ZetaChain - ZetaNode

Cosmos Security Audit

Prepared by: **Halborn**

Date of Engagement: **February 26th, 2023 - March 31st, 2023**

Visit: **Halborn.com**

DRAFT

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	02/26/2023	John Saigle
0.2	Document Updates	03/31/2023	John Saigle
0.3	Document Updates	04/04/2023	Gokberk Gulgun
0.4	Draft Review	04/04/2023	Gokberk Gulgun
0.5	Draft Review	04/04/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com
John Saigle	Halborn	John.Saigle@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

ZetaChain engaged Halborn to conduct a security audit on the ZetaChain Cosmos blockchain and related smart contracts beginning on February 26th, 2023 and ending on March 31st, 2023. The security assessment was scoped to the zeta-node repository. A related audit covering the Solidity smart contracts was also performed with this audit, and the resulting report should be viewed as a companion to this one.

1.2 AUDIT SUMMARY

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that the Cosmos chain and related smart contracts operate as intended.
- Identify potential security issues within the system.
- Respect the system security and invariants as defined in the ZetaChain whitepaper.

In summary, Halborn identified some security risks to be addressed by the ZetaChain team before the project is deployed in a live environment. The main areas of concern are the following:

- The process of minting and burning Zeta should be reviewed thoroughly in order to maintain the integrity of the system. Manual testing should be supplemented by unit tests and invariant testing. Currently, routine operations appear to inflate the supply of Zeta, which is unsuitable for a token that is meant to directly represent quantities of other tokens, such as Bitcoin, Ether, and ERC20 tokens.

- Concepts in the system such as block heights, fees, and prices should each have a consistent type, such as int64 or uint64. Type conversions can lead to overflow, underflow, and truncation errors that could seriously harm the system. Special attention should be paid to use of these values when they cross an interface from one software ecosystem to another (such as Bitcoin to CosmosSDK, or Solidity to Go).
- It is advised for the project to subject more functionality to governance processes in order to ensure proper operation and security.
- ZetaChain must take extra care to ensure all of its dependencies are up-to-date and secure. Packages such as Bitcoin core, CosmosSDK, the Go programming language, and the tss-lib are notable examples.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the custom modules. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of structures and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose.
- Static Analysis of security for scoped repository, and imported functions. (e.g., staticcheck, gosec, unconvert, codeql, ineffassign and semgrep)
- Manual Assessment for discovering security vulnerabilities on codebase.
- Ensuring correctness of the codebase.
- Dynamic Analysis on files and modules related to ZetaChain.
- Custom fuzz testing using Go's built-in fuzzing tools.

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - **8** - HIGH
7 - **6** - MEDIUM
5 - **4** - LOW

3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

The audit was scoped to ZetaChain's zeta-node repository at the following URL

- https://github.com/zeta-chain/zeta-node/tree/pre-audit-review

The review was conducted on the following commit hash:

- bf5aa35ed2258e9d12a92578faaf5ece991c74e7

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
4	5	6	16	13

LIKELIHOOD

IMPACT

			(HAL-05)	(HAL-01) (HAL-02) (HAL-03) (HAL-04)
	(HAL-12)		(HAL-06) (HAL-07) (HAL-08) (HAL-09)	
(HAL-26) (HAL-29)		(HAL-10) (HAL-13) (HAL-15)		
	(HAL-16) (HAL-17) (HAL-18) (HAL-22) (HAL-23) (HAL-24) (HAL-27) (HAL-28) (HAL-37)		(HAL-14)	
(HAL-32) (HAL-33) (HAL-34) (HAL-35) (HAL-36) (HAL-38) (HAL-39) (HAL-40) (HAL-41) (HAL-42) (HAL-43) (HAL-44)	(HAL-31)	(HAL-19) (HAL-20) (HAL-21)	(HAL-25) (HAL-30)	(HAL-11)

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - ZETA SUPPLY DOES NOT TRACK ASSETS CORRECTLY	Critical	-
HAL-02 - OVERFLOW IN ZETA BLOCK HEIGHT CAUSES EXPONENTIAL INCREASE IN GAS PRICE FOR ALL PENDING TRANSACTIONS	Critical	-
HAL-03 - POSSIBLE DIVISION BY ZERO COULD CAUSE CHAIN HALT DUE TO PANIC	Critical	-
HAL-04 - BITCOIN TRANSACTIONS REQUIRE ONLY ONE CONFIRMATION	Critical	-
HAL-05 - LACK OF MECHANISM TO LIMIT SUPPLY OF ZETA	High	-
HAL-06 - PRICE MANIPULATION AND DENIAL-OF-SERVICE VIA UpdatePrices FUNCTION	High	-
HAL-07 - ERROR CONDITION FOR KEY SIGNING IS UNCHECKED	High	-
HAL-08 - ITERATION OVER MAPS MAY BE A SOURCE OF NON-DETERMINISM	High	-
HAL-09 - SYBIL ATTACK RISK DUE TO USE OF MEDIAN GAS VOTES FOR SETTING GAS PRICE	High	-
HAL-10 - MALICIOUS GAS PRICE VOTING: DENIAL-OF-SERVICE BY SETTING LARGE GAS PRICES FOR EVM NETWORKS	Medium	-
HAL-11 - MALICIOUS GAS PRICE VOTING: DENIAL-OF-SERVICE OR PRICE MANIPULATION BY SETTING GAS PRICES FOR BITCOIN	Medium	-
HAL-12 - INTEGER OVERFLOW BREAKS GRPC COMMUNICATION FOR LARGE BLOCK HEIGHTS	Medium	-
HAL-13 - RELIANCE ON UNISWAPV2 POOLS FOR PRICES EXPOSE ZETACHAIN TO PRICE MANIPULATION RISK	Medium	-

HAL-14 - ARBITRARY MINTING OF ZETA VIA MintZetaToEVMAccount FUNCTION	Medium	-
HAL-15 - SUPPORT FOR A TOKEN CANNOT BE REMOVED FROM THE PROTOCOL	Medium	-
HAL-16 - USE OF VULNERABLE COSMOSSDK VERSION	Low	-
HAL-17 - ValidateBasic INCOMPLETE FOR SOME MESSAGE TYPES	Low	-
HAL-18 - CENTRALIZATION RISK	Low	-
HAL-19 - LACK OF UNIT TESTS	Low	-
HAL-20 - LACK OF FUZZ TESTING	Low	-
HAL-21 - BITCOIN TRANSACTIONS WITH LARGE NONCE VALUES MAY BE SENT REPEATEDLY	Low	-
HAL-22 - UNBOUNDED ARRAY IN Signers IN ChainNonces COULD CAUSE RESOURCE EXHAUSTION	Low	-
HAL-23 - USE OF VULNERABLE DEPENDENCIES	Low	-
HAL-24 - MNEMONIC PHRASES PRESENT IN CODEBASE	Low	-
HAL-25 - VULNERABLE TSS-LIB VERSION CONTAINS HASH COLLISION ISSUES	Low	-
HAL-26 - CALCULATION ERRORS IN getSatoshis FUNCTION FOR EXTREMELY SMALL OR LARGE FLOAT VALUES	Low	-
HAL-27 - GAS LIMITS CANNOT BE CONFIGURED	Low	-
HAL-28 - VALIDATORS CANNOT EFFECTIVELY SET THE ZETA GAS PRICE FOR BITCOIN TRANSACTIONS	Low	-
HAL-29 - GO VERSIONS PRIOR TO 1.20.2 CONTAIN CRYPTOGRAPHIC ISSUES AND OTHER BUGS	Low	-

HAL-30 - ZETACLIENT AND ZETACORE TRACK BLOCK HEIGHT USING DIFFERENT TYPES	Low	-
HAL-31 - USE OF DEPRECATED GO VERSION	Informational	-
HAL-32 - TYPE CONVERSION ISSUE FOR FIELD Decimals ON STRUCT MsgDeployFungibleCoinZRC20	Informational	-
HAL-33 - UNUSED FIELD GASLIMIT ON STRUCT MsgDeployFungibleCoinZRC20	Informational	-
HAL-34 - INTEGER OVERFLOW CONVERSION FOR GAS PRICE IN BITCOIN SIGNER	Informational	-
HAL-35 - REFERENCES TO DEPRECATED ETHEREUM NETWORK	Informational	-
HAL-36 - MESSAGE QUERYBALLOTBYIDENTIFIERREQUEST RETURNS RESULTS FOR BALLOTS THAT DO NOT EXIST	Informational	-
HAL-37 - DOCKER FILES USES A DIFFERENT GO VERSION THAN THE PROJECT	Informational	-
HAL-38 - UPGRADING TO A MORE RECENT VERSION OF COSMOSSDK COULD INCREASE PERFORMANCE	Informational	-
HAL-39 - TESTING ENVIRONMENT IS USING OUTDATED BITCOIN DAEMON	Informational	-
HAL-40 - GAS PRICE VOTER CLI INTERFACE USES THE WRONG TYPE	Informational	-
HAL-41 - DOCKER IGNORE FILE SHOULD INCLUDE GIT FILES	Informational	-
HAL-42 - TODOS IN CODEBASE	Informational	-
HAL-43 - SPELLING MISTAKES IN CODE BASE	Informational	-
HAL-44 - INCORRECT CODE COMMENTS	Informational	-

FINDINGS & TECH DETAILS

3.1 (HAL-01) ZETA SUPPLY DOES NOT TRACK ASSETS CORRECTLY - CRITICAL

Description:

The Zeta token represents the value of tokens linked to ZetaChain. ZetaChain uses a mint-and-burn model to maintain a fixed total supply of Zeta across chains. In order to properly represent various assets and their values, the total supply must not increase. This requirement is outlined in more detail in the ZetaChain whitepaper, section 7. As a result, the stability of the supply of Zeta is a critical part of the system's proper functioning as a whole. During the audit, we considered the property that the total supply of Zeta must not increase as a system invariant that must never be violated.

The Zeta team has created a set of tests to verify that high-level actions succeed within the network. These actions replicate a user's view of interacting with the protocol and include examples like deposit Bitcoin into Zeta or transfer Zeta messages within the EVM.

During the audit, Halborn modified the existing tests to examine the total supply of Zeta (measured as the sum of Cosmos-native azeta and wrapped Zeta in the EVM) before and after actions occur.

In many cases, it was discovered that the total supply of Zeta increases between high-level actions. This was observed both in cases where Zeta crosses one blockchain interface to another, and operations within a single blockchain environment, for example an EVM-to-EVM operation.

Due to this unintended inflation of the core token, the protocol cannot safely track external assets in its current state.

Code Location & Proof Of Concept:

The set of tests were modified to print the following values before and after each test: Total azeta in Cosmos, Total wZeta in the EVM, and the sum of these values both before and after a given test. Otherwise, the existing tests were not modified.

The following functions were added to contrib/localnet/orchestrator/smoketest/utils.go in order to test invariants.

Listing 1

```go
// HAL: Utility method to print the total supply of token azeta between runs
func GetTotalZetaCosmos() *big.Int {
    grpcConn, err := grpc.Dial(
        "zetacore0:9090",
        grpc.WithInsecure(),
        grpc.WithDefaultCallOptions(grpc.ForceCodec(codec.NewProtoCodec(nil).GRPCCodec())),
    )
    if err != nil {
        panic(fmt.Sprintf("Could not create gRPC connection. Error: `%s`", err.Error()))
    }
    defer grpcConn.Close()

    bankClient := banktypes.NewQueryClient(grpcConn)
    bankRes, err := bankClient.TotalSupply(
        context.Background(),
        &banktypes.QueryTotalSupplyRequest{
            //Key:        []byte(),
            //Offset:     0,
            //Limit:      100,
            //CountTotal: false,
            //Reverse:    false,
        },
    )
    if err != nil {
        panic(fmt.Sprintf("Could not query bank supply. Error: `%s`", err.Error()))
    }

    amountString := strings.TrimSuffix(bankRes.Supply.String(), "azeta") // Response supply string is in format like '200010000200000000000000000azeta'. Remove suffix
    supply := new(big.Int)
    _, ok := supply.SetString(amountString, 10)
```

```go
    if !ok {
        panic("Problem parsing azeta supply string to big Int")
    }
    return supply
}

// use sm here to make use of zevmClient
func GetTotalZetaEVM(sm *SmokeTest) *big.Int {
    // Taken from test_zeta_in_and_out.go
    wzetaAddr := ethcommon.HexToAddress("0x5F0b1a82749cb4E2278EC87F8BF6B618dC71a8bf")
    zevmClient := sm.zevmClient
    wzeta, err := zevm.NewWZETA(wzetaAddr, zevmClient)
    if err != nil {
        panic(fmt.Sprintf("Problem calling newWZeta. Error: `%s`", err.Error()))
    }
    supply, err := wzeta.TotalSupply(&bind.CallOpts{}) // returns a big.Int (contracts/zevm/WZETA.go)
    if err != nil {
        panic(fmt.Sprintf("Problem querying total supply of wzeta in EVM. Error: `%s`", err.Error()))
    }
    return supply
}

func InvariantTotalSupplyIncreased(previousSupply *big.Int, totalCosmos *big.Int, totalEVM *big.Int) (newSupply *big.Int, err error) {
    fmt.Printf("Total azeta in Cosmos: %d\n", totalCosmos)
    fmt.Printf("Total Zeta in EVM: %d\n", totalEVM)
    newSupply = new(big.Int)
    newSupply = newSupply.Add(totalCosmos, totalEVM) // totalCosmos + totalEVM
    fmt.Printf("Previous supply: %d\n", previousSupply)
    fmt.Printf("New supply: %d\n", newSupply)
    if newSupply == nil {
        panic("New supply should never be nil in InvariantTotalSupplyIncreased")
    }
    if newSupply.Cmp(previousSupply) > 0 { // newSupply > previousSupply
        return newSupply, errors.New(fmt.Sprintf("*** INVARIANT VIOLATED***\nTotal supply has increased from %v to %v\n(
```

```
↳ azetaSupply %v + wzetaSupply %v)\n", previousSupply, newSupply,
↳ totalCosmos, totalEVM))
    }
    return newSupply, nil
}

func PostTestInvariantCheck(previousSupply *big.Int, totalCosmos *
↳ big.Int, totalEVM *big.Int) (newSupply *big.Int, err error) {
    newSupply, err = InvariantTotalSupplyIncreased(previousSupply,
↳  totalCosmos, totalEVM)
    if err != nil {
        LoudPrintf(err.Error())
        return newSupply, err
    }
    if newSupply == nil {
        panic("New supply returned from
↳ InvariantTotalSupplyIncreased is nil")
    }
    return newSupply, nil
}
```


The main file containing the smoke tests (contrib/localnet/orchestrator/smoketest/main.go) was modified to print the total supply before and after:

Listing 2: Modifications to smoke tests in main.go

```
    smokeTest := NewSmokeTest(goerliClient, zevmClient, cctxClient
↳ , fungibleClient, goerliAuth, zevmAuth, btcRPCClient)
    totalEVM := GetTotalZetaEVM(smokeTest)
    totalCosmos := GetTotalZetaCosmos()
    supply := new(big.Int)
    supply = supply.Add(totalEVM, totalCosmos)
    LoudPrintf("Initial supplies: EVM = %v \nCosmos = %v \nTotal =
↳  %v\n", totalEVM, totalCosmos, supply)
    // The following deployment must happen here and in this order
↳ , please do not change
    // ==================== Deploying contracts
↳ ====================
    startTime := time.Now()
    smokeTest.TestBitcoinSetup()
    supply, totalCosmos, totalEVM = InvariantCheck(smokeTest,
↳ supply)
```

```
    smokeTest.TestSetupZetaTokenAndConnectorContracts()
    supply, totalCosmos, totalEVM = InvariantCheck(smokeTest, supply)
    ...

```{language=go caption="Function added to smoke test code in main.go}```
func InvariantCheck(sm *SmokeTest, previousSupply *big.Int) (newSupply *big.Int, totalCosmos *big.Int, totalEVM *big.Int) {
    if previousSupply == nil {
        panic("Previous supply should never be nil in InvariantCheck")
    }
    totalEVM = GetTotalZetaEVM(sm)
    totalCosmos = GetTotalZetaCosmos()
    newSupply, _ = PostTestInvariantCheck(previousSupply, totalCosmos, totalEVM)
    return
}
```

An example of the supply inflation occurs in the following excerpt from the test logs:

Listing 3: Excerpt from smoke test showing intial supply and first invariant violation

```
2023-03-30 14:52:22 =======================================
2023-03-30 14:52:22 Initial supplies: EVM = 1000000000100000000
2023-03-30 14:52:22 Cosmos = 20001000020000000000000000
2023-03-30 14:52:22 Total = 20001000030000000010000000
2023-03-30 14:52:22 =======================================
...
2023-03-30 14:53:48 Total azeta in Cosmos: 20001000020000000000000000
2023-03-30 14:53:48 Total Zeta in EVM: 1000000000100000000
2023-03-30 14:53:48 Previous supply: 20001000030000000010000000
2023-03-30 14:53:48 New supply: 20001000030000000010000000
2023-03-30 14:53:48 =======================================
2023-03-30 14:53:48 Step 3: Sending ZETA to ZetaChain
2023-03-30 14:53:48 =======================================
...
2023-03-30 14:54:06 Total azeta in Cosmos:
```
```

```
↳ 200010100200000000000000000
2023-03-30 14:54:06 Total Zeta in EVM: 100000000010000000
2023-03-30 14:54:06 Previous supply: 200010000300000000010000000
2023-03-30 14:54:06 New supply: 200010100300000000010000000
2023-03-30 14:54:06 ========================================
2023-03-30 14:54:06 *** INVARIANT VIOLATED***
2023-03-30 14:54:06 Total supply has increased from
↳ 200010000300000000010000000 to 200010100300000000010000000
2023-03-30 14:54:06 (azetaSupply 200010100200000000000000000 +
↳ wzetaSupply 100000000010000000)
2023-03-30 14:54:06 ========================================
```

Risk Level:

Likelihood - 5
Impact - 5

Recommendation:

Add strict controls on the total supply of Zeta so that the supply does not increase. This invariant must be encoded into the Cosmos code and incorporated into smoke tests.

3.2 (HAL-02) OVERFLOW IN ZETA BLOCK HEIGHT CAUSES EXPONENTIAL INCREASE IN GAS PRICE FOR ALL PENDING TRANSACTIONS - CRITICAL

Description:

x/crosschain/keeper/end_block_scrub_send.go. changes gas prices for stuck transactions. There is a cast to int64 on Line 40 that can overflow for large values of cctx.InboundTxParams.InboundTxFinalizedZetaHeight. The result of this expression will be a large negative number. When this number is subtracted from the Block Height on this line, the result will be a large positive number, causing the if-statement to always be true. This will take every pending transaction and multiply its gas cost by 20%. This function is called in EndBlock so the gas price of all pending transactions will be multiplied by 20% each block, causing the gas price to compound and double every 3-4 blocks. Eventually this will have the effect of making the chain unusable.

It is important to note that there does not appear to be a location in the codebase where InboundTxFinalizedZetaHeight is user-controlled. This case will arise for very large block heights, though it will take a long time for this to occur in practice. However, it is recommended addressing this issue as it will eventually arise in normal ZetaChain operation in the long term and could be exploited if InboundTxFinalizedZetaHeight becomes user-controlled during future development of the codebase.

Code Location:

x/crosschain/module.go

Listing 4: Call to ScrubGasPriceOfStuckOutTx in EndBlock, which in turn calls vulnerable function ScrubUtility

```
func (am AppModule) EndBlock(ctx sdk.Context, _ abci.RequestEndBlock) []abci.ValidatorUpdate {
    //am.keeper.InitializeGenesisKeygen(sdk.WrapSDKContext(ctx))
    am.keeper.ScrubGasPriceOfStuckOutTx(sdk.WrapSDKContext(ctx))
    return []abci.ValidatorUpdate{}
}
```

Listing 5: Overflow occurs when converting large values of InboundTxFinalizedZetaHeight to int64

```
func (k Keeper) ScrubUtility(ctx sdk.Context, store sdk.KVStore, p []byte) {
    sendStore := prefix.NewStore(store, p)
    iterator := sdk.KVStorePrefixIterator(sendStore, []byte{})
    defer iterator.Close()
    for ; iterator.Valid(); iterator.Next() {
        var cctx types.CrossChainTx
        k.cdc.MustUnmarshal(iterator.Value(), &cctx)
        // if the status of send is pending, which means Finalized/Revert
        if cctx.CctxStatus.Status == types.CctxStatus_PendingOutbound || cctx.CctxStatus.Status == types.CctxStatus_PendingRevert {
            if ctx.BlockHeight()-int64(cctx.InboundTxParams.InboundTxFinalizedZetaHeight) > 100 { // stuck send
                var chainID int64
                currentOutTxParam := cctx.GetCurrentOutTxParam()
                chainID = currentOutTxParam.ReceiverChainId

                gasPrice, isFound := k.GetGasPrice(ctx, chainID)
                if !isFound {
                    continue
                }
                mi := gasPrice.MedianIndex
                newGasPrice := big.NewInt(0).SetUint64(gasPrice.Prices[mi])
                oldGasPrice, ok := big.NewInt(0).SetString(currentOutTxParam.OutboundTxGasPrice, 10)
                if !ok {
                    k.Logger(ctx).Error("failed to parse old gas price")
```

```
                    continue
                }
                // do nothing if new gas price is even lower than old price
                if newGasPrice.Cmp(oldGasPrice) < 0 {
                    continue
                }
                targetGasPrice := oldGasPrice.Mul(oldGasPrice, big.NewInt(4))
                targetGasPrice = targetGasPrice.Div(targetGasPrice, big.NewInt(3)) // targetGasPrice = oldGasPrice * 1.2
                // if current new price is not much higher; make it at least 20% higher
                // otherwise replacement tx will be rejected by the node
                if newGasPrice.Cmp(targetGasPrice) < 0 {
                    newGasPrice = targetGasPrice
                }
                currentOutTxParam.OutboundTxGasPrice = newGasPrice.String()
                // No need to migrate as this function does not change the status of Send
                k.SetCrossChainTx(ctx, cctx)
                EmitCCTXScrubbed(ctx, cctx, chainID, oldGasPrice.String(), newGasPrice.String())
            }
        }
    }
}
```


Risk Level:

Likelihood - 5
Impact - 5

Recommendation:

Avoid converting InboundTxFinalizedZetaHeight to int64 and use uint64 instead.

This can be addressed by using unsigned integers rather than signed integers in the calculation on Line 40. This should be appropriate as both terms of the expression represent Block Height, and thus it is not valid for them to have negative values.

3.3 (HAL-03) POSSIBLE DIVISION BY ZERO COULD CAUSE CHAIN HALT DUE TO PANIC - CRITICAL

Description:

The function GetBondFactor x/emissions/abci.go uses the Quo method to divide the return value of the CosmosSDK function stakingKeeper.BondedRatio(). As it is possible for the function BondedRatio to return 0 in some cases (see: cosmos-sdk@v0.46.8/x/staking/keeper/pool.go), this function could panic for some chain states when a division by zero is attempted.

GetBondFactor is called by a BeginBlock function. Therefore, a panic in this location could cause a chain halt.

Code Location:

x/emissions/abci.go

Listing 6: BeginBlocker calls GetBlockRewardComponents which in turn calls the vulnerable function

```go
func BeginBlocker(ctx sdk.Context, keeper keeper.Keeper, stakingKeeper types.StakingKeeper, bankKeeper types.BankKeeper) {
    //fmt.Println("Executing begin block emisson")
    reservesFactor, bondFactor, durationFactor := GetBlockRewardComponents(ctx, bankKeeper, stakingKeeper, keeper)
```

Listing 7: This function calls GetBondFactor

```go
func GetBlockRewardComponents(ctx sdk.Context, bankKeeper types.BankKeeper, stakingKeeper types.StakingKeeper, emissionKeeper keeper.Keeper) (sdk.Dec, sdk.Dec, sdk.Dec) {
    reservesFactor := GetReservesFactor(ctx, bankKeeper)
    if reservesFactor.LTE(sdk.ZeroDec()) {
        return sdk.ZeroDec(), sdk.ZeroDec(), sdk.ZeroDec()
    }
```

```
    bondFactor := GetBondFactor(ctx, stakingKeeper, emissionKeeper)
    durationFactor := GetDurationFactor(ctx, emissionKeeper)
    return reservesFactor, bondFactor, durationFactor
}
```

Listing 8: Possible divide-by-zero via call to Quo with result from stakingKeeper.BondedRatio

```
func GetBondFactor(ctx sdk.Context, stakingKeeper types.StakingKeeper, keeper keeper.Keeper) sdk.Dec {
    targetBondRatio := sdk.MustNewDecFromStr(keeper.GetParams(ctx).TargetBondRatio)
    maxBondFactor := sdk.MustNewDecFromStr(keeper.GetParams(ctx).MaxBondFactor)
    minBondFactor := sdk.MustNewDecFromStr(keeper.GetParams(ctx).MinBondFactor)

    currentBondedRatio := stakingKeeper.BondedRatio(ctx)
    // Bond factor ranges between minBondFactor (0.75) to maxBondFactor (1.25)
    bondFactor := targetBondRatio.Quo(currentBondedRatio)
    if bondFactor.GT(maxBondFactor) {
        return maxBondFactor
    }
    if bondFactor.LT(minBondFactor) {
        return minBondFactor
    }
    return bondFactor
}
```


Proof Of Concept:

- Analyze the GetBondFactor function in x/emissions/abci.go and identify the point where the division by zero could occur due to the stakingKeeper.BondedRatio() function returning 0. Understand the impact of a panic caused by the division by zero on the chain's stability and operations.
- Interact with the chain and call the GetBondFactor function using the prepared chain state. This will cause the function to use the Quo

method to divide by the return value of stakingKeeper.BondedRatio(), which is 0 in this case.

Risk Level:

Likelihood - 5
Impact - 5

Recommendation:

There are other cases of explicit panics that occur in the emissions module's BeginBlocker method, so we understand that any failure in payouts is considered critical by ZetaChain and a chain halt is desired in these cases as a defensive move. If a BondedRatio of 0 is also considered a critical state, we recommend using an explicit panic to indicate this rather than allowing a division-by-zero error to cause the panic. This will clarify the intention of the code. On the other hand, if a BondedRatio of 0 is not critical, the code should be rewritten to return early.

In any case, the code should not attempt to divide by zero.

3.4 (HAL-04) BITCOIN TRANSACTIONS REQUIRE ONLY ONE CONFIRMATION - CRITICAL

Description:

The function IsSendOutTxProcessed in bitcoin_client.go requires only one confirmation for the transaction to be considered as processed by Zeta. The number of confirmations for a transaction refers to the number of blocks that are added to the blockchain after the block containing a given transaction. If a transaction has one confirmation, this means that only one block follows the block containing that transaction.

Transactions with a low number of confirmations may be at risk of being reversed due to the details of Bitcoin's consensus mechanism. There is a risk that a different history of the blockchain that does not contain this transaction will ultimately become the most widely used block sequence. This risk always exists, but a greater number of confirmations reduces the risk that a transaction will be discarded by the network.

In the case of ZetaChain, if only one confirmation is required for a transaction to be confirmed, the protocol is at a relatively high risk of processing transactions that will ultimately be reversed. This could cause the ZetaChain's accounting to become incorrect and ultimately could result in Bitcoin transactions failing due to miscalculations of transaction inputs and outputs in the Bitcoin network.

Code Location:

bitcoin_client.go

Listing 9: The transaction is considered processed so long as res.Confirmations is greater than zero.

```
func (ob *BitcoinChainClient) IsSendOutTxProcessed(sendHash string
 ↳ , nonce int, _ common.CoinType) (bool, bool, error) {
```

```go
    chain := ob.chain.ChainId
    outTxID := fmt.Sprintf("%d-%d", chain, nonce)
    ob.logger.Info().Msgf("IsSendOutTxProcessed %s", outTxID)

    res, found := ob.submittedTx[outTxID]
    if !found {
        return false, false, nil
    }
    if res.Confirmations == 0 {
        return true, false, nil
    } else if res.Confirmations > 0 { // FIXME: use configured block confirmation
        amountInSat, _ := big.NewFloat(res.Amount * 1e8).Int(nil)
        zetaHash, err := ob.zetaClient.PostReceiveConfirmation(
            sendHash,
            res.TxID,
            uint64(res.BlockIndex),
            amountInSat,
            common.ReceiveStatus_Success,
            ob.chain,
            nonce,
            common.CoinType_Gas,
        )
        if err != nil {
            ob.logger.Error().Err(err).Msgf("error posting to zeta core")
        } else {
            ob.logger.Info().Msgf("Bitcoin outTx confirmed: PostReceiveConfirmation zeta tx: %s", zetaHash)
        }
        return true, true, nil
    }
    return false, false, nil
}
```


Proof Of Concept:

- Prepare a double-spending attack on the ZetaChain network. To achieve this, create two conflicting transactions: one that sends the same Bitcoin funds to a legitimate recipient and another that sends the same funds to an attacker-controlled address.
- Broadcast the legitimate transaction to the Bitcoin network, ensur-

ing that it enters the mempool and eventually gets added to a block. Monitor the transaction to ensure that it receives one confirmation.
- With only one confirmation, the IsSendOutTxProcessed function would consider the transaction as processed. Exploit this vulnerability by using the ZetaChain protocol to process the transaction and update its internal accounting.

Risk Level:

Likelihood - 5
Impact - 5

Recommendation:

It is recommended to use a higher number of confirmations, especially for transactions with larger values. An algorithm could be developed to scale the number of confirmations alongside the value of the transactions.

The confirmation number should likely be determined via governance so that stakeholders can determine the level of risk they wish to incur.

FINDINGS & TECH DETAILS

3.5 (HAL-05) LACK OF MECHANISM TO LIMIT SUPPLY OF ZETA - HIGH

Description:

No source of truth across the protocol for the total supply of Zeta minted. The whitepaper mentions that Chainlink will be used in the future, but this is not yet implemented and there is no alternative mechanism.

As a result, the protocol can mint an unbounded number of Zeta tokens in either the Cosmos environment (in the form of the protocol's main token azeta) or in the EVM environment in the form of wrapped Zeta.

This undermines the integrity of the system as the supply of Zeta is meant to represent other tokens, such as Bitcoin and Ether, and transfer their value across blockchains. If the supply is not capped or checked in any way, the risks of minting extra Zeta greatly increase. This in turn increases the likelihood that the amount of foreign tokens represented in the system will not correspond to actual assets in the supported chains.

Risk Level:

Likelihood - 4
Impact - 5

Recommendation:

It is advised to implement a mechanism to cap the supply of Zeta on the Cosmos side. When the code interfaces with an EVM environment, strict checks must be performed such that the total supply of Zeta and wrapped Zeta across networks never increases.

3.6 (HAL-06) PRICE MANIPULATION AND DENIAL-OF-SERVICE VIA UpdatePrices FUNCTION - HIGH

Description:

The UpdatePrices function checks whether a sufficient fee has been paid in order to carry out a given transaction. However, this check occurs after certain critical side effects have taken place. There are three state changes to the overall system: Coins are minted, an asset swap occurs in an Uniswap pool (and thus the prices of both assets are changed, and ZRC20 tokens are burned.

This violates the checks-effects pattern where prerequisites for a state change must occur and cause early termination before the state is changed. In this case, it is possible that an attacker would be able to drastically undermine system integrity by spamming transactions with insufficient gas. While these transactions will eventually cause an error, the attacker will succeed in modifying prices and shifting tokens from the EVM to the Cosmos environment (via burning EVM tokens and minting Cosmos tokens). This could result in price manipulation for the attacker's benefit or service interruptions to the protocol if the EVM tokens are depleted to a point where normal operations cannot occur.

Code Location:

The UpdatePrices function performs error-checking, but the last condition checked is whether sufficient fees are attached to the transaction in question. By this time, three critical side effects have occurred:

- Minting new Zeta
- Swapping coins in an Uniswap pool (changing price)
- Burning ZRC20 tokens

Listing 10: Calls to mint, swap, and burn coins occur before checking if fees are sufficient.

```go
func (k Keeper) UpdatePrices(ctx sdk.Context, chainID int64, cctx *types.CrossChainTx) error {
    chain := k.zetaObserverKeeper.GetParams(ctx).GetChainFromChainID(chainID)
    medianGasPrice, isFound := k.GetMedianGasPriceInUint(ctx, chain.ChainId)
    if !isFound {
        return sdkerrors.Wrap(types.ErrUnableToGetGasPrice, fmt.Sprintf(" chain %d | Identifiers : %s ", cctx.GetCurrentOutTxParam().ReceiverChainId, cctx.LogIdentifierForCCTX()))
    }
    cctx.GetCurrentOutTxParam().OutboundTxGasPrice = medianGasPrice.String()
    gasLimit := sdk.NewUint(cctx.GetCurrentOutTxParam().OutboundTxGasLimit)
    outTxGasFee := gasLimit.Mul(medianGasPrice)

    // the following logic computes outbound tx gas fee, and convert into ZETA using system uniswapv2 pool wzeta/gasZRC20
    gasZRC20, err := k.fungibleKeeper.QuerySystemContractGasCoinZRC4(ctx, big.NewInt(chain.ChainId))
    if err != nil {
        return sdkerrors.Wrap(err, "UpdatePrices: unable to get system contract gas coin")
    }
    outTxGasFeeInZeta, err := k.fungibleKeeper.QueryUniswapv2RouterGetAmountsIn(ctx, outTxGasFee.BigInt(), gasZRC20)
    if err != nil {
        return sdkerrors.Wrap(err, "UpdatePrices: unable to QueryUniswapv2RouterGetAmountsIn")
    }
    feeInZeta := types.GetProtocolFee().Add(math.NewUintFromBigInt(outTxGasFeeInZeta))

    // swap the outTxGasFeeInZeta portion of zeta to the real gas ZRC20 and burn it
    coins := sdk.NewCoins(sdk.NewCoin(config.BaseDenom, sdk.NewIntFromBigInt(feeInZeta.BigInt())))
    err = k.bankKeeper.MintCoins(ctx, types.ModuleName, coins)
    if err != nil {
        return sdkerrors.Wrap(err, "UpdatePrices: unable to mint
```

```
coins")
    }
    amounts, err := k.fungibleKeeper.CallUniswapv2RouterSwapExactETHForToken(ctx, types.ModuleAddressEVM, types.ModuleAddressEVM, outTxGasFeeInZeta, gasZRC20)
    if err != nil {
        return sdkerrors.Wrap(err, "UpdatePrices: unable to CallUniswapv2RouterSwapExactETHForToken")
    }
    ctx.Logger().Info("gas fee", "outTxGasFee", outTxGasFee, "outTxGasFeeInZeta", outTxGasFeeInZeta)
    ctx.Logger().Info("CallUniswapv2RouterSwapExactETHForToken", "zetaAmountIn", amounts[0], "zrc20AmountOut", amounts[1])
    err = k.fungibleKeeper.CallZRC20Burn(ctx, types.ModuleAddressEVM, gasZRC20, amounts[1])
    if err != nil {
        return sdkerrors.Wrap(err, "UpdatePrices: unable to CallZRC20Burn")
    }

    cctx.ZetaFees = cctx.ZetaFees.Add(feeInZeta)

    if cctx.ZetaFees.GT(cctx.InboundTxParams.Amount) && cctx.InboundTxParams.CoinType == common.CoinType_Zeta {
        return sdkerrors.Wrap(types.ErrNotEnoughZetaBurnt, fmt.Sprintf("feeInZeta(%s) more than zetaBurnt (%s) | Identifiers : %s ", cctx.ZetaFees, cctx.InboundTxParams.Amount, cctx.LogIdentifierForCCTX()))
    }
    cctx.GetCurrentOutTxParam().Amount = cctx.InboundTxParams.Amount.Sub(cctx.ZetaFees)

    return nil
}
```


Risk Level:

Likelihood - 4

Impact - 4

Recommendation:

Ensure that all preconditions for a piece of logic are met before executing the logic. In this case, the fees should be checked before any swaps, minting, or burns take place.

3.7 (HAL-07) ERROR CONDITION FOR KEY SIGNING IS UNCHECKED - HIGH

Description:

The function TestKeysign returns an error, but the error is not checked. If this function fails, it could have serious impacts on the protocol, as operation may continue after a failed signature.

Code Location:

zetaclient/tss_signer.go

Listing 11: TestKeysign has several cases where an error may be returned

```go
func TestKeysign(tssPubkey string, tssServer *tss.TssServer) error {
    log.Info().Msg("trying keysign...")
    data := []byte("hello meta")
    H := crypto.Keccak256Hash(data)
    log.Info().Msgf("hash of data (hello meta) is %s", H)

    keysignReq := keysign.NewRequest(tssPubkey, []string{base64.StdEncoding.EncodeToString(H.Bytes())}, 10, nil, "0.14.0")
    ksRes, err := tssServer.KeySign(keysignReq)
    if err != nil {
        log.Warn().Msg("keysign fail")
    }
    signature := ksRes.Signatures
    // [{cyP8i/UuCVfQKDsLr1kpg09/CeIHje1FU6GhfmyMD5Q= D4jXTH3/CSgCg+9kLjhhfnNo3ggy9DTQSlloe3bbKAs= eY++Z2LwsuKG1JcghChrsEJ4u9grLloaaFZNtXI3Ujk= AA==}]
    // 32B msg hash, 32B R, 32B S, 1B RC
    log.Info().Msgf("signature of helloworld... %v", signature)

    if len(signature) == 0 {
        log.Info().Msgf("signature has length 0, skipping verify")
        return fmt.Errorf("signature has length 0")
    }
    verifySignature(tssPubkey, signature, H.Bytes())
```

```
    if verifySignature(tssPubkey, signature, H.Bytes()) {
        return nil
    }
    return fmt.Errorf("verify signature fail")
}
```

tss_signer.go is called by zetaclient/zetacore_observer.go on L117.

Listing 12: Errors from TestKeysign are discarded.

```
115                 // Keysign test: sanity test
116                 co.logger.Info().Msgf("test keysign...")
117                 _ = TestKeysign(co.tss.CurrentPubkey, co.tss.Server)
118                 co.logger.Info().Msg("test keysign finished. exit keygen loop. ")
```

Risk Level:

Likelihood - 4
Impact - 4

Recommendation:

Always check and confirm error cases. This is especially critical for code pertaining to cryptographic and monetary operations.

3.8 (HAL-08) ITERATION OVER MAPS MAY BE A SOURCE OF NON-DETERMINISM - HIGH

Description:

There are some instances in the codebase where an iteration over a map is performed. Map ordering is not deterministic in Go. As a result, iterations over maps can be a source of non-determinism. In a blockchain context, this can result in a chain halt if used in a consensus-critical context.

Code Location:

Map iterations are present at the following locations in the codebase:

Listing 13

```
app/app.go:635
app/app.go:743
app/setup_handlers.go:17
cmd/zetaclientd/main.go:315
zetaclient/out_tx_processor_manager.go:72
zetaclient/zetacore_observer.go:167
zetaclient/zetacore_observer.go:260
```

It is important to note that these iterations take place outside the CosmosSDK context and are present primarily outside a consensus context. However, this is considered a highly dangerous pattern that should be avoided.

Risk Level:

Likelihood - 4
Impact - 4

Recommendation:

Avoid iterating over maps where possible.

If this is necessary, ensure maps are sorted before iterating over them. Avoid returning early from the loop or making state-changing operations. Pay special attention to code that may cause side effects that could impact consensus.

3.9 (HAL-09) SYBIL ATTACK RISK DUE TO USE OF MEDIAN GAS VOTES FOR SETTING GAS PRICE - HIGH

Description:

ZetaChain uses the Zeta token as a way to represent and exchange value across blockchains. It is possible to create e.g. Bitcoin to Ethereum transactions by depositing Bitcoin, minting a corresponding amount of Zeta within Cosmos, and then doing a second mint of a wrapped version of Zeta within an EVM smart contract. Zeta can also be used to represent the gas price in native tokens across networks involved in this kind of transaction.

The gas price in Zeta for transactions is set by the validators. Validators automatically post the price of transactions involving Bitcoin. They can optionally post a price using a manual process for all other supported networks.

The method for determining the actual price is to take the median of all prices posted to the chain. This mechanism is problematic for two main reasons.

1. The gas price is subject to very large swings in prices. A change in the median calculation could result in a gigantic change in price in a relatively short time.
2. It is possible for a set of validators to collude and rig the price by posting a range of prices such that their target price falls at the median of all posted prices.

The gas price for a transaction on any network supported by ZetaChain is determined by a set of validators. For Bitcoin networks, all validators automatically post a price at a fixed interval based on a value returned by a Bitcoin RPC call. For EVM networks, validators can optionally publish a gas price. Validators that post a gas price are also referred to as

Signers. Each supported network has its price determined by a separate list of Signers.

The actual gas price used is the median of all the prices posted by validators. This creates an opportunity for risk, as a subset of validators can collude to set prices that are beneficial to themselves. They need to agree on a series of gas prices such that the ideal price lands at the median. This can be done as long as the colluding set of validators represents most Signers for a given network.

For Bitcoin, this number of colluding signers `Cn` must be equal to `Sn/2` where `Sn` is the total number of Signers; this effectively means most total validators in the protocol, as all validators are Signers for Bitcoin.

For EVM networks, the number `Cn = Sn/2` can be much smaller than `Vn/2` as validators do not automatically post prices. Put simply, if an attacker can control at least half of the signers for a network, they control the gas price for that network.

Note that this attack requirement is actually much easier to achieve than a normal 51% attack in blockchain protocols. The attacker only needs to control a fixed number of validators, irrespective of their share of overall stake. For example, if there are 5 validators only 3 validators need to collude. This is true even if the 2 honest validators control 99.99999% of the staking power. As a result, the cost of this attack in `azeta` equal to `minStake * Vn/2` (where `minStake` is the minimum threshold of staked `azeta` needed to become a validator and `Vn` is the total number of ZetaChain validators) plus a small amount of gas used to post new gas prices.

This ability to manipulate the gas price for a relatively low cost increases the severity of potential denial-of-service attacks, such as the ones explored in HAL-10 and HAL-11.

Code Location:

The code below demonstrates the mechanism for storing a gas price and selecting the true price based on the median.

x/crosschain/keeper/keeper_gas_price.go

Listing 14: The gas price is determined by selecting the median index of all posted prices

```go

func (k msgServer) GasPriceVoter(goCtx context.Context, msg *types.MsgGasPriceVoter) (*types.MsgGasPriceVoterResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)

    validators := k.StakingKeeper.GetAllValidators(ctx)
    if !IsBondedValidator(msg.Creator, validators) {
        return nil, sdkerrors.Wrap(sdkerrors.ErrorInvalidSigner, fmt.Sprintf("signer %s is not a bonded validator", msg.Creator))
    }
    chain := k.zetaObserverKeeper.GetParams(ctx).GetChainFromChainID(msg.ChainId)
    if chain == nil {
        return nil, sdkerrors.Wrap(types.ErrUnsupportedChain, fmt.Sprintf("ChainID : %d ", msg.ChainId))
    }

    gasPrice, isFound := k.GetGasPrice(ctx, chain.ChainId)
    if !isFound {
        gasPrice = types.GasPrice{
            Creator:     msg.Creator,
            Index:       strconv.FormatInt(chain.ChainId, 10), // TODO : Not needed index set at keeper
            ChainId:     chain.ChainId,
            Prices:      []uint64{msg.Price},
            BlockNums:   []uint64{msg.BlockNumber},
            Signers:     []string{msg.Creator},
            MedianIndex: 0,
        }
    } else {
        signers := gasPrice.Signers
        exist := false
        for i, s := range signers {
            if s == msg.Creator { // update existing entry
                gasPrice.BlockNums[i] = msg.BlockNumber
                gasPrice.Prices[i] = msg.Price
                exist = true
                break
            }
        }
```

```go
            if !exist {
                gasPrice.Signers = append(gasPrice.Signers, msg.Creator)
                gasPrice.BlockNums = append(gasPrice.BlockNums, msg.BlockNumber)
                gasPrice.Prices = append(gasPrice.Prices, msg.Price)
            }
            // recompute the median gas price
            mi := medianOfArray(gasPrice.Prices)
            gasPrice.MedianIndex = uint64(mi)
        }
        k.SetGasPrice(ctx, gasPrice)
        chainIDBigINT := big.NewInt(chain.ChainId)
        gasUsed, err := k.fungibleKeeper.SetGasPrice(ctx, chainIDBigINT, big.NewInt(int64(gasPrice.Prices[gasPrice.MedianIndex])))
        if err != nil {
            return nil, err
        }

        // reset the gas count
        k.ResetGasMeterAndConsumeGas(ctx, gasUsed)

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


For example, a malicious validator submits a bad gas price:

Listing 15

```
export PRICE="123456789"
zetacored tx crosschain gas-price-voter 18444 $PRICE 100 100 --keyring-backend=test --yes --chain-id=athens_101-1 --broadcast-mode=block --gas=auto --gas-adjustment=2 --gas-prices=0.1azeta --from=val
```

Check the gas prices.

Listing 16

```
zetacored q crosschain list-gas-price
```

Result: median_index == 1 and our price is included, meaning we control the gas price.

Listing 17

```
/usr/local/bin # zetacored q crosschain list-gas-price
GasPrice:
...
- block_nums:
  - "196"
  - "100"
  chain_id: "18444"
  creator: zeta1lz2fqwzjnk6qy48fgj753h48444fxtt7hekp52
  index: "18444"
  median_index: "1"
  prices:
  - "1000"
  - "123456789"
  signers:
  - zeta1lz2fqwzjnk6qy48fgj753h48444fxtt7hekp52
  - zeta1z46tdw75jvh4h39y3vu758ctv34rw5z9kmyhgz
pagination:
  next_key: null
  total: "0"
```

Risk Level:

Likelihood - 4
Impact - 4

Recommendation:

It is recommended using a time-weighted average price (TWAP) mechanism for calculating the gas prices in the chain. This would allow ZetaChain to reach a fair price for gas. Since there is no median index calculation, this removes the risk described in this finding.

3.10 (HAL-10) MALICIOUS GAS PRICE VOTING: DENIAL-OF-SERVICE BY SETTING LARGE GAS PRICES FOR EVM NETWORKS - MEDIUM

Description:

Validators have the option to post gas prices for various transactions in ZetaChain. If the protocol has a whole is not able to afford the gas price, the transaction will revert. As a result there is an opportunity for malicious validators to post gas prices to the protocol such that all EVM-related transactions will fail.

A colluding set of validators that coordinate on posting prices could manipulate gas prices in order to control when transactions are able to get through the system. This could be done by posting a very low price or a very high price: low prices will be insufficient on the level of the EVM to carry out transaction instructions; high prices will not be affordable by the protocol.

Depending on how many validators post prices, this attack could be done by a very small number of validators. For example even if there were 100 validators, if only 5 of them have ever posted prices on the network, a total of 3 validators could manipulate the price. This is described in detail in HAL-09.

Code Location:

See HAL-11 for an example exploit. The mechanism for attacking EVM is similar but would use a different chain ID.

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

When setting gas prices, consider imposing a cap. For example, it is unlikely that the gas price should ever approach the limits of the Uint64 data type.

Overall, we recommend using a TWAP mechanism rather than calculating the gas price based on the median value of posted prices.

3.11 (HAL-11) MALICIOUS GAS PRICE VOTING: DENIAL-OF-SERVICE OR PRICE MANIPULATION BY SETTING GAS PRICES FOR BITCOIN - MEDIUM

Description:

Similar to HAL-10, it is also possible for validators to cause uncertainty on the Bitcoin side of the network through gas price manipulation, though the mechanism for doing so is somewhat different than the EVM scenario.

As detailed in HAL-28, the Bitcoin price resets at a fixed interval (five seconds, as seen in zetaclient/config/config_mainnet.go). However, it is possible for validators to race this interval by posting new gas price updates at a shorter interval, for example every 2.5 seconds. This attack can cause a denial-of-service on the network. If a very low price is posted, the transaction will never be picked up by miners and so it will fail.

A simple exploit script for a scenario with two validators would look like this:

Listing 18: Post an extremely large gas price every 0.5 seconds

```
export PRICE="9223372036854775807"
while true; do
    zetacored tx crosschain gas-price-voter 18444 $PRICE 100 100
    --keyring-backend=test --yes --chain-id=athens_101-1 --broadcast-
    mode=block --gas=auto --gas-adjustment=2 --gas-prices=0.1azeta --
    from=val
    sleep 0.5
done
```

Listing 19: Response after querying the new gas price for the Bitcoin test network.

```
zetacored q crosschain list-gas-price
...
- block_nums:
  - "185"
  - "185"
  chain_id: "18444"
  creator: zeta1lz2fqwzjnk6qy48fgj753h48444fxtt7hekp52
  index: "18444"
  median_index: "1"
  prices:
  - "1000"
  - "9223372036854775807"
  signers:
  - zeta1lz2fqwzjnk6qy48fgj753h48444fxtt7hekp52
  - zeta1z46tdw75jvh4h39y3vu758ctv34rw5z9kmyhgz
```

The `median_index` is 1, so the extremely high price will be the gas price for Bitcoin transactions until the next round of automatic price updates.

Given that all nodes automatically post a price, this attack can occur if at least half of the number of signers collude and post a set of prices such that the target prices is selected as the median. (Note that the requirement of getting half of the validators to collude is easier to achieve than the typical definition of a 51% attack; see HAL-09 for details.) Compared to the related issue for EVM chains, this attack is somewhat more difficult to carry out as it would require at least half of all of the validators to continually race the automated price updates by posting their own prices.

Attackers can take one of two strategies when selecting the gas price. If the gas price is high but still affordable, a participant in the network may end up paying a very high fee for their transaction, resulting in a small resulting value on the receiving end of the transaction. If the gas price is instead set to an extremely high value, all transactions involving Bitcoin will be impossible. This is easy to achieve as the gas price is stored in a `uint256` variable. The highest number this type can store far exceeds the maximum supply of Bitcoin (even when it is represented in Satoshis). Therefore, the attackers can simply choose a gas price of MAX_UINT64 in order to stop all Bitcoin transactions.

Because of this behavior, it is impossible for honest validators to set a useful gas price for transactions that involve Bitcoin (as described in HAL-28). However, it is still possible for malicious validators to set inappropriate gas prices for Bitcoin transactions that effectively disable Bitcoin functionality in ZetaChain.

Code Location:

The sections of code that create the possibility for this attack are detailed in HAL-09 and HAL-28.

Risk Level:

Likelihood - 5
Impact - 1

Recommendation:

When setting gas prices, consider imposing a cap. For example, it is unlikely that the gas price should ever approach the limits of the Uint64 data type.

Overall, we recommend using a TWAP mechanism rather than calculating the gas price based on the median value of posted prices.

3.12 (HAL-12) INTEGER OVERFLOW BREAKS GRPC COMMUNICATION FOR LARGE BLOCK HEIGHTS - MEDIUM

Description:

The file x/crosschain/keeper/grpc_zevm.go contains an endpoint that allows for querying information about a block using its height. An unsigned integer is converted to a signed integer in two locations here, resulting in an overflow.
Typically, when a value for height is higher than the current block height, an error is thrown. However, when an overflow occurs, the code does not return an error and instead returns information about the latest block.

This could result in usability issues:

1. Other software that interacts with the node should correctly assume that they will get an error message on illegal block heights. They will for blockHeight < x <= MAX_INT64 but outside this range they will get unexpected behavior.
2. All queries will break when ZetaChain's real block heights become very large, as it will be impossible to query any blocks with a height > MAX_INT64. The latest block will always be returned.

Code Location:

The following code was used to query ZEVMGetBlock with various values and demonstrate the overflow issue. The resulting output follows.

Listing 20

```
package main

import (
    "context"
```

```go
	"fmt"
	"math"
	"strconv"
	"strings"

	"google.golang.org/grpc"

	"github.com/cosmos/cosmos-sdk/codec"
	//banktypes "github.com/cosmos/cosmos-sdk/x/bank/types"
	crosschaintypes "github.com/zeta-chain/zetacore/x/crosschain/types"
	//crosschaintypes "github.com/zeta-chain/zeta-node/x/crosschain/types"
)

func queryState(height uint64) (string, error) {
	// Create a connection to the gRPC server.
	grpcConn, err := grpc.Dial(
		"127.0.0.1:9090",    // your gRPC server address.
		grpc.WithInsecure(), // The Cosmos SDK doesn't support any transport security mechanism.
		// This instantiates a general gRPC codec which handles proto bytes. We pass in a nil interface registry
		// if the request/response types contain interface instead of 'nil' you should pass the application specific codec.
		grpc.WithDefaultCallOptions(grpc.ForceCodec(codec.NewProtoCodec(nil).GRPCCodec())),
	)
	if err != nil {
		return "", err
	}
	defer grpcConn.Close()

	// This creates a gRPC client to query the service.
	crosschainClient := crosschaintypes.NewQueryClient(grpcConn)
	res, err := crosschainClient.ZEVMGetBlock(
		context.Background(),
		//&crosschaintypes.QueryZEVMGetBlockRequest{Height: height},
		&crosschaintypes.QueryZEVMGetBlockByNumberRequest{Height: height},
	)
	if err != nil {
		return "", err
```

```
    }

    return fmt.Sprintf("Block Number: %s\nBlock Hash: %s\n", res.Number, res.Hash), nil
}

func main() {
    var latestBlock uint64 = 0
    fmt.Printf("Querying the max int64 value (%s).\nThis will cause an error\n", uint(math.MaxInt64))
    res, err := queryState(uint64(math.MaxInt64))
    if err != nil {
        // We expect to get here
        fmt.Printf("Error occurred in call 1: %s\n", err)

        // Extract the latest block number. It will be the last value in the error string
        words := strings.Split(err.Error(), " ")
        latestBlock, err = strconv.ParseUint(words[len(words)-1], 10, 64)
        if err != nil {
            panic("Could not parse block from error string")
        }
    }
    fmt.Println(res)

    fmt.Printf("Querying the max int64 value + 1 (%s). This will overflow and get the latest block\n", uint64(math.MaxInt64+1))
    res, err = queryState(uint64(math.MaxInt64) + 1)
    if err != nil {
        fmt.Printf("Error occurred in call 2: %s\n", err)
    }
    fmt.Println(res)

    if latestBlock == 0 {
        panic("latestBlock is (still) 0. This should not happen.")
    }

    fmt.Printf("Querying the latest block as given in first request(%s).\nThis will match the results from the previous request. This demonstrates that when the code overflows, it returns the latest block\n", latestBlock)
    res, err = queryState(latestBlock)
    if err != nil {
```

```go
            fmt.Printf("Error occurred in call 3: %s\n", err)
        }
        fmt.Printf(res)
}
```

Listing 21: Output demonstrates overflow

```
Querying the max int64 value (%!s(uint=9223372036854775807)).
This will cause an error
Error occurred in call 1: rpc error: code = Internal desc = failed to get block by height 9223372036854775807: height 9223372036854775807 must be less than or equal to the current blockchain height 13698

Querying the max int64 value + 1 (%!s(uint64=9223372036854775808)). This will overflow and get the latest block
Block Number: 0x8000000000000000
Block Hash: 0x141e0e146ea4ecc707918883cd8b9fd1fe775dd10191fc08586787baea441c97

Querying the latest block as given in first request(%!s(uint64=13698)).
This will match the results from the previous request. This demonstrates that when the code overflows, it returns the latest block
Block Number: 0x3582
Block Hash: 0x141e0e146ea4ecc707918883cd8b9fd1fe775dd10191fc08586787baea441c97
```

Risk Level:

Likelihood - 2
Impact - 4

Recommendation:

Use consistent types in the codebase. For block heights native to the Cosmos portion of the codebase, uint64 should be used, as this conforms to the protobuf definition for block height. uint64 can support a higher number of blocks. As block height should never be negative, there is no reason to use a signed data type in this case.

3.13 (HAL-13) RELIANCE ON UNISWAPV2 POOLS FOR PRICES EXPOSES ZETACHAIN TO PRICE MANIPULATION RISK - MEDIUM

Description:

ZetaChain makes extensive use of UniswapV2 pools in order to exchange ERC20 representation of assets, including its native token Zeta. This causes ZetaChain to inherit the security risks inherent in UniswapV2 pools, including risks such as front-running, price manipulation, de-pegging risks, and so on.

Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

Using UniswapV3 pools could represent a security improvement for the protocol. The newer version of Uniswap contains additional security features and its oracles are calculated in a way that is appropriate to Ethereum's Proof-of-Stakes consensus (whereas V2 was modeled on Proof-of-Work).

3.14 (HAL-14) ARBITRARY MINTING OF ZETA VIA MintZetaToEVMAccount FUNCTION - MEDIUM

Description:

It is possible for the function MintZetaToEVMAccount to return an error in some cases. When this error occurs, the protocol will return a valid new state rather than revert. However, the coins that were just minted are not burned. This could create a scenario where extra Zeta tokens are minted even when errors occur. As the total supply of Zeta is intended to be capped at a certain number, this could undermine the integrity of the protocol.

Code Location:

Risk Level:

Likelihood - 4
Impact - 2

BVSS - AO:A/AC:M/AX:H/C:N/I:C/A:C/D:C/Y:C/R:N/S:C - 5.0 - Medium

Recommendation:

When an error occurs in this function, consider burning the tokens that were just minted in order to avoid changing the total supply.

3.15 (HAL-15) SUPPORT FOR A TOKEN CANNOT BE REMOVED FROM THE PROTOCOL - MEDIUM

Description:

A RemoveForeignCoin function is commented out in the codebase. This function appears to allow an admin to remove a coin from the protocol, but it is not operative in its current state.

Even if this function was restored, the protocol is still at risk because there is no mechanism for another stakeholder in the network to initiate a vote or action to remove bad coins. It would be possible for a malicious or compromised admin account to e.g. add a fake version of a stablecoin to the network. Users may lose money by exchanging other tokens for this malicious coin.

Alternatively, a coin may lose all of its value in the case of e.g. a stablecoin depeg, an exit scam, and so on.

There are mechanisms an admin can invoke to stop all transactions, as well as to remove support for a chain. However, this mechanism does not have the granularity needed to remove support for a specific token. For example, it may be possible to remove support for the Ethereum main net, but it would not be possible to remove support for e.g. USDT if it loses its peg. This puts the system at risk.

Code Location:

x/fungible/keeper/msg_server_remove_foreign_coin.go, Line 8.

Listing 22: RemoveForeignCoin is commented out.

```
8 func (k msgServer) RemoveForeignCoin(goCtx context.Context, msg *types.MsgRemoveForeignCoin) (*types.MsgRemoveForeignCoinResponse, error) {
9     //ctx := sdk.UnwrapSDKContext(goCtx)
```

```
    //
    //if msg.Creator != types.AdminAddress {
    //  return nil, sdkerrors.Wrap(sdkerrors.ErrUnauthorized, "only admin can remove foreign coin")
    //}
    //index := msg.Name

    //_, found := k.GetForeignCoins(ctx, index)
    //if !found {
    //  return nil, sdkerrors.Wrapf(sdkerrors.ErrInvalidRequest, "foreign coin not found")
    //}
    //k.RemoveForeignCoins(ctx, index)
    // TODO : FIX THIS MSG
    return &types.MsgRemoveForeignCoinResponse{}, nil
}
```

x/fungible/keeper/foreign_coins.go, Line 41

Listing 23: RemoveForeignCoins is commented out.

```
41 // RemoveForeignCoins removes a foreignCoins from the store
42 //func (k Keeper) RemoveForeignCoins(
43 //  ctx sdk.Context,
44 //  index string,
45 //
46 //) {
47 //  store := prefix.NewStore(ctx.KVStore(k.storeKey), types.KeyPrefix(types.ForeignCoinsKeyPrefix))
48 //  store.Delete(types.ForeignCoinsKey(
49 //      index,
50 //  ))
51 //}
```


Risk Level:

Likelihood - 3
Impact - 3

Recommendation:

It is advised to implement a mechanism to remove tokens that could undermine the health of the network. This should be subject to governance.

3.16 (HAL-16) USE OF VULNERABLE COSMOSSDK VERSION - LOW

Description:

Versions of Cosmos SDK < 1.46.10 are susceptible to denial-of-service attacks. For more information, see the release notes.

Code Location:

go.mod

Listing 24: Vulnerable version of CosmosSDK.

```
module github.com/zeta-chain/zetacore

go 1.19

require (
    github.com/cosmos/cosmos-sdk v0.46.8
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Update to a version of Cosmos SDK that does not have known vulnerabilities. In general, we recommend monitoring the Cosmos SDK repository and following the security instructions provided by the developers.

3.17 (HAL-17) ValidateBasic INCOMPLETE FOR SOME MESSAGE TYPES - LOW

Description:

While the ValidateBasic functionality is implemented for messages in ZetaChain, often the validation is incomplete and may lead to issues as a result.

Code Location:

An example of incomplete validation can be found in file x/crosschain/types/messages_tss_voter.go. The fields Address and Pubkey could be verified.

The message is defined as follows x/crosschain/types/tx.pb.go

Listing 25

```go
type MsgCreateTSSVoter struct {
    Creator string `protobuf:"bytes,1,opt,name=creator,proto3" json:"creator,omitempty"`
    Chain   string `protobuf:"bytes,3,opt,name=chain,proto3" json:"chain,omitempty"`
    Address string `protobuf:"bytes,4,opt,name=address,proto3" json:"address,omitempty"`
    Pubkey  string `protobuf:"bytes,5,opt,name=pubkey,proto3" json:"pubkey,omitempty"`
}
```

Listing 26: Incomplete validation: Chain, Address, and Pubkey are not validated

```go
func (msg *MsgCreateTSSVoter) ValidateBasic() error {
    _, err := sdk.AccAddressFromBech32(msg.Creator)
    if err != nil {
```

```
        return sdkerrors.Wrapf(sdkerrors.ErrInvalidAddress, "invalid creator address (%s)", err)
    }
    return nil
}
```

This is just one example. Overall the Messages in the system should be reviewed for incomplete validation.

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

It is important to take advantage of the ValidateBasic functionality in Cosmos in order to ensure that only valid messages are accepted and processed. This can have performance and security benefits. Even if the values are checked elsewhere (such as the client used to build the request or deeper in the Cosmos system), it is good to add as much validation as possible as part of a defense-in-depth approach.

In the case of ZetaChain, it is recommended to check for negative integers for values of block height, chain id, and so on. Many of these types are defined as or converted to signed data types, but should not have negative values. Rejecting them in ValidateBasic could help to avoid overflow issues similar to the ones outlined elsewhere in this report.

3.18 (HAL-18) CENTRALIZATION RISK - LOW

Description:

An admin user can control critical aspects of the projects, such as:
- whether deposits can be made (i.e. whether inbound transactions are enabled) via disabling the tracking of events from external networks that indicate transactions into Zeta
- whether a new token can be added to the network

In addition, several aspects of the protocol are hard-coded and cannot be modified by validators or governance, such as gas limits for transactions.

Code Location:

x/observer/types/params.go, Lines 21-43.

Listing 27: Admin policies configured to be controlled by a single address

```
func DefaultParams() Params {
    chains := common.DefaultChainsList()
    observerParams := make([]*ObserverParams, len(chains))
    for i, chain := range chains {
        observerParams[i] = &ObserverParams{
            IsSupported:           true,
            Chain:                 chain,
            BallotThreshold:       sdk.MustNewDecFromStr("0.66"),
            MinObserverDelegation: sdk.MustNewDecFromStr("10000000000"),
        }
    }
    adminPolicy := []*Admin_Policy{
        {
            PolicyType: Policy_Type_stop_inbound_cctx,
            Address:    "zeta1afk9zr2hn2jsac63h4hm60vl9z3e5u69gndzf7c99cqge3vzwjzsxn0x73",
        },
        {
```

```
                PolicyType: Policy_Type_deploy_fungible_coin,
                Address:    "
↳ zeta1afk9zr2hn2jsac63h4hm60vl9z3e5u69gndzf7c99cqge3vzwjzsxn0x73",
            },
        }
        return NewParams(observerParams, adminPolicy)
}
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Allowing a single account to control critical aspects of the protocol poses a decentralization risk. If a private key controlling the admin account is compromised, an attacker could manipulate the network. It also poses a risk to users if an admin can make a sudden change to the protocol that impacts user's funds.

It is recommended to decentralize key features of the protocol so that stakeholders can decide on the project's operation via governance. This can also mitigate the effects of an admin's private key being compromised.

3.19 (HAL-19) LACK OF UNIT TESTS - LOW

Description:

The project makes some use of unit tests, but coverage is not very thorough. Where unit tests exist, they primarily verify only the happy path and error conditions are not simulated.

Code Location:

Two examples are listed below to indicate missing unit tests. These serve to demonstrate the types of test coverage that should be in place, but they are not an exhaustive list of all missing or flawed tests.

Example 1: Changes to bank balances after minting new coins to an address

x/fungible/keeper/zeta.go

Listing 28: Run-time checking is used,but there is no unit test checking the general behavior (as confirmed by TODO note)

```
// Mint ZETA (gas token) to the given address
// TODO balanceCoinAfter != expCoin , be replicated in an unit test
func (k *Keeper) MintZetaToEVMAccount(ctx sdk.Context, to sdk.AccAddress, amount *big.Int) error {
    balanceCoin := k.bankKeeper.GetBalance(ctx, to, config.BaseDenom)
    coins := sdk.NewCoins(sdk.NewCoin(config.BaseDenom, sdk.NewIntFromBigInt(amount)))
    // Mint coins
    if err := k.bankKeeper.MintCoins(ctx, types.ModuleName, coins); err != nil {
        return err
    }

    // Send minted coins to the receiver
    if err := k.bankKeeper.SendCoinsFromModuleToAccount(ctx, types
```

```go
.ModuleName, to, coins); err != nil {
        return err
    }

    // Check expected receiver balance after transfer
    balanceCoinAfter := k.bankKeeper.GetBalance(ctx, to, config.BaseDenom)
    expCoin := balanceCoin.Add(coins[0])

    if ok := balanceCoinAfter.IsEqual(expCoin); !ok {
        return sdkerrors.Wrapf(
            types.ErrBalanceInvariance,
            "invalid coin balance - expected: %v, actual: %v",
            expCoin, balanceCoinAfter,
        )
    }

    return nil
}
```

Example 2: Error-checking is absent in GetBallot unit test

Although there are four methods in x/observer/keeper/ballot.go, only one method, GetBallot has a unit test in the file ballot_test.go

Listing 29: Existing unit test for GetBallot

```go
func TestKeeper_GetBallot(t *testing.T) {
    k, ctx := SetupKeeper(t)
    identifier := "0x9ea007f0f60e32d58577a8cf25678942d2b10791c2a34f48e237b76a7e998e4d"
    k.SetBallot(ctx, &types.Ballot{
        Index:            "",
        BallotIdentifier: identifier,
        VoterList:        nil,
        ObservationType:  0,
        BallotThreshold:  sdk.Dec{},
        BallotStatus:     0,
    })

    k.GetBallot(ctx, identifier)
}
```

This test does not check the return values of GetBallot, which include both a Ballot struct and a boolean indicating whether the Ballot was found. Since these values are not checked, this test can only help to catch a panic but does not assist in determining whether GetBallot correctly returns expected values for the Ballot or whether the Ballot exists.

Risk Level:

Likelihood - 3
Impact - 1

Recommendation:

Bridges are an extremely valuable target for attackers. They are also highly complex. For these reasons, extensive unit tests should be considered a requirement for a protocol that contains bridging functionality.

In the examples listed above, unit tests checking account balances should be added to the module in addition to run-time error checking. Additionally, methods should be covered by unit tests and these tests should deliberately include failure cases as well as malicious and malformed input to simulate edge-cases and attacks.

When a function has a return value, it should always be checked to ensure that it correctly succeeds and fails in the appropriate contexts.

3.20 (HAL-20) LACK OF FUZZ TESTS - LOW

Description:

Fuzz testing is a testing technique to simulate a wide range of potential system states as well as data inputs in order to determine whether critical properties of the system remain true. Properly used, fuzz testing can provide excellent support to unit tests and smoke tests.

While there is some code referring to Cosmos simulations (the CosmosSDK's built-in fuzzing tool), it appears to have been generated by a build tool and the code does not contain actual fuzz tests.

Risk Level:

Likelihood - 3
Impact - 1

Recommendation:

For a project providing bridging functionality, fuzz testing should be considered a requirement. We recommend the Zeta team to agree on a set of invariants for the system and then incorporate them into Cosmos simulations. Example invariants for ZetaChain might include:

- The total supply of Zeta should not increase.
- No Zeta should be minted if the system reaches an error state (or else any Zeta minted should also be burned).
- The sum of all balances of Zeta should not exceed the total supply.

3.21 (HAL-21) BITCOIN TRANSACTIONS WITH LARGE NONCE VALUES MAY BE SENT REPEATEDLY - LOW

Description:

To track Bitcoin transactions, ZetaChain calculates an identifier for each transaction and evaluates whether it has been processed.

The identifier is created by merging several values together, including a Nonce value tracked by Zeta.

There is a type conversion issue in these calculations, where the Nonce (which is of type uint64) is converted to an int64. This overflows for values greater than MAX_INT64. When this occurs, the transaction ID will be calculated incorrectly. As a result, a transaction will not be marked as processed.

This could have severe consequences for the protocol. If a transaction is not marked processed, the function TryProcessOutTx could have the effect of processing a single Bitcoin transaction multiple times. This could result in a loss of funds from the ZetaChain Bitcoin account and potentially cease Bitcoin transactions from functioning if the wallet becomes totally depleted.

While the impact here is very high, it would require the Nonce value stored in ZetaChain to become extremely large (> 9223372036854775807) through legitimate use or through a separate exploit. As a result, the overall likelihood of this occurring is quite low.

Code Location:

In zetaclient/bitcoin_client.go, the ob.submittedTx array is used to track outbound transactions. It is populated using a outTxID calculated using tracker.Nonce which has the type Uint64 (defined in x/crosschain/types/out_tx_tracker.pb.go)

The zetaclient observes logs of transactions and performs actions based on their status. This code is an excerpt of a loop that examines incoming transactions. Here the current transaction is called send and is passed to the function TryProcessOutTx. (The values for outTxID can be ignored here, as it is used to manage a separate mutex system that is not relevant for this finding).

zetaclient/zetacore_observer.go

Listing 30: The current transaction 'send' is passed to TryProcessOutTx

```
chain := GetTargetChain(send)
nonce := send.GetCurrentOutTxParam().OutboundTxTssNonce
outTxID := fmt.Sprintf("%s-%d-%d", send.Index, send.GetCurrentOutTxParam().ReceiverChainId, nonce) // should be the outTxID?

// FIXME: config this schedule; this value is for localnet fast testing
if nonce%1 == bn%1 && !outTxMan.IsOutTxActive(outTxID) {
    outTxMan.StartTryProcess(outTxID)
    fmt.Printf("chain %s: Sign outtx %s with value %d\n", chain, send.Index, send.GetCurrentOutTxParam().Amount)
    go signer.TryProcessOutTx(send, outTxMan, outTxID, chainClient, co.bridge)
}
```

The Nonce value is then extracted from send and converted to an int64 before it is passed to IsSendOutTxProcessed. When this function returns true, the code will return early. Otherwise, it will post a new Bitcoin transaction.

Listing 31: Nonce is converted to an int64 from uint64. An overflow occurs here for larger values of Nonce

```
func (signer *BTCSigner) TryProcessOutTx(send *types.CrossChainTx, outTxMan *OutTxProcessorManager, outTxID string, chainclient ChainClient, zetaBridge *ZetaCoreBridge) {
    toAddr, err := hex.DecodeString(send.GetCurrentOutTxParam().Receiver[2:])
    if err != nil {
        signer.logger.Error().Msgf("BTC TryProcessOutTx: %s,
```

```
↳ decode to address err %v", send.Index, err)
            return
        }
        fmt.Printf("BTC TryProcessOutTx: %s, value %d to %s\n", send.
↳ Index, send.GetCurrentOutTxParam().Amount.BigInt(), toAddr)
        defer func() {
            outTxMan.EndTryProcess(outTxID)
        }()
        btcClient, ok := chainclient.(*BitcoinChainClient)
        if !ok {
            signer.logger.Error().Msgf("chain client is not a bitcoin
↳ client")
            return
        }

        logger := signer.logger.With().
            Str("sendHash", send.Index).
            Logger()

        myid := zetaBridge.keys.GetAddress().String()

        // Early return if the send is already processed
        // FIXME: handle revert case
        included, confirmed, _ := btcClient.IsSendOutTxProcessed(send.
↳ Index, int(send.GetCurrentOutTxParam().OutboundTxTssNonce), common
↳ .CoinType_Gas)
        if included || confirmed {
            logger.Info().Msgf("CCTX already processed; exit signer")
            return
        }
```

The int64 nonce is used to calculate outTxID and check whether it exists in the map ob.submittedTX.
btc_signer.go.

Listing 32: outTxID calculated using int64 nonce

```
func (ob *BitcoinChainClient) IsSendOutTxProcessed(sendHash string
↳ , nonce int, _ common.CoinType) (bool, bool, error) {
        chain := ob.chain.ChainId
        outTxID := fmt.Sprintf("%d-%d", chain, nonce)
        ob.logger.Info().Msgf("IsSendOutTxProcessed %s", outTxID)

```

```
    res, found := ob.submittedTx[outTxID]
    if !found {
        return false, false, nil
    }
```

The calculation of the map ob.submittedTx uses tracker.Nonce to calculate obTxID. The tracker is a OutTxTracker struct, which uses an Uint64 value for Nonce.

Listing 33

```
func (ob *BitcoinChainClient) observeOutTx() {
    ticker := time.NewTicker(2 * time.Second)
    for {
        select {
        case <-ticker.C:
            trackers, err := ob.zetaClient.
 ↳ GetAllOutTxTrackerByChain(ob.chain)
            if err != nil {
                ob.logger.Error().Err(err).Msg("error
 ↳ GetAllOutTxTrackerByChain")
                continue
            }
            for _, tracker := range trackers {
                outTxID := fmt.Sprintf("%d-%d", tracker.ChainId,
 ↳ tracker.Nonce)
                ob.logger.Info().Msgf("tracker outTxID: %s",
 ↳ outTxID)
                for _, txHash := range tracker.HashList {
                    hash, err := chainhash.NewHashFromStr(txHash.
 ↳ TxHash)
                    if err != nil {
                        ob.logger.Error().Err(err).Msg("error
 ↳ NewHashFromStr")
                        continue
                    }
                    getTxResult, err := ob.rpcClient.
 ↳ GetTransaction(hash)
                    if err != nil {
                        ob.logger.Warn().Err(err).Msg("error
 ↳ GetTransaction")
                        continue
                    }
```

```
                    if getTxResult.Confirmations >= 0 {
                        ob.submittedTx[outTxID] = *getTxResult
                    }
                }
            }
        case <-ob.stop:
            ob.logger.Info().Msg("observeOutTx stopped")
            return
        }
    }
}
```

Taken together, the function IsSendOutTxProcessed will always fail for large nonces. When TryProcessOutTx overflows, the outTxID calculated using the now-overflowed Nonce value will not match the outTxID that is correctly calculated using uint64 nonce values.

Risk Level:

Likelihood - 3
Impact - 1

Recommendation:

When converting types, ensure that a consistent value is used for each concept (such as gas price) in the system. If the value begins as uint64 it should also end up as a uint64. If it is necessary to use an alternate representation of the value, such as big.Int or string when passing a message from one system to another, ensure that no downcasting occurs. This means that a data type that holds large values should not be converted to one that cannot contain the entire range of values of the former. Extra caution should be used when converting between signed and unsigned types, especially when these values are used to calculate funds.

3.22 (HAL-22) UNBOUNDED ARRAY IN Signers IN ChainNonces COULD CAUSE RESOURCE EXHAUSTION - LOW

Description:

The array Signers in the ChainNonces in the file x/crosschain/keeper/keeper_chain_nonces.go will grow each time the same signer sends a MsgNonceVoter message. An attacker could abuse this to consume system resources and potentially degrade the performance of the network.

An attacker can create a loop to issue the MsgNonceVoter transaction repeatedly so that the array grows to a very large size. Storing this array and performing calculations with it could then cause major problems as it is never cleared.

There is existing code in this function to omit duplicates from this array, but it is commented-out.

It is important to note that this function can only be called by bonded validators.

Code Location:

Listing 34: msg.Creator is appended to chanNonce.Signers for ChainNonces that already exist.

```
func (k msgServer) NonceVoter(goCtx context.Context, msg *types.MsgNonceVoter) (*types.MsgNonceVoterResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)

    validators := k.StakingKeeper.GetAllValidators(ctx)
    if !IsBondedValidator(msg.Creator, validators) {
        return nil, sdkerrors.Wrap(sdkerrors.ErrorInvalidSigner, fmt.Sprintf("signer %s is not a bonded validator", msg.Creator))
    }

```

```go
    chain := msg.Chain
    chainNonce, isFound := k.GetChainNonces(ctx, chain)
    //if isDuplicateSigner(msg.Creator, chainNonce.Signers) {
    //  return nil, sdkerrors.Wrap(sdkerrors.ErrorInvalidSigner, fmt.Sprintf("signer %s double signing!!", msg.Creator))
    //}
    if isFound {
        chainNonce.Signers = append(chainNonce.Signers, msg.Creator)
        chainNonce.Nonce = msg.Nonce
    } else if !isFound {
        chainNonce = types.ChainNonces{
            Creator: msg.Creator,
            Index:   msg.Chain,
            Chain:   msg.Chain,
            Nonce:   msg.Nonce,
            Signers: []string{msg.Creator},
        }
    } else {
        return nil, sdkerrors.Wrap(sdkerrors.ErrInvalidRequest, fmt.Sprintf("chainNonce vote msg does not match state: %v vs %v", msg, chainNonce))
    }

    //if hasSuperMajorityValidators(len(chainNonce.Signers), validators) {
    //  chainNonce.FinalizedHeight = uint64(ctx.BlockHeader().Height)
    //}

    k.SetChainNonces(ctx, chainNonce)
    return &types.MsgNonceVoterResponse{}, nil
}
```

```
> zetacored q crosschain list-chain-nonces
ChainNonces:
- chain: Goerli
  creator: zeta1syavy2npfyt9tcncdtsdzf7kny9lh777heefxk
  finalizedHeight: "0"
  index: Goerli
  nonce: "101"
  signers:
  - zeta1syavy2npfyt9tcncdtsdzf7kny9lh777heefxk
  - zeta1syavy2npfyt9tcncdtsdzf7kny9lh777heefxk
  - zeta1syavy2npfyt9tcncdtsdzf7kny9lh777heefxk
pagination:
  next_key: null
  total: "0"
```

Figure 1: Example of duplicated signers

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Do not add duplicate values for Signers. In general, avoid storing and unbounded sequences of data. Often, an attacker can abuse unbounded data storage in order to exhaust system resources.

3.23 (HAL-23) USE OF VULNERABLE DEPENDENCIES - LOW

Description:

In addition to the specific issues with the CosmosSDK and Go versions used by the project, a variety of other vulnerabilities exist in dependencies used by the project.

Code Location:

Vulnerabilities flagged by the tool nancy:

ID	Package	Rating	Description
CVE-2022-44797	btcd	CRITICAL	Improper Restriction of Operations
sonatype-2022-39389	btcd	MEDIUM	Improper Input Validation
CVE-2021-0076	go-ethereum	HIGH	Uncontrolled Resource Consumption
CVE-2022-23328	go-ethereum	HIGH	Uncontrolled Resource Consumption
CVE-2022-37450	go-ethereum	MEDIUM	Improper Input Validation

Excerpt from the tool govulncheck:

Listing 35: Sample out from govulncheck

```
govulncheck is an experimental tool. Share feedback at https://go.
↳ dev/s/govulncheck-feedback.

Scanning for dependencies with known vulnerabilities...
Found 18 known vulnerabilities.
...
```


Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Where possible, keep dependencies patched in order to reduce the risk of the system being attacked using known vulnerabilities. A tool like govulncheck can be added to ZetaChain's CI pipeline. This can then be configured to show serious issues that could affect the project.

It is important to note that many of these vulnerabilities flagged by govulncheck are unlikely to be exploitable in practice, as they larger refer to a Web2 context.

It is recommended that the Zeta chain run the nancy and govulncheck tools regularly and fix as many warnings as possible.

3.24 (HAL-24) MNEMONIC PHRASES PRESENT IN CODEBASE - LOW

Description:

Bash scripts in the repository contain hard-coded mnemonic phrases corresponding to Cosmos wallets. In general, we recommend that mnemonics are not committed to a repository, as it increases the risk that the wallets will be used in a production environment.

Code Location:

Listing 36: Mnemonic phrases present in repository

```
init.sh:30:"race draft rival universe maid cheese steel logic crowd fork comic easy truth drift tomorrow eye buddy head time cash swing swift midnight borrow"
init.sh:33:"hand inmate canvas head lunar naive increase recycle dog ecology inhale december wide bubble hockey dice worth gravity ketchup feed balance parent secret orchard"
standalone-network/upgrade-integration-download.sh:34:"race draft rival universe maid cheese steel logic crowd fork comic easy truth drift tomorrow eye buddy head time cash swing swift midnight borrow"
standalone-network/upgrade-integration-download.sh:35:"hand inmate canvas head lunar naive increase recycle dog ecology inhale december wide bubble hockey dice worth gravity ketchup feed balance parent secret orchard"
standalone-network/init.sh:13:"race draft rival universe maid cheese steel logic crowd fork comic easy truth drift tomorrow eye buddy head time cash swing swift midnight borrow"
standalone-network/init.sh:14:"hand inmate canvas head lunar naive increase recycle dog ecology inhale december wide bubble hockey dice worth gravity ketchup feed balance parent secret orchard"
standalone-network/init.sh:15:"lounge supply patch festival retire duck foster decline theme horror decline poverty behind clever harsh layer primary syrup depart fantasy session fossil dismiss east"
standalone-network/upgrade-integration.sh:63:"race draft rival universe maid cheese steel logic crowd fork comic easy truth drift
```

```
↳  tomorrow eye buddy head time cash swing swift midnight borrow"
standalone-network/upgrade-integration.sh:65:"hand inmate canvas
↳ head lunar naive increase recycle dog ecology inhale december wide
↳  bubble hockey dice worth gravity ketchup feed balance parent
↳ secret orchard"
standalone-network/upgrade-integration-client.sh:58:"race draft
↳ rival universe maid cheese steel logic crowd fork comic easy truth
↳  drift tomorrow eye buddy head time cash swing swift midnight
↳ borrow"
standalone-network/upgrade-integration-client.sh:61:"hand inmate
↳ canvas head lunar naive increase recycle dog ecology inhale
↳ december wide bubble hockey dice worth gravity ketchup feed
↳ balance parent secret orchard"
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Avoid committing mnemonic phrases to the repository. Use temporary credentials instead.

3.25 (HAL-25) VULNERABLE TSS-LIB CONTAINS HASH COLLISION ISSUES - LOW

Description:

During the audit, an issue in Binance Smart Chain's tss-lib software was discovered. Older versions of this software are susceptible to hash collisions. The security properties of TSS rely on the use of collision-resistant hash functions. If a hash function used in a higher-level cryptographic construction does not have this property, it may be possible for an attacker to forge a signature. This undermines the security of the TSS mechanism within tss-lib and in ZetaChain as a result, as this library is used to create addresses that control funds in the protocol.

Halborn has confirmed that tss-lib version 1.3.2 used by ZetaChain is vulnerable.

Code Location:

Listing 37: go.mod shows tss-lib

```
require (
    cosmossdk.io/math v1.0.0-beta.4
    github.com/99designs/keyring v1.2.1
    github.com/binance-chain/tss-lib v1.3.2
    github.com/btcsuite/btcd v0.22.2
```

Risk Level:

Likelihood - 4
Impact - 1

Recommendation:

Update to the latest version of tss-lib in order to avoid any cryptographic issues that could undermine ZetaChain’s use of the TSS library.

More details can be found at https://github.com/bnb-chain/tss-lib/pull/233.

3.26 (HAL-26) CALCULATION ERRORS IN getSatoshis FUNCTION FOR EXTREMELY SMALL OR LARGE FLOAT VALUES - LOW

Description:

The function getSatoshis in btc_util.go converts a float64 amount of Bitcoins to an int64 satoshis representation of the same value. At the extremes of the float64 data type, this calculation breaks down and returns erroneous results. This was discovered using property-based fuzz testing.

For very large float values of Bitcoin, the calculation will overflow and return very small values of Satoshis.

Very small float values of Bitcoin will be truncated when converted to Satoshis and in this case, represented value may be destroyed.

A copy of the fuzzing harness used has been shared with ZetaChain.

Code Location:

Listing 38

```
func getSatoshis(btc float64) (int64, error) {
    // The amount is only considered invalid if it cannot be represented
    // as an integer type.  This may happen if f is NaN or +-Infinity.
    switch {
    case math.IsNaN(btc):
        fallthrough
    case math.IsInf(btc, 1):
        fallthrough
    case math.IsInf(btc, -1):
        return 0, errors.New("invalid bitcoin amount")
    }
    return round(btc * satoshiPerBitcoin), nil
```

```
}
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

Modify the getSatoshis function to check for edge-cases on the float64 parameter.

In practice, these scenarios are unlikely to occur: the maximum float64 value far exceeds the total supply of Bitcoins and the smallest float value is far smaller than a single satoshi.

However, it is recommended that the `getSatoshis` function handles such cases explicitly and rejects extremely large or tiny values of bitcoin as input to the function. For example, the total supply of Bitcoins will never exceed 21 million. For this reason, a float64 with an absolute value outside this range should be rejected. Similarly, the smallest unit of Bitcoin is one satoshi, so this function should not return a value with a magnitude larger than `21_000_000 * 1e8`, as this is the maximum number of Satoshis.

When such properties are encoded into this function, fuzz testing can be introduced to ensure that these properties are never violated.

3.27 (HAL-27) GAS LIMITS CANNOT BE CONFIGURED - LOW

Description:

When issuing a transaction across chains, a gas fee is calculated by multiplying a variable gasPrice (determined by the validators as detailed elsewhere in this report) with a value GAS_LIMIT.

This latter variable is stored in a Solidity smart contract representing the token, and its value is set in the contract's constructor. The GAS_LIMIT is chosen when the contract is deployed. These values are hard-coded to 100 for Bitcoin and 21_000 for all other networks. While there is a function in the Solidity code to update this value, there is no reference to it on the Cosmos side. Furthermore, the Solidity function has an access control mechanism that is configured to allow access from only the fungible module in the Cosmos code. As there is no corresponding function to change the GAS_LIMIT on the Cosmos side, it is effectively impossible to do so.

While these values are sufficient for a typical transaction to in their respective networks, they may be too low in practice. When creating blocks, transactions with higher gas fees are preferred by block creators. The values configured by Zeta are the minimal amount needed for a transaction to be processed. For this reason, Zeta transactions will be considered a lower priority relative to other transactions in the networks. This will have the effect of causing Zeta transactions to be slow or even stuck, especially when the Bitcoin or EVM network in question is congested.

Code Location:

x/fungible/keeper/gas_coin_and_pool.go, L19

Listing 39

```go
func (k Keeper) setupChainGasCoinAndPool(ctx sdk.Context, c string, gasAssetName string, symbol string, decimals uint8) (ethcommon.Address, error) {
    name := fmt.Sprintf("%s-%s", gasAssetName, c)
    chainName := common.ParseChainName(c)
    chain := k.zetaobserverKeeper.GetParams(ctx).GetChainFromChainName(chainName)
    if chain == nil {
        return ethcommon.Address{}, zetaObserverTypes.ErrSupportedChains
    }

    transferGasLimit := big.NewInt(21_000)
    if chain.IsEVMChain() {
        transferGasLimit = big.NewInt(21_000)
    } else if chain.IsBitcoinChain() {
        transferGasLimit = big.NewInt(100) // 100B for a typical tx
    }
```


Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

It is recommended to create a mechanism to change the gas limit so that Zeta transactions will be selected by block creators during periods where the gas auctions are more competitive.

3.28 (HAL-28) VALIDATORS CANNOT EFFECTIVELY SET THE ZETA GAS PRICE FOR BITCOIN TRANSACTIONS - LOW

Description:

For all chains, validators can set gas prices via the median index method. In this way, the gas price is set to a rough approximation of the total range of gas prices in the system. However, the bitcoin client automatically polls the bitcoin price using the bitcoin RPC call `estimatesmartfee`.

As a result, validators have limited control over the Bitcoin fee used in transactions on ZetaChain. While the gas price voting mechanism should allow validators to agree on a price, this is frustrated in practice if the fee is automatically reset.

Note that while it is possible for the validators to race the price polling if they were to coordinate setting a new median price before the 5-second window, this is impractical due to the uncertainty whether the transaction will be processed with the new price within 5 seconds.

This can be demonstrated by posting a gas price and then querying the updated gas prices.

Listing 40: Submitting a new gas price for the Bitcoin test network

```
export PRICE="123456789"
zetacored tx crosschain gas-price-voter 18444 $PRICE 100 100 --keyring-backend=test --yes --chain-id=athens_101-1 --broadcast-mode=block --gas=auto --gas-adjustment=2 --gas-prices=0.1azeta --from=val
# Query the new price
zetacored q crosschain list-gas-price
```

Listing 41: Response for list-gas-price query. The updated price appears

```
GasPrice:
...
- block_nums:
  - "196"
  - "100"
  chain_id: "18444"
  creator: zeta1lz2fqwzjnk6qy48fgj753h48444fxtt7hekp52
  index: "18444"
  median_index: "1"
  prices:
  - "1000"
  - "123456789"
  signers:
  - zeta1lz2fqwzjnk6qy48fgj753h48444fxtt7hekp52
  - zeta1z46tdw75jvh4h39y3vu758ctv34rw5z9kmyhgz
pagination:
  next_key: null
  total: "0"
```

If the price is queried again just a few seconds later:

Listing 42: Response for second list-gas-price query. The price has reset.

```
GasPrice:
...
- block_nums:
  - "185"
  - "185"
  chain_id: "18444"
  creator: zeta1lz2fqwzjnk6qy48fgj753h48444fxtt7hekp52
  index: "18444"
  median_index: "1"
  prices:
  - "1000"
  - "1000"
  signers:
  - zeta1lz2fqwzjnk6qy48fgj753h48444fxtt7hekp52
  - zeta1z46tdw75jvh4h39y3vu758ctv34rw5z9kmyhgz
pagination:
```

```
next_key: null
total: "0"
```

The price has been overwritten automatically.

Code Location:

zetaclient/bitcoin_client.go Lines 305-320.

Listing 43: WatchGasPrice() posts a new gas price at a fixed interval

```
305 func (ob *BitcoinChainClient) WatchGasPrice() {
306     gasTicker := time.NewTicker(time.Duration(config.BitcoinConfig.WatchGasPricePeriod) * time.Second)
307     for {
308         select {
309         case <-gasTicker.C:
310             err := ob.PostGasPrice()
311             if err != nil {
312                 ob.logger.Error().Err(err).Msg("PostGasPrice error on " + ob.chain.String())
313                 continue
314             }
315         case <-ob.stop:
316             ob.logger.Info().Msg("WatchGasPrice stopped")
317             return
318         }
319     }
320 }
```

zetaclient/bitcoin_client.go Lines 322-356

Listing 44: The PostGasPrice function determines the price by querying the Bitcoin RPC.

```
322 func (ob *BitcoinChainClient) PostGasPrice() error {
323     if ob.chain.ChainId == 18444 { //bitcoin regtest
324         bn, err := ob.rpcClient.GetBlockCount()
325         if err != nil {
326             return err
327         }
```

```go
        _, err = ob.zetaClient.PostGasPrice(ob.chain, 1000, "100", uint64(bn))
        if err != nil {
            ob.logger.Err(err).Msg("PostGasPrice:")
            return err
        }
        return nil
    }
    // EstimateSmartFee returns the fees per kilobyte (BTC/kb) targeting given block confirmation
    feeResult, err := ob.rpcClient.EstimateSmartFee(1, &btcjson.EstimateModeConservative)
    if err != nil {
        return err
    }
    if feeResult.Errors != nil || feeResult.FeeRate == nil {
        return fmt.Errorf("error getting gas price: %s", feeResult.Errors)
    }
    gasPrice := big.NewFloat(0)
    gasPriceU64, _ := gasPrice.Mul(big.NewFloat(*feeResult.FeeRate), big.NewFloat(1e8)).Uint64()
    bn, err := ob.rpcClient.GetBlockCount()
    if err != nil {
        return err
    }
    _, err = ob.zetaClient.PostGasPrice(ob.chain, gasPriceU64, "100", uint64(bn))
    if err != nil {
        ob.logger.Err(err).Msg("PostGasPrice:")
        return err
    }
    _ = feeResult
    return nil
}
```


Risk Level:

Likelihood - 2

Impact - 2

Recommendation:

We recommend using a time-weighted average price (TWAP) mechanism for calculating the gas prices in the chain. This would allow ZetaChain to reach a fair price for gas. Since there is no median index calculation, this removes the risk described in this finding.

If it is preferable for the Bitcoin-Zeta gas price to be calculated by the RPC call, consider removing the gas price voting feature for Bitcoin-like chains.

FINDINGS & TECH DETAILS

3.29 (HAL-29) GO VERSIONS PRIOR TO 1.20.2 CONTAIN CRYPTOGRAPHIC ISSUES AND OTHER BUGS - LOW

Description:

Go version 1.20.2 contains security and performance enhancements. Specifically, this release fixes problems in cryptographic libraries. Older versions of go are more susceptible to cryptography issues and side-channel attacks on cryptographic implementations.

Code Location:

go.mod

Listing 45

```
module github.com/zeta-chain/zetacore

go 1.19
```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

Update to Go v1.20.2 or newer when possible. More information can be found in the Go release notes.

3.30 (HAL-30) ZETACLIENT AND ZETACORE TRACK BLOCK HEIGHT USING DIFFERENT TYPES - LOW

Description:

Zetacore and zetaclient represent two different but tightly related parts of the overall ZetaChain. There is a potential logical issue that could arise during intercommunication between these two subsystems as they are tracking the same concept, block height, using different types.

The correct type for block height as defined by zetacore is uint64. However, zetaclient tracks block height as int64. As a result, there is an opportunity for overflow or data truncation when converting between these two types.

It is important to note that an overflow in this value is unlikely to occur for a long time, as it would represent a massive amount of time in the future. However, if another security issue arises where the block height is in some way interpreted as a very large number, this issue could complete disrupt the chain.

Code Location:

zetaclient/zetabridge.go

Listing 46: Definition of int64 block height in zetaclient

```
// ZetaCoreBridge will be used to send tx to ZetaCore.
type ZetaCoreBridge struct {
    logger              zerolog.Logger
    blockHeight         int64
    accountNumber       uint64
    seqNumber           uint64
    grpcConn            *grpc.ClientConn
    httpClient          *retryablehttp.Client
    cfg                 config.ClientConfiguration
```

```go
    keys               *Keys
    broadcastLock      *sync.RWMutex
    ChainNonces        map[string]uint64 // FIXME: Remove this?
    lastOutTxReportTime map[string]time.Time
}
```

One potential source of issues is in the Broadcast function.
zetaclient/broadcast.go

Listing 47

```go
// Broadcast Broadcasts tx to metachain. Returns txHash and error
func (b *ZetaCoreBridge) Broadcast(gaslimit uint64, msgs ...stypes.Msg) (string, error) {
    b.broadcastLock.Lock()
    defer b.broadcastLock.Unlock()
    var err error
    blockHeight, err := b.GetZetaBlockHeight()
    if err != nil {
        return "", err
    }

    if int64(blockHeight) > b.blockHeight {
        b.blockHeight = int64(blockHeight)
        accountNumber, seqNumber, err := b.GetAccountNumberAndSequenceNumber()
        if err != nil {
            return "", err
        }
        b.accountNumber = accountNumber
        if b.seqNumber < seqNumber {
            b.seqNumber = seqNumber
        }
    }
```

The function GetZetaBlockHeight above eventually returns the value from a struct in zetacore that uses an uint64 type for block height.

x/crosschain/types/query.pb.go

Listing 48

```
type QueryLastMetaHeightResponse struct {
    Height uint64 `protobuf:"varint,1,opt,name=Height,proto3" json
:"Height,omitempty"`
}
```

When the height gets to a value above MAX_INT64, the code in zetaclient/broadcast.go will overflow. This may have unintended results, as broadcasted transactions will likely be processed incorrectly given that the height will be interpreted as a negative integer.

Risk Level:

Likelihood - 4
Impact - 1

Recommendation:

When converting types, ensure that a consistent value is used for each concept (such as gas price) in the system. If the value begins as uint64 it should also end up as a uint64. If it is necessary to use an alternate representation of the value, such as big.Int or string when passing a message from one system to another, ensure that no downcasting occurs. This means that a data type that holds large values should not be converted to one that cannot contain the entire range of values of the former. Extra caution should be used when converting between signed and unsigned types, especially when these values are used to calculate funds.

3.31 (HAL-31) TYPE CONVERSION ISSUE FOR FIELD Decimals ON STRUCT MsgDeployFungibleCoinZRC20 - INFORMATIONAL

Description:

Decimals is uint32 but converted to uint8 in x/fungible/keeper/msg_server_deploy_fungible_coin_zrc_4.go L20. This will result in data truncation, which may be undesirable.

For example, if the value for Decimals is 255 (binary 1111 1111), there will be no problem as this number fits into 8 bits. However, if the value is 256 (binary: 1 0000 0000), the conversion to uint8 will remove the leading bit and result in a value of 0. This could be extremely problematic as it would likely make the deployed smart contract unusable. Any value exceeding MAX_UINT8 will give unexpected results between 0-255 depending on how the integer is represented on a binary level.

Code Location:

x/fungible/keeper/msg_server_deploy_fungible_coin_zrc_4.go

Listing 49: Data truncation occurs when converting to uint8

```
14 func (k msgServer) DeployFungibleCoinZRC20(goCtx context.Context, msg *types.MsgDeployFungibleCoinZRC20) (*types.MsgDeployFungibleCoinZRC20Response, error) {
15     ctx := sdk.UnwrapSDKContext(goCtx)
16     if msg.Creator != k.zetaobserverKeeper.GetParams(ctx).GetAdminPolicyAccount(zetaObserverTypes.Policy_Type_deploy_fungible_coin) {
17         return nil, sdkerrors.Wrap(sdkerrors.ErrUnauthorized, "Deploy can only be executed by the correct policy account")
18     }
19     if msg.CoinType == zetacommon.CoinType_Gas {
```

```
        _, err := k.setupChainGasCoinAndPool(ctx, msg.ForeignChain, msg.Name, msg.Symbol, uint8(msg.Decimals))
        if err != nil {
            return nil, sdkerrors.Wrapf(err, "failed to setupChainGasCoinAndPool")
        }
    } else {
        addr, err := k.DeployZRC20Contract(ctx, msg.Name, msg.Symbol, uint8(msg.Decimals), msg.ForeignChain, msg.CoinType, msg.ERC20, big.NewInt(int64(msg.GasLimit)))
        if err != nil {
            return nil, err
        }
```

Risk Level:

Likelihood - 2
Impact - 1

Recommendation:

We recommend being consistent with either uint8 or uint32 for this field, so there is no possibility for unexpected errors.

It is important to note that the likelihood of this error occurring is very low: the Keeper ensures that only Admin users can deploy contracts, and typical decimal values for smart contracts fit easily within an uint8 data type.

3.32 (HAL-32) USE OF DEPRECATED GO VERSION - INFORMATIONAL

Description:

The Docker environments used by ZetaChain use Go version 1.18. This version has been deprecated. See the Go release notes for their policy on supporting major versions of Go.

Code Location:

Dockerfile

Listing 50

```
FROM golang:1.18-alpine AS builder
```

contrib/localnet/orchestrator/Dockerfile

Listing 51

```
FROM zetanode:latest as zeta
FROM ethereum/client-go:v1.10.26 as geth
FROM golang:1.18-alpine as orchestrator
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Update to a supported version of Go in order to receive ongoing security updates.

3.33 (HAL-33) UNUSED FIELD GasLimit ON STRUCT MsgDeployFungibleCoinZRC20 - INFORMATIONAL

Description:

The field gasLimit in the struct MsgDeployFungibleCoinZRC20 is unused; instead, it defaults to 0 which causes the system to estimate the gas used to deploy a contract.

Code Location:

Defined: x/fungible/types/tx.pb.go

Used only here:
x/fungible/client/cli/tx_deploy_fungible_coin_zrc_4.go, Line 17

Listing 52

```
17 func CmdDeployFungibleCoinZRC4() *cobra.Command {
18     cmd := &cobra.Command{
19         Use:    "deploy-fungible-coin-zrc-4 [erc-20] [foreign-chain
↳ ] [decimals] [name] [symbol] [coin-type]",
20         Short: "Broadcast message DeployFungibleCoinZRC20",
21         Args:   cobra.ExactArgs(6),
22         RunE: func(cmd *cobra.Command, args []string) (err error)
↳ {
23             argERC20 := args[0]
24             argForeignChain := args[1]
25             argDecimals, err := strconv.ParseInt(args[2], 10, 32)
26             if err != nil {
27                 return err
28             }
29             argName := args[3]
30             argSymbol := args[4]
31             argCoinType, err := strconv.ParseInt(args[5], 10, 32)
32             if err != nil {
```

```
            return err
        }

        clientCtx, err := client.GetClientTxContext(cmd)
        if err != nil {
            return err
        }
        fmt.Printf("CLI address: %s\n", clientCtx.GetFromAddress().String())
        msg := types.NewMsgDeployFungibleCoinZRC20(
            clientCtx.GetFromAddress().String(),
            argERC20,
            argForeignChain,
            uint32(argDecimals),
            argName,
            argSymbol,
            common.CoinType(argCoinType),
        )
```

Processed by x/fungible/keeper/msg_server_deploy_fungible_coin_zrc_4.go

Listing 53

```
func (k msgServer) DeployFungibleCoinZRC20(goCtx context.Context, msg *types.MsgDeployFungibleCoinZRC20) (*types.MsgDeployFungibleCoinZRC20Response, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)
    if msg.Creator != k.zetaobserverKeeper.GetParams(ctx).GetAdminPolicyAccount(zetaObserverTypes.Policy_Type_deploy_fungible_coin) {
        return nil, sdkerrors.Wrap(sdkerrors.ErrUnauthorized, "Deploy can only be executed by the correct policy account")
    }
    if msg.CoinType == zetacommon.CoinType_Gas {
        _, err := k.setupChainGasCoinAndPool(ctx, msg.ForeignChain, msg.Name, msg.Symbol, uint8(msg.Decimals))
        if err != nil {
            return nil, sdkerrors.Wrapf(err, "failed to setupChainGasCoinAndPool")
        }
    } else {
        addr, err := k.DeployZRC20Contract(ctx, msg.Name, msg.Symbol, uint8(msg.Decimals), msg.ForeignChain, msg.CoinType, msg.
```

```
↳ ERC20, big.NewInt(int64(msg.GasLimit)))
26          if err != nil {
27              return nil, err
28          }
```

Note that there is also an overflow issue with this field. gasLimit is uint64 but converted to int64 in x/fungible/keeper/msg_server_deploy_fungible_coin_zrc_4.go L25 (for non-gas chains i.e. ERC20 tokens). This will result in an overflow to a large negative value, which may cause issues in deployment.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

If this field is not intended to be used, it can be removed from the struct. If it will be used in the future, avoid casting it to int64 or reject values that exceed MAX_INT64.

3.34 (HAL-34) INTEGER OVERFLOW CONVERSION FOR GAS PRICE IN BITCOIN SIGNER - INFORMATIONAL

Description:

The correct type for gasPrice as defined in the protobuf file uint256. When a Crosschain transaction is created, this value is represented by a string in x/crosschain/types/cross_chain_tx.pb.go. Later, it is converted to int64 in btc_singer.go. This creates a possible risk for an overflow issue, as an uint256 price may be converted to an int64. In this scenario, large gas prices could be interpreted as negative numbers.

This gas price is later processed by the Bitcoin signer in order to subtract a fee from the Value of a Bitcoin transaction. Should the fee overflow and become a large negative number, then the fee will actually increase the Value when the negative fee is subtracted. This could cause a transaction to revert, or else cause loss of funds.

In practice, it does not appear to be possible to create a transaction such that GetCurrentOutTxParam().OutboundTxGasPrice is a large enough value that it would overflow, as it would require that an extremely large amount of gas is subtracted from the source contract in the EVM. This is unlikely to occur either due to a low token balance in the ERC20 account, a low ERC20 allowance that would limit spending, or a corresponding overflow in Solidity that would cause the transaction to revert (given that the Solidity pragmas are greater than 0.8.0).

However, we recommend being consistent with these types so that there are no unexpected issues that arise should other aspects of the system change and create an opportunity for an overflow to occur.

Code Location:

zetaclient/btc_signer.go, Lines 207-227

Listing 54: The gas price is converted from a string to a big Int on Line 207. Later it is converted to a int64. The correct type is uint64.

```go
207     gasprice, ok := new(big.Int).SetString(send.GetCurrentOutTxParam().OutboundTxGasPrice, 10)
208     if !ok {
209         logger.Error().Msgf("cannot convert gas price  %s ", send.GetCurrentOutTxParam().OutboundTxGasPrice)
210         return
211     }
212     // FIXME: config chain params
213     addr, err := btcutil.DecodeAddress(string(toAddr), config.BitconNetParams)
214     if err != nil {
215         logger.Error().Err(err).Msgf("cannot decode address %s ", send.GetCurrentOutTxParam().Receiver)
216         return
217     }
218     to, ok := addr.(*btcutil.AddressWitnessPubKeyHash)
219     if err != nil || !ok {
220         logger.Error().Err(err).Msgf("cannot decode address %s ", send.GetCurrentOutTxParam().Receiver)
221         return
222     }
223
224     logger.Info().Msgf("SignWithdrawTx: to %s, value %d", addr.EncodeAddress(), send.GetCurrentOutTxParam().Amount.Uint64()/1e8)
225     logger.Info().Msgf("using utxos: %v", btcClient.utxos)
226     // FIXME: gas price?
227     tx, err := signer.SignWithdrawTx(to, float64(send.GetCurrentOutTxParam().Amount.Uint64())/1e8, float64(gasprice.Int64())/1e8*1024, btcClient.utxos, btcClient.pendingUtxos)
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

When converting types, ensure that a consistent value is used for each concept (such as gas price) in the system. If the value begins as uint64 it should also end up as a uint64. If it is necessary to use an alternate representation of the value, such as big.Int or string when passing a message from one system to another, ensure that no downcasting occurs. This means that a data type that holds large values should not be converted to one that cannot contain the entire range of values of the former. Extra caution should be used when converting between signed and unsigned types, especially when these values are used to calculate funds.

3.35 (HAL-35) REFERENCE TO DEPRECATED ETHEREUM NETWORK - INFORMATIONAL

Description:

The deprecated EVM test network Ropsten is referenced in testing code.

Code Location:

Listing 55: References to Ropsten

```
contrib/localnet/scripts/env.sh
6:export ROPSTEN_MPI_ADDRESS=0
↳ x000054d3A0Bc83Ec7808F52fCdC28A96c89F6C5c
14:#export ROPSTEN_POOL_ADDRESS=V2:0
↳ x3b45806771fa4508f11ec1601240e81f577a9fd1:ZETAETH
24:export ROPSTEN_ENDPOINT=https://ropsten.infura.io/v3/50
↳ b6673dc48443e59047246df462902c

cmd/zetaclientd/main.go
50:      enabledChains := flag.String("enable-chains", "GOERLI,
↳ BSCTESTNET,MUMBAI,ROPSTEN,BAOBAB", "enable chains, comma separated
↳  list")

cmd/zetacored/observerAccounts.go
69:                               "Ropsten"
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Test networks should be changed to connect with either Goerli or Sepolia.

3.36 (HAL-36) MESSAGE QueryBallotByIdentifierRequest RETURNS RESULTS FOR BALLOTS THAT DO NOT EXIST - INFORMATIONAL

Description:

When querying a ballot that does not exist via the command-line, a Ballot is returned. This is a result of the protocol returning a default Ballot with all of its values set to the equivalent of nil or zero for their respective types.

This may cause user confusion, as it is not clear that the Ballot does not exist.

Code Location:

x/observer/keeper/ballot.go

Listing 56: The variable voter is set to an empty Ballot when BallotIdentifier is not found.

```
func (k Keeper) BallotByIdentifier(goCtx context.Context, req *types.QueryBallotByIdentifierRequest) (*types.QueryBallotByIdentifierResponse, error) {
	if req == nil {
		return nil, status.Error(codes.InvalidArgument, "invalid request")
	}
	ctx := sdk.UnwrapSDKContext(goCtx)
	voter, _ := k.GetBallot(ctx, req.BallotIdentifier)
	return &types.QueryBallotByIdentifierResponse{Ballot: &voter}, nil
}
```

```
> zetacored q observer show-ballot 0
ballot:
  BallotThreshold: "0.000000000000000000"
  ballot_identifier: ""
  ballot_status: BallotFinalized_SuccessObservation
  index: ""
  observation_type: EmptyObserverType
  voter_list: []
  votes: []
> zetacored q observer show-ballot 9999999999999999999999999
ballot:
  BallotThreshold: "0.000000000000000000"
  ballot_identifier: ""
  ballot_status: BallotFinalized_SuccessObservation
  index: ""
  observation_type: EmptyObserverType
  voter_list: []
  votes: []
> zetacored q observer show-ballot 6c80546cdd972567769fbc747ecea44f528edba020ab36170013237a401cdb8f
ballot:
  BallotThreshold: "0.000000000000000000"
  ballot_identifier: ""
  ballot_status: BallotFinalized_SuccessObservation
  index: ""
  observation_type: EmptyObserverType
  voter_list: []
  votes: []
```

Figure 2: Results when querying Ballots that do not exist

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider whether it is more appropriate to return an error or another response in order to reduce ambiguity. This can improve the user experience of the protocol.

FINDINGS & TECH DETAILS

3.37 (HAL-37) DOCKER FILES USES A DIFFERENT GO VERSION THAN THE PROJECT - LOW

Description:

The Docker file uses Go version 1.18 whereas the project is configured to use version 1.19. This could cause subtle differences in code execution and, as a result, the testing environment may not replicate the execution of the deployed code.

Code Location:

go.mod

Listing 57

```
module github.com/zeta-chain/zetacore

go 1.19
```

Dockerfile

Listing 58

```
FROM golang:1.18-alpine AS builder
```

contrib/localnet/orchestrator/Dockerfile

Listing 59

```
FROM zetanode:latest as zeta
FROM ethereum/client-go:v1.10.26 as geth
FROM golang:1.18-alpine as orchestrator
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Use the same version of Go in the testing and production contexts.

3.38 (HAL-38) UPGRADING TO A MORE RECENT VERSION OF CosmosSDK COULD INCREASE PERFORMANCE - INFORMATIONAL

Description:

The project uses CosmosSDK version 0.46.8. The release v0.46.9 contains performance enhancements that could benefit the project.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Update to the latest release of CosmosSDK for performance and security enhancements.

Note that CosmosSDK v0.46.11 requires the use of Go 1.19. As a result, the Docker environment used by ZetaChain will need to be updated to use a newer version of Go.

3.39 (HAL-39) TESTING ENVIRONMENT IS USING OUTDATED BITCOIN DAEMON - INFORMATIONAL

Description:

The test environment uses Docker to set up a local Bitcoin daemon. It pulls the bitcoin-core software at version 22, which was released in September 2021. The latest version is 24.0.1.

Code Location:

contrib/localnet/docker-compose.yml

Listing 60

```
...
  bitcoin:
    image: ruimarinho/bitcoin-core:22 # version 23 is not working
 ↳ with btcd 0.22.0 due to change in createwallet rpc
    container_name: bitcoin
    hostname: bitcoin
    ...
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Where possible, use the most up-to-date software in order to benefit from security and performance enhancements. Test environments should match production contexts as closely as possible so that they can simulate the same operations that will occur in the real world.

INFORMATIONAL

Description:

price

Uint MsgGasPriceVoter

uint64

ParseInt ParseUint

MAX_INT64

price

Code Location:

x/crosschain/client/cli/cli_gas_price.go

Listing 61: The price parameter is parsed to an Int instead of a Uint

```
func CmdGasPriceVoter() *cobra.Command {
    cmd := &cobra.Command{
        Use:   "gas-price-voter [chain] [price] [supply] [
↳ blockNumber]",
        Short: "Broadcast message gasPriceVoter",
        Args:  cobra.ExactArgs(4),
        RunE: func(cmd *cobra.Command, args []string) error {
            argsChain, err := strconv.ParseInt(args[0], 10, 64)
            if err != nil {
                return err
            }
            argsPrice, err := strconv.ParseInt(args[1], 10, 64)
            if err != nil {
                return err
            }
            argsSupply := args[2]

            argsBlockNumber, err := strconv.ParseInt(args[3], 10,
↳ 64)
            if err != nil {
```

```
                return err
            }
            clientCtx, err := client.GetClientTxContext(cmd)
            if err != nil {
                return err
            }

            msg := types.NewMsgGasPriceVoter(clientCtx.GetFromAddress().String(), argsChain, uint64(argsPrice), argsSupply, uint64(argsBlockNumber))
            if err := msg.ValidateBasic(); err != nil {
                return err
            }
            return tx.GenerateOrBroadcastTxCLI(clientCtx, cmd.Flags(), msg)
        },
    }

    flags.AddTxFlagsToCmd(cmd)

    return cmd
}
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Use the function ParseUint instead of ParseInt in order to ensure a smooth user experience.

3.41 (HAL-41) DOCKER IGNORE FILE SHOULD INCLUDE GIT FILES - INFORMATIONAL

Description:

Docker is used in the repository for creating test images. The folder .git/ is 7.3M in size, making it the largest folder in the repository by far. By adding the entries .git, .github, and .gitignore to the .dockerignore file, it will be possible to speed up operation time when using Docker for testing.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Add Git and GitHub folders to .dockerignore in order to optimize builds.

3.42 (HAL-42) TODOS IN CODEBASE - INFORMATIONAL

Description:

Numerous code comments in the codebase contain **TODO** messages or other developer notes indicating malfunctioning or missing functionality.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to use a separate issue tracker or other task management software to track bugs and features rather than using code comments. Developer notes in comments are very likely to be overlooked and to become out of date relative to the code.

If the source code is shared publicly, such developer notes indicate areas of confusion or complexity which may be leveraged by an attacker reading the code.

3.43 (HAL-43) SPELLING MISTAKES IN CODE BASE - INFORMATIONAL

Description:

There are spelling mistakes in the codebase.

Code Location:

Here are a few examples of spelling mistakes in the codebase:

- In the file. zeta-node/src/common/chain.go, SigninAlgo should be SigningAlgo.
- x/observer/keeper/hooks.go defines a type BTCInTxEvnet struct, which should be called BTCInTxEvent.

x/fungible/keeper/systemcontract.go

Listing 62: Spelling mistake Facotry instead of Factory

```
func (k *Keeper) GetUniswapv2FacotryAddress(ctx sdk.Context) (
↳ ethcommon.Address, error) {
```

zetaclient/bitcoin_client.go L358, various mistakes:

Listing 63: Spelling mistakes

```
// CleanObservers cleans a observer Mapper checking delegation
↳ amount for a speficific delagator. It is used when delgator is the
↳  validator .
```

Risk Level:

Likelihood - 1
Impact - 1

FINDINGS & TECH DETAILS

Recommendation:

Fix all spelling mistakes. This can help convey a sense of professionalism to various project stakeholders.

3.44 (HAL-44) INCORRECT CODE COMMENTS - INFORMATIONAL

Description:

Some code comments in the codebase do not match the actual code.

Code Location:

Example 1: zetaclient/zetacore_observer.go, line 221.

The number 60 is used, but the number 50 is included in the comment.

Example 2:

x/crosschain/keeper/keeper_cross_chain_tx_vote_inbound_tx.go, Line 63-64.

The comment states that only two values are valid for the Cctx type in this context, but the code assigns it a third status (types.CctxStatus_PendingInbound).

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

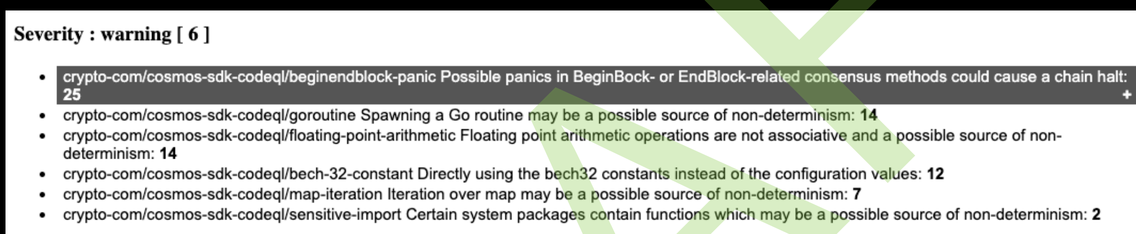
Fix the comments to reflect the code.

AUTOMATED TESTING

4.1 Automated Testing -- Overview

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped component. Among the tools used were **codeql**, **gosec**, **govulncheck** and **Nancy**. After Halborn verified all the modules and scoped structures in the repository and was able to compile them correctly, these tools were leveraged on scoped structures. With these tools, Halborn can statically verify security related issues across the entire codebase.

4.2 codeql



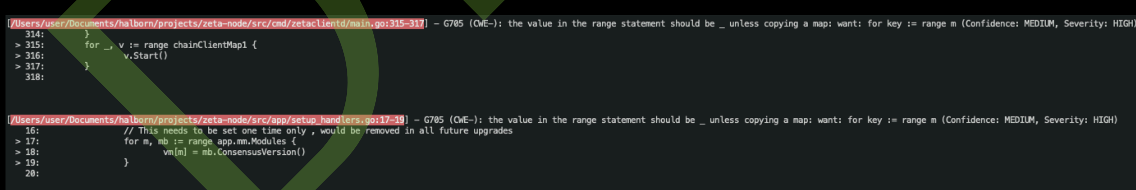
Severity : warning [6]

- crypto-com/cosmos-sdk-codeql/beginendblock-panic Possible panics in BeginBock- or EndBlock-related consensus methods could cause a chain halt: **25**
- crypto-com/cosmos-sdk-codeql/goroutine Spawning a Go routine may be a possible source of non-determinism: **14**
- crypto-com/cosmos-sdk-codeql/floating-point-arithmetic Floating point arithmetic operations are not associative and a possible source of non-determinism: **14**
- crypto-com/cosmos-sdk-codeql/bech-32-constant Directly using the bech32 constants instead of the configuration values: **12**
- crypto-com/cosmos-sdk-codeql/map-iteration Iteration over map may be a possible source of non-determinism: **7**
- crypto-com/cosmos-sdk-codeql/sensitive-import Certain system packages contain functions which may be a possible source of non-determinism: **2**

Figure 3: CodeQL results

4.3 gosec

The following as an excerpt from running the tool gosec:



```
[/Users/user/Documents/halborn/projects/zeta-node/src/cmd/zetaclientd/main.go:315-317] - G705 (CWE-): the value in the range statement should be _ unless copying a map: want: for key := range m (Confidence: MEDIUM, Severity: HIGH)
  314:          }
> 315:          for _, v := range chainClientMap1 {
> 316:                  v.Start()
> 317:          }
  318:


[/Users/user/Documents/halborn/projects/zeta-node/src/app/setup_handlers.go:17-19] - G705 (CWE-): the value in the range statement should be _ unless copying a map: want: for key := range m (Confidence: MEDIUM, Severity: HIGH)
  16:                   // This needs to be set one time only , would be removed in all future upgrades
> 17:                   for m, mb := range app.mm.Modules {
> 18:                           vm[m] = mb.ConsensusVersion()
> 19:                   }
  20:
```

Figure 4: gosec excerpt

4.4 nancy

The tool nancy was used to search for known vulnerabilities within project dependencies. Here is an excerpt of the output from this tool:

pkg:golang/github.com/tendermint/tendermint@v0.34.24
1 known vulnerabilities affecting installed version

1 vulnerability found	
Description	1 non-CVE vulnerability found. To see more details, please create a free account at https://ossindex.sonatype.org/ and request for this information using your registered account
OSS Index ID	sonatype-2021-0598
CVSS Score	4.8/10 (Medium)
CVSS Vector	CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:L/A:N
Link for more info	https://ossindex.sonatype.org/vulnerability/sonatype-2021-0598

pkg:golang/github.com/btcsuite/btcd@v0.22.1
2 known vulnerabilities affecting installed version

[CVE-2022-44797] CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	
Description	btcd before 0.23.2, as used in Lightning Labs lnd before 0.15.2-beta and other Bitcoin-related products, mishandles witness size checking.
OSS Index ID	CVE-2022-44797
CVSS Score	9.8/10 (Critical)
CVSS Vector	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
Link for more info	https://ossindex.sonatype.org/vulnerability/CVE-2022-44797?component-type=golang&component-name=github.com%2Fbtcsuite%2Fbtcd&utm_source=nancy-client&utm_medium=integration&utm_content=1.0.42

[CVE-2022-39389] CWE-20: Improper Input Validation	
Description	Lightning Network Daemon (lnd) is an implementation of a lightning bitcoin overlay network node. All lnd nodes before version `v0.15.4` are vulnerable to a block parsing bug that can cause a node to enter a degraded state once encountered. In this degraded state, nodes can continue to make payments and forward HTLCs, and close out channels. Opening channels is prohibited, and also on chain transaction events will be undetected. This can cause loss of funds if a CSV expiry is researched during a breach attempt or a CLTV delta expires forgetting the funds in the HTLC. A patch is available in `lnd` version 0.15.4. Users are advised to upgrade. Users unable to upgrade may use the `lncli updatechanpolicy` RPC call to increase their CLTV value to a very high amount or increase their fee policies. This will prevent nodes from routing through your node, meaning that no pending HTLCs can be present.
OSS Index ID	CVE-2022-39389
CVSS Score	6.5/10 (Medium)
CVSS Vector	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:L
Link for more info	https://ossindex.sonatype.org/vulnerability/CVE-2022-39389?component-type=golang&component-name=github.com%2Fbtcsuite%2Fbtcd&utm_source=nancy-client&utm_medium=integration&utm_content=1.0.42

Figure 5: Nancy excerpt

4.5 Fuzz Testing

Fuzz testing, also known as fuzzing, is a software testing technique that involves inputting large amounts of random data, or fuzz, into a program to see how it reacts and if it can handle unexpected or invalid input. The goal of fuzz testing is to identify and prevent software vulnerabilities, such as buffer overflows and memory leaks, by exposing the program to a wide range of inputs that it may not have been designed to handle.

Halborn performed fuzz testing on the function getSatoshis defined in the file zetaclient/btc_util.go.

This function was selected because it contains custom logic to convert Bitcoin to its representation in Satoshis. Furthermore, it works with the float type in Go. Mathematical operations for floats are a common source of problems and can have a high impact in blockchain projects when they are used to representing currencies.

Fuzz Harness:

Listing 64: Fuzz harness with test cases and properties defined

```go
func FuzzGetSatoshis(f *testing.F) {
    // Both test cases testing the limits of the Float64 data type will cause a crash
    testcases := []float64{
        0,
        math.SmallestNonzeroFloat64,
        math.MaxFloat64,
    }
    for _, tc := range testcases {
        f.Add(tc) // Use f.Add to provide a seed corpus
    }
    f.Fuzz(func(t *testing.T, input float64) {
        sats, err := GetSatoshis(input)
        if err != nil {
            t.Errorf("Got error: %s; Input: %g", err.Error(), input)
        }
        if input == 0 && sats != 0 {
            t.Errorf("Input of zero results in non-zero output: %g, after: %d", input, sats)
        }
        if sats > 0 && !(input > 0) {
            t.Errorf("Signed changed: %g, after: %d", input, sats)
        }
        if sats < 0 && !(input < 0) {
            t.Errorf("Signed changed: %g, after: %d", input, sats)
        }

        // Converting to satoshis should always result in a higher number w.r.t. magnitude compared with the input
        if math.Abs(input) > math.Abs(float64(sats)) {
            t.Errorf("Absolute value of input %g is greater than absolute value of output %g", math.Abs(input), math.Abs(float64(sats)))
        }
    })
}
```

Summary:

Halborn identified two crashes while fuzzing this function. Pictured below is an example:

```
fuzz: elapsed: 0s, gathering baseline coverage: 0/116 completed
failure while testing seed corpus entry: FuzzGetSatoshis/seed#1
fuzz: elapsed: 0s, gathering baseline coverage: 0/116 completed
--- FAIL: FuzzGetSatoshis (0.25s)
    --- FAIL: FuzzGetSatoshis (0.00s)
        btc_util_test.go:35: Absolute value of input 5e-324 is greater than absolute value of output 0

FAIL
exit status 1
FAIL    github.com/zeta-chain/zetacore/zetaclient      2.246s
```

Figure 6: Results of fuzz testing

Other notes:

There are several limitations to fuzz testing when done with a time constraint, especially regarding security:

- Limited coverage: When done with a time constraint, the amount of fuzzing that can be done is limited, which means that the software may not be fully tested and vulnerabilities may go undetected.
- False negatives: Fuzz testing may not be able to uncover all vulnerabilities in the software, especially when done with a time constraint. This is particularly true for complex software systems, where a significant amount of time is required to uncover all possible vulnerabilities.
- Limited scope: Only a section of the codebase was fuzzed.

THANK YOU FOR CHOOSING

// HALBORN