdoc()`, it will increment the nonce to `n+1` and store it in the job store, but since the previous run failed, the on-chain nonce is still `n-1`, so the next run will be rejected cause of the on-chain validation in keeper.SubmitOracleData:

```
nonce := requestDoc.GetNonce()      // chain holds n-1
if msg.DataSet.Nonce != nonce+1 {   // requires n
    return error "nonce is not correct"
}
```

Impact

Every failure advances the local nonce, causing the next success attempt to be rejected; data point `n` is never recorded. If no re-emit occurs the worker stops scheduling; oracle feed halts at `n-1`. Resource waste – Network bandwidth and gas fees spent on doomed `n + k` retries.

Recommendation

Defer the `task.Nonce` increment and `jobStore.Set()` until all of fetch/parse/extract succeed and on error, leave the stored nonce unchanged so the next schedule re-attempts the same number.

Discussion

PamLa-gurufin

I'm always grateful. Here is the PR! https://github.com/gurufinglobal/guru/pull/38

Bauchibred

Fix verified.

Issue M-12: Disabled oracle requests cannot be re-enabled [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/169

Summary

updateOracleRequestDoc has a check that rejects any modification when the stored document's status is `REQUEST_STATUS_DISABLED`, later in code however we are meant to accept a status change. This means the check is intended to block any change to other structs of the OracleRequestDoc asides the status, but since the first check is blanket `if existingDoc.Status == types.RequestStatus_REQUEST_STATUS_DISABLED {return fmt.Errorf("cannot modify Request Doc with disabled status")}` it blocks the enable/disable lifecycle of the request doc itself.

Vulnerability Details

x/oracle/keeper/keeper.go

```
func (k Keeper) updateOracleRequestDoc(ctx sdk.Context, doc types.OracleRequestDoc)
 ↪  error {
    // Retrieve the existing oracle request document
    existingDoc, err := k.GetOracleRequestDoc(ctx, doc.RequestId)
    if err != nil {
        return err
    }

    //@audit here we get a blanket rejection
    if existingDoc.Status == types.RequestStatus_REQUEST_STATUS_DISABLED {
    return fmt.Errorf("cannot modify Request Doc with disabled status")
    }
    // Update the period if it is not empty
    if doc.Period != 0 {
        existingDoc.Period = doc.Period
    }
    // @audit here the code intends to set a new status (never reached)
    if doc.Status != types.RequestStatus_REQUEST_STATUS_UNSPECIFIED {
    existingDoc.Status = doc.Status
    }
    // ..snip
    }
```

Since branch 1 returns early, branch 2 is unreachable for a disabled request; even a message that only changes `Status` back to `ENABLED` fails.

Impact

Any oracle request once set to `DISABLED` is locked forever, so operators cannot reinstate feeds after maintenance or error, forcing them to create a new request ID and update all downstream consumers.

Recommendation

Allow status-only updates for disabled docs:

```
if existingDoc.Status == types.RequestStatus_REQUEST_STATUS_DISABLED &&
   doc.Status == types.RequestStatus_REQUEST_STATUS_UNSPECIFIED {
    return fmt.Errorf("cannot modify disabled Request Doc except status")
}
```

Discussion

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/33

Bauchibred

Fix verified.

Issue M-13: Fee-discount mis-ordering lets zero-fee accounts be prioritized over higher-paying users [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/170

Summary

gurud/ante/cosmos_fees.go::DeductFeeDecorator.AnteHandle computes a transaction's mem-pool priority before applying any discount fetched from `x/feepolicy`.

Designated accounts can receive a discount of up to **100 %** (fee = `0`). Because priority is locked in at the pre-discount amount, those accounts are queued ahead of ordinary users who actually pay the same or higher real fees.

Vulnerability Details

Original fee and priority calculations in the ante handler

```
func (dfd DeductFeeDecorator) AnteHandle(ctx sdk.Context, tx sdk.Tx, simulate bool,
↪   next sdk.AnteHandler) (sdk.Context, error) {
    feeTx, ok := tx.(sdk.FeeTx)
    if !ok {
        return ctx, errorsmod.Wrap(sdkerrors.ErrTxDecode, "Tx must be a FeeTx")
    }

    if !simulate && ctx.BlockHeight() > 0 && feeTx.GetGas() == 0 {
        return ctx, errorsmod.Wrap(sdkerrors.ErrInvalidGasLimit, "must provide
        ↪   positive gas")
    }

    var (
        priority int64
        err      error
    )

    fee := feeTx.GetFee()
    if !simulate {
|>      fee, priority, err = dfd.txFeeChecker(ctx, tx)//@audit priority is gotten
↪   here
        if err != nil {
            return ctx, err
        }
    }

    addrCodec := address.Bech32Codec{
        Bech32Prefix: sdk.GetConfig().GetBech32AccountAddrPrefix(),
    }
    feePayer, err := addrCodec.BytesToString(feeTx.FeePayer())
    if err != nil {
        return ctx, err
    }

    //@audit discount calculated and applied-but priority is not recomputed

    discount := dfd.feepolicyKeeper.GetDiscount(...)
    deductedFee = applyDiscount(fee, discount)       // can drop to 0

    // ..snip
    //@audit mempool stores the transaction:

    newCtx := ctx.WithPriority(priority)
}
```

Impact

Zero-fee or heavily-discounted accounts consistently jump ahead of higher-paying users, so paying a larger fee no longer guarantees faster inclusion breaking the priority tx ordering logic

Recommendation

Recalculate priority after the discount is applied.

Issue L-1: Discounts remain unchecked when calling RegisterDiscounts [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/140

Description

When calling `RegisterDiscounts` to add new discounts, there is no validation performed for the validity of the discount values.

Meaning that discounts can be set to:

- Negative values
- Values over 100%
- Invalid DiscountTypes.

You can also verify this by tweaking the existing `TestRegisterDiscounts` function

```
            name: "pass - valid msg",
            request: &types.MsgRegisterDiscounts{
                ModeratorAddress:
                 ↪  authtypes.NewModuleAddress(govtypes.ModuleName).String(),
                Discounts: []types.AccountDiscount{
                    {
                        Address: "guru1gzsvk8rruqn2sx64acfsskrwy8hvrmaf6dvhj3",
                        Modules: []types.ModuleDiscount{
                            {
                                Module: "bank",
                                Discounts: []types.Discount{
                                    {
                                        DiscountType: "percent",
                                        MsgType: "/cosmos.bank.v1beta1.MsgSend",
-                                       Amount:       math.LegacyNewDec(100),
+                                       Amount:       math.LegacyNewDec(-20),
                                    },
```

This will pass even though the `Amount` is negative.

You can also change the `DiscountType` to something other than ”percent”, such as ”invalid” and this will still pass.

Impact

It seems that only trusted parties are able to register discounts, if they are expected not to act maliciously the impact of this finding would be less severe. Since I am unaware of this information I will submit it as a `Medium`.

Fix

I have noticed that there is a verification function for the RegisterDiscount messages, however this is not added yet to the flow of `RegisterDiscounts`. I would advise to add this to the flow path.

Issue L-2: Totalburned can be set to invalid values [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/141

Hi friends,

The `ResetTotalBurned` function allows the moderator to reset the total burned amount for a given denom.

The issue is that the newly set `msg.Amount` is not validated. This means a moderator can arbitrarily set the value to anything.

For existing coins:

```
totalBurned[i] = sdk.NewCoin(msg.Denom, msg.Amount)
```

For new coins:

```
totalBurned = totalBurned.Add(sdk.NewCoin(msg.Denom, msg.Amount))
```

By doing this there will be an incorrect reflection of the actual burned amount of the particular token.

I'm not entirely sure what this function is intended to be used for, which makes it difficult to fully gauge the impact. For now, I've classified this as `medium`, but if the function has no real use-case feel free to downgrade the severity.

Recommendation

The appropriate fix for `msg.Amount` depends on the intended purpose of the `ResetTotalBurned` function. For example:

- If the reset is only meant to set the amount back to `0`, enforce a validation that requires `msg.Amount == 0`.

Issue L-3: SubmitOracleData can be used to submit stale requests [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/142

Description

Inside `SubmitOracleData` the following check is performed:

```
if msg.DataSet.Nonce != nonce+1 {
    return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, "nonce is not correct")
}
```

This check is intended to ensure nonces are unique. However, because `msg.DataSet.Nonce` can be set arbitrarily by the caller, the check alone is insufficient.

Compounding the problem, `requestId` can also be set by the caller since it is derived from `msg.DataSet.RequestId`:

```
requestId := msg.DataSet.RequestId
```

Because both `RequestId` and `Nonce` are controlled by the sender, invalid/stale data can be submitted.

Example:

- Attacker Bob chooses an old `RequestId`, for example `msg.DataSet.RequestId = 20`.
- Bob sets `msg.DataSet.Nonce` to match the expected nonce which is checked here:

```
if msg.DataSet.Nonce != nonce+1 {
    return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, "nonce is not correct")
}
```

- This allows Bob to submit data for `RequestId(20)` even though the data is invalid and stale.

Recommendation

Ensure that both `nonce` and `requestId` are validated strictly to prevent reuse of old or invalid requests.

Discussion

iammxuse

Hi friends,

with regards to this message on discord:

> If you have any suggestions for preventing the reuse of outdated or incorrect requests, we would greatly appreciate your input.

You could for example keep track of the most recent requests and ensure that whenever `SubmitOracleData` is called it does not use any of these already existing requests.

Obviously you guys have more context than I do so I will leave the exact fix up to you.

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/36

Issue L-4: Oracle jobs periods are stored in mismatched units [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/149

Summary

ProcessRequestDoc stores `requestDoc.Period` directly in `OracleJob.Period` by casting it to `time.Duration` without multiplying by `time.Second` unlike `delay`.

```
Period: time.Duration(requestDoc.Period), // ← units mismatch
```

Because the protobuf field represents **seconds** while `time.Duration` defaults to **nanoseconds**, all later calculations that treat `job.Period` as seconds operate on values way way smaller, this then skews the delay logic and make oracle jobs fire in wrong time.

Vulnerability Details

First see ProcessRequestDoc()

```
func (wp *WorkerPool) ProcessRequestDoc(ctx context.Context, requestDoc
 ↪  oracletypes.OracleRequestDoc, timestamp uint64) {
    if requestDoc.Status != oracletypes.RequestStatus_REQUEST_STATUS_ENABLED {
        wp.logger.Info("request document is not enabled", "request_id",
         ↪  requestDoc.RequestId)
        return
    }

// ..snip
periodSec := uint64(requestDoc.Period)   // correct: seconds
// ...snip
job := &types.OracleJob{
    // ...snip
    Delay:  time.Duration(max(int64(0), dsec)) * time.Second,//@audit delay is
     ↪  correctly computed in seconds
    Period: time.Duration(requestDoc.Period), // nanoseconds!
}

    wp.executeJob(ctx, job)
}
```

Going to the docs of common durations we can see:

```
const (
    Nanosecond  Duration = 1
    Microsecond          = 1000 * Nanosecond
    Millisecond             = 1000 * Microsecond
    Second                  = 1000 * Millisecond
    Minute                  = 60 * Second
    Hour                    = 60 * Minute
)
```

This would then have our nanosecond/second mismatch break short periods

Impact

Low, considering this value "Period" is noit used when executing the job and is only used in `processComplete` where the integer value is taken back, so currently we would only have issues with sub seconds periods which is not intended according to the protobuf.

Recommendation

Store the period consistently in seconds:

```
Period: time.Duration(requestDoc.Period) * time.Second,
```

and when converting back in `processComplete`:

```
periodSec := uint64(job.Period / time.Second)
```

Issue L-5: `SubmitOracleData` has a redundant nil-check post `GetOracleRequestDoc` query [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/150

Summary

SubmitOracleData verifies that the target request document exists:

```
// msg_server.go
requestDoc, err := k.GetOracleRequestDoc(ctx, requestId)
if err != nil {
    return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, "request document not
    ↪  found")
}

if requestDoc == nil {                              //@audit this is unreachable
    return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, "request document not
    ↪  found")
}
```

GetOracleRequestDoc however already guarantees that a successful call returns a non-nil pointer:

```
func (k Keeper) GetOracleRequestDoc(ctx sdk.Context, id uint64)
↪  (*types.OracleRequestDoc, error) {
    store := ctx.KVStore(k.storeKey)
    bz := store.Get(types.GetOracleRequestDocKey(id))
    if len(bz) == 0 {
        return nil, fmt.Errorf("not exist RequestDoc(req_id: %d)", id)
    }

    var doc types.OracleRequestDoc
    k.cdc.MustUnmarshal(bz, &doc)
    return &doc, nil  //@audit never nil when err == nil
}
```

Impact

Since any “not found” case sets `err != nil`, the subsequent `requestDoc == nil` check in SubmitOracleData can never evaluate to true. Making the second branch unreachable.

Recommendation

Delete the redundant nil-check and simplify control flow.

Issue L-6: Disabled or paused oracle requests remain in memory and keep executing [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/151

Summary

WorkerPool adds every active OracleJob to `jobStore`, but never removes a job when the on-chain request is at `PAUSED` or `DISABLED`. The entry stays in the concurrent map, causing unnecessary load and unbounded memory growth.

Vulnerability Details

```
func (wp *WorkerPool) ProcessRequestDoc(ctx context.Context, requestDoc
 ↪  oracletypes.OracleRequestDoc, timestamp uint64) {
    if requestDoc.Status != oracletypes.RequestStatus_REQUEST_STATUS_ENABLED {
        wp.logger.Info("request document is not enabled", "request_id",
         ↪  requestDoc.RequestId) //@audit log and return - no deletion
        return
    }
    wp.jobStore.Set(reqID, job)         // insert or overwrite
}
```

As seen, update / Register events with non-enabled status exit early and existing map entries are left untouched, ProcessComplete later reads the job and reschedules it without re-checking status, so disabled requests still produce traffic and we have no periodic sweep or `Pop` on context shutdown.

Impact

Memory leak over long-running nodes as thousands of obsolete jobs accumulate.

Recommendation

Remove jobs on status downgrade.

Issue L-7: Comments wrongly reference a `cex` module in the oracle module [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/154

Summary

Several comment strings in x/oracle/module.go still reference the **cex** module even though the file is of the oracle module. These include the headers for GetTxCmd, GetQueryCmd, and the AppModuleBasic struct description.

Vulnerability Details

module.go::AppModuleBasic

```go
// Name returns the cex module's name.
func (AppModuleBasic) Name() string {
    return types.ModuleName
}

// GetTxCmd returns no root tx command for the cex module
func (AppModuleBasic) GetTxCmd() *cobra.Command { ... }

// GetQueryCmd returns the root query command for the cex module
func (AppModuleBasic) GetQueryCmd() *cobra.Command { ... }

// AppModuleBasic defines the basic application module used by the cex module
type AppModuleBasic struct{}
```

Impact

low

Recommendation

Replace every occurrence of “cex” with “oracle” .

Discussion

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/22

Bauchibred

Fixes look good!

Issue L-8: Attached signature in oracle data submission is never verified [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/156

Summary

keeper::validateSubmitData and the surrounding SubmitOracleData flow never verify that the `Signature` field in `MsgSubmitOracleData` was produced by the `Provider`. Any whitelisted account can submit arbitrary oracle values since the signature to be passed in the data set can be set for all, and the transaction will pass both ValidateBasic and keeper validation.

Vulnerability Details

First see types::MsgSubmitOracleData.ValidateBasic()

```
if msg.DataSet.Signature == "" {
    return errorsmod.Wrap(errortypes.ErrInvalidRequest, "signature cannot be empty")
}
```

Only checks that the field is non-empty.

keeper::validateSubmitData

```
func (k Keeper) validateSubmitData(data types.SubmitDataSet) error {
    if data.RequestId == 0 { return … }
    if data.Nonce == 0     { return … }
    if data.Provider == "" { return … }
    if data.RawData == ""  { return … }
    return nil             // Signature never examined
}
```

Since no cryptographic verification is performed, the keeper later stores the dataset unchanged:

keeper::SetSubmitData

```
k.SetSubmitData(ctx, *msg.DataSet)   // Signature accepted as-is
```

Therefore the entire oracle pipeline trusts unauthenticated input.

Impact

Any address in account_list can submit arbitrary oracle values without possessing the separate oracle-signing key that the `DataSet.Signature` field was intended to prove.

> Considering the current trust model around the account lists and oracle providers, this can only be classified as `low/info` as we have very rare likelihood of malicious intent.

Recommendation

During SubmitOracleData, recover the signer from `DataSet.Signature` and `RawData` and check that it matches `Provider`; reject otherwise. Perform this verification inside validateSubmitData.

Discussion

PamLa-gurufin

Here is the PR! https://github.com/gurufinglobal/guru/pull/40

Bauchibred

Fix verified.

Issue L-9: The oracleRequestDoc count check is not fully implemented [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/157

Summary

oracle::InitGenesis only panics when `oracleRequestDocCount` is less than the actual number of request documents, but allows it to be `greater`, creating an inconsistent counter in state.

Vulnerability Details

oracle::InitGenesis

https://github.com/sherlock-audit/2025-09-gurufin-chain/blob/d63b56ad665dc01eeb38a7c5cf17b42219ca8cc1/guru-v2/x/oracle/genesis.go#L11-L48

```go
// InitGenesis new oracle genesis
func InitGenesis(ctx sdk.Context, k keeper.Keeper, data types.GenesisState) {
// ..snip

    if uint64(len(data.OracleRequestDocs)) > 0 && data.OracleRequestDocCount <
     ↪  uint64(len(data.OracleRequestDocs)) {
        panic(errorsmod.Wrapf(errortypes.ErrInvalidRequest, "%s: oracle request doc
         ↪  count is less than the number of oracle request docs",
         ↪  types.ModuleName))
    }//@audit

    // Set oracle request doc count
    k.SetOracleRequestDocCount(ctx, data.OracleRequestDocCount)
}
```

As seen, comparison misses the case where `oracleRequestDocCount` is larger than `len(docs)`, the function then writes the inflated value with `k.SetOracleRequestDocCount(ctx, oracleRequestDocCount)` leaving the keeper's stored counter out of sync with the real number of documents.

Impact

The safety check of matching the docs count with the exact requested docs is not fully implemented, allowing the counter to be inflated, leading to iterators and queries that rely on the counter may traverse non-existent keys, returning empty results.

Recommendation

Consider replacing the conditional with an equality check:

```
if uint64(len(docs)) != oracleRequestDocCount {
    panic("oracle request doc count must match actual documents")
}
```

Discussion

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/24

Bauchibred

Verified, we now have a `!=` check.

Issue L-10: Lack of error logging in retry loop hides HTTP failure [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/160

Summary

fetchRawData retries silently when `hc.client.Do(req)` returns an error. Because the error is ignored and not logged, operators have no visibility into the network/server problems that prevent data retrieval.

Vulnerability Details

client.go::fetchRawData():

```
func (hc *httpClient) fetchRawData(url string) ([]byte, error) {
// ..snip
res, err := hc.client.Do(req)
if err != nil {
    continue            // ← error discarded
}
// ..snip
}
```

Impact

Low, persistent DNS or TLS errors would then look like “oracle stuck” with no clue why.

Recommendation

Log the error.

Discussion

PamLa-gurufin

Here is the PR! https://github.com/gurufinglobal/guru/pull/34

Bauchibred

Everything looks good!

Issue L-11: Missing log for shutdown-time worker errors reduces observability [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/164

Summary

When the oracle worker pool shuts down it waits for all running tasks, but discards the error returned by `taskgroup.Wait()`.

Vulnerability Details

See worker/pool.go#new()

```go
func New(ctx context.Context, logger log.Logger) *WorkerPool {
    wp := new(WorkerPool)
    wp.logger = logger

    wp.jobStore = cmap.New[*types.OracleJob]()
    wp.resultCh = make(chan *types.OracleJobResult, config.ChannelSize())

    wp.workerGroup, wp.workerFunc = taskgroup.New(nil).Limit(2 * runtime.NumCPU())
    go func() {
        <-ctx.Done()
        wp.workerGroup.Wait()//@audit error ignored
        close(wp.resultCh)
    }()

    wp.client = newHTTPClient(wp.logger)

    return wp
}
```

`taskgroup.Wait()` returns the first non-nil error from any task.
If the last batch of HTTP fetches fails during shutdown, that diagnostic is silently lost.

Impact

Low as this produces no functional failure, but operators lose visibility into errors that happen during context cancellation.

Recommendation

Capture and log the error:

```go
if err := wp.workerGroup.Wait(); err != nil {
    wp.logger.Error("worker group shutdown error", "error", err)
}
```

Discussion

PamLa-gurufin

Thanks for checking everything down to the smallest detail!

PR: https://github.com/gurufinglobal/guru/pull/26

Bauchibred

The pleasure is mine, fix verified, everything looks good!

Issue L-12: Potential nil-dereference in `SubmitOracleData` when DataSet is absent [RESOLVED]

Source: https://github.com/sherlock-audit/2025-09-gurufin-chain/issues/168

Summary

SubmitOracleData always assumes `msg.DataSet` is non-nil and dereferences it immediately. Because `MsgSubmitOracleData` has `DataSet` as an optional pointer, a transaction that omits the field triggers a runtime panic.

Vulnerability Details

See x/oracle/keeper/msg_server.go

```
func (k Keeper) SubmitOracleData(c context.Context, msg *types.MsgSubmitOracleData)
 ↪  (*types.MsgSubmitOracleDataResponse, error) {
    ctx := sdk.UnwrapSDKContext(c)

    err := k.validateSubmitData(*msg.DataSet)
    if err != nil {
        return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, err.Error())
    }

// ..snip
    k.SetSubmitData(ctx, *msg.DataSet)


}
```

`MsgSubmitOracleData` definition:

```
type MsgSubmitOracleData struct {
    // The oracle data set to be submitted, containing the raw data and metadata
    DataSet *SubmitDataSet
     ↪  `protobuf:"bytes,2,opt,name=data_set,json=dataSet,proto3"
     ↪  json:"data_set,omitempty"`
}
```

When `DataSet == nil` both dereferences raise a panic aborting the handler.

Impact

A malformed transaction causes a crash.

Likelihood is very low as honest providers should always supply `DataSet`.

Recommendation

Add a simple validation that rejects empty payloads:

```
if msg.DataSet == nil {
    return nil, errorsmod.Wrap(errortypes.ErrInvalidRequest, "data_set must be
     ↪  provided")
}
```

Discussion

Eddy-gurufin

PR: https://github.com/gurufinglobal/guru/pull/32

lpetroulakis

fixed in the PR above

Disclaimers