HALBORN

ZetaChain - ZetaNode

Cosmos Security Assessment

Prepared by: **Halborn**

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

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY 14

CONTACTS 14

1 EXECUTIVE OVERVIEW 15

1.1 INTRODUCTION 16

1.2 ASSESSMENT SUMMARY 17

1.3 TEST APPROACH & METHODOLOGY 19

2 RISK METHODOLOGY 20

2.1 EXPLOITABILITY 21

2.2 IMPACT 22

2.3 SEVERITY COEFFICIENT 24

2.4 SCOPE 26

3 ASSESSMENT SUMMARY & FINDINGS OVERVIEW 27

4 FINDINGS & TECH DETAILS 31

4.1 (HAL-01) ZETA SUPPLY DOES NOT TRACK ASSETS CORRECTLY - CRITICAL(10) 32

Description 32

Code Location & Proof Of Concept 33

BVSS 37

Recommendation 37

Remediation Plan 37

4.2 (HAL-02) OVERFLOW IN ZETA BLOCK HEIGHT CAUSES EXPONENTIAL INCREASE IN GAS PRICE FOR ALL PENDING TRANSACTIONS - CRITICAL(10) 38

Description 38

Code Location 38

BVSS 40

Recommendation 40

Remediation Plan 41

4.3 (HAL-03) POSSIBLE DIVISION BY ZERO COULD CAUSE CHAIN HALT DUE TO PANIC - CRITICAL(10) 42

Description 42

Code Location 42

Proof Of Concept 43

BVSS 44

Recommendation 44

Remediation Plan 44

4.4 (HAL-04) BITCOIN TRANSACTIONS REQUIRE ONLY ONE CONFIRMATION - CRITICAL(10) 45

Description 45

Code Location 45

Proof Of Concept 46

BVSS 47

Recommendation 47

Remediation Plan 47

4.5 (HAL-05) LACK OF MECHANISM TO LIMIT SUPPLY OF ZETA - HIGH(7.8) 48

Description 48

BVSS 48

Recommendation 48

Remediation Plan 48

4.6 (HAL-06) PRICE MANIPULATION AND DENIAL-OF-SERVICE VIA UpdatePrices FUNCTION - HIGH(7.8) 49
Description 49
Code Location 49
BVSS 51
Recommendation 52
Remediation Plan 52
4.7 (HAL-07) ERROR CONDITION FOR KEY SIGNING IS UNCHECKED - HIGH(7.8) 53
Description 53
Code Location 53
BVSS 54
Recommendation 54
Remediation Plan 54
4.8 (HAL-08) ITERATION OVER MAPS MAY BE A SOURCE OF NON-DETERMINISM - HIGH(7.8) 55
Description 55
Code Location 55
BVSS 55
Recommendation 56
Remediation Plan 56
4.9 (HAL-09) SYBIL ATTACK RISK DUE TO USE OF MEDIAN GAS VOTES FOR SETTING GAS PRICE - HIGH(7.8) 57
Description 57
Code Location 58
BVSS 61
Recommendation 61

Remediation Plan 61

4.10 (HAL-10) MALICIOUS GAS PRICE VOTING: DENIAL-OF-SERVICE BY SETTING LARGE GAS PRICES FOR EVM NETWORKS - MEDIUM(5.5) 62

Description 62

Code Location 62

BVSS 62

Recommendation 63

Remediation Plan 63

4.11 (HAL-11) MALICIOUS GAS PRICE VOTING: DENIAL-OF-SERVICE OR PRICE MANIPULATION BY SETTING GAS PRICES FOR BITCOIN - MEDIUM(5.5) 64

Description 64

Code Location 66

BVSS 66

Recommendation 66

Remediation Plan 66

4.12 (HAL-12) INTEGER OVERFLOW BREAKS GRPC COMMUNICATION FOR LARGE BLOCK HEIGHTS - MEDIUM(5.5) 67

Description 67

Code Location 67

BVSS 70

Recommendation 70

Remediation Plan 71

4.13 (HAL-13) RELIANCE ON UNISWAPV2 POOLS FOR PRICES EXPOSES ZETACHAIN TO PRICE MANIPULATION RISK - MEDIUM(5.5) 72

Description 72

BVSS 72

Recommendation 72

Remediation Plan 72

4.14 (HAL-14) ARBITRARY MINTING OF ZETA VIA MintZetaToEVMAccount FUNCTION - MEDIUM(5.5) 73

Description 73

BVSS 73

Recommendation 73

Remediation Plan 73

4.15 (HAL-15) SUPPORT FOR A TOKEN CANNOT BE REMOVED FROM THE PROTOCOL - MEDIUM(5.5) 74

Description 74

Code Location 74

BVSS 75

Recommendation 76

Remediation Plan 76

4.16 (HAL-16) USE OF VULNERABLE COSMOSSDK VERSION - LOW(2.3) 77

Description 77

Code Location 77

BVSS 77

Recommendation 77

Remediation Plan 77

4.17 (HAL-17) ValidateBasic INCOMPLETE FOR SOME MESSAGE TYPES - LOW(2.3) 79

Description 79

Code Location 79

BVSS 80

Recommendation 80
Remediation Plan 80
4.18 (HAL-18) CENTRALIZATION RISK - LOW(2.3) 81
Description 81
Code Location 81
BVSS 82
Recommendation 82
Remediation Plan 82
4.19 (HAL-19) LACK OF UNIT TESTS - LOW(2.3) 83
Description 83
Code Location 83
BVSS 85
Recommendation 85
Remediation Plan 85
4.20 (HAL-20) LACK OF FUZZ TESTS - LOW(2.3) 86
Description 86
BVSS 86
Recommendation 86
Remediation Plan 86
4.21 (HAL-21) BITCOIN TRANSACTIONS WITH LARGE NONCE VALUES MAY BE SENT REPEATEDLY - LOW(2.3) 87
Description 87
Code Location 87
BVSS 91
Recommendation 91
Remediation Plan 91

4.22 (HAL-22) UNBOUNDED ARRAY IN Signers IN ChainNonces COULD CAUSE RESOURCE EXHAUSTION - LOW(2.3) 92
Description 92
Code Location 92
BVSS 94
Recommendation 94
Remediation Plan 94
4.23 (HAL-23) USE OF VULNERABLE DEPENDENCIES - LOW(2.3) 95
Description 95
Code Location 95
BVSS 95
Recommendation 96
Remediation Plan 96
4.24 (HAL-24) MNEMONIC PHRASES PRESENT IN CODEBASE - LOW(2.3) 97
Description 97
Code Location 97
BVSS 98
Recommendation 98
Remediation Plan 98
4.25 (HAL-25) VULNERABLE TSS-LIB CONTAINS HASH COLLISION ISSUES - LOW(2.3) 99
Description 99
Code Location 99
BVSS 99

Recommendation 100
Remediation Plan 100
4.26 (HAL-26) CALCULATION ERRORS IN getSatoshis FUNCTION FOR EXTREMELY SMALL OR LARGE FLOAT VALUES - LOW(2.3) 101
Description 101
Code Location 101
BVSS 102
Recommendation 102
Remediation Plan 102
4.27 (HAL-27) GAS LIMITS CANNOT BE CONFIGURED - LOW(2.3) 103
Description 103
Code Location 103
BVSS 104
Recommendation 104
Remediation Plan 104
4.28 (HAL-28) VALIDATORS CANNOT EFFECTIVELY SET THE ZETA GAS PRICE FOR BITCOIN TRANSACTIONS - LOW(2.3) 105
Description 105
Code Location 107
BVSS 108
Recommendation 109
Remediation Plan 109
4.29 (HAL-29) GO VERSIONS PRIOR TO 1.20.2 CONTAIN CRYPTOGRAPHIC ISSUES AND OTHER BUGS - LOW(2.3) 110
Description 110

Code Location 110
BVSS 110
Recommendation 110
Remediation Plan 110
4.30 (HAL-30) ZETACLIENT AND ZETACORE TRACK BLOCK HEIGHT USING DIFFERENT TYPES - LOW(2.3) 111
Description 111
Code Location 111
BVSS 113
Recommendation 113
Remediation Plan 113
4.31 (HAL-31) TYPE CONVERSION ISSUE FOR FIELD Decimals ON STRUCT MsgDeployFungibleCoinZRC20 - INFORMATIONAL(0.0) 114
Description 114
Code Location 114
BVSS 115
Recommendation 115
Remediation Plan 115
4.32 (HAL-32) USE OF DEPRECATED GO VERSION - INFORMATIONAL(0.0) 116
Description 116
Code Location 116
BVSS 116
Recommendation 116
Remediation Plan 117
4.33 (HAL-33) UNUSED FIELD GasLimit ON STRUCT MsgDeployFungibleCoinZRC20 - INFORMATIONAL(0.0) 118
Description 118

Code Location 118
BVSS 120
Recommendation 120
Remediation Plan 121
4.34 (HAL-34) INTEGER OVERFLOW CONVERSION FOR GAS PRICE IN BITCOIN SIGNER - INFORMATIONAL(0.0) 122
Description 122
Code Location 122
BVSS 123
Recommendation 123
Remediation Plan 124
4.35 (HAL-35) REFERENCE TO DEPRECATED ETHEREUM NETWORK - INFORMATIONAL(0.0) 125
Description 125
Code Location 125
BVSS 125
Recommendation 125
Remediation Plan 126
4.36 (HAL-36) MESSAGE QueryBallotByIdentifierRequest RETURNS RESULTS FOR BALLOTS THAT DO NOT EXIST - INFORMATIONAL(0.0) 127
Description 127
Code Location 127
BVSS 128
Recommendation 128
Remediation Plan 128
4.37 (HAL-37) DOCKER FILES USES A DIFFERENT GO VERSION THAN THE PROJECT - INFORMATIONAL(0.0) 129
Description 129

Code Location 129
BVSS 130
Recommendation 130
Remediation Plan 130
4.38 (HAL-38) UPGRADING TO A MORE RECENT VERSION OF CosmosSDK COULD INCREASE PERFORMANCE - INFORMATIONAL(0.0) 131
Description 131
BVSS 131
Recommendation 131
Remediation Plan 131
4.39 (HAL-39) TESTING ENVIRONMENT IS USING OUTDATED BITCOIN DAEMON - INFORMATIONAL(0.0) 132
Description 132
Code Location 132
BVSS 132
Recommendation 132
Remediation Plan 133
4.40 (HAL-40) GAS PRICE VOTER CLI INTERFACE USES THE WRONG TYPE - INFORMATIONAL(0.0) 134
Description 134
Code Location 134
BVSS 135
Recommendation 135
Remediation Plan 135
4.41 (HAL-41) DOCKER IGNORE FILE SHOULD INCLUDE GIT FILES - INFORMATIONAL(0.0) 136
Description 136

BVSS 136
Recommendation 136
Remediation Plan 136
4.42 (HAL-42) TODOS IN CODEBASE - INFORMATIONAL(0.0) 137
Description 137
BVSS 137
Recommendation 137
Remediation Plan 137
4.43 (HAL-43) SPELLING MISTAKES IN CODE BASE - INFORMATIONAL(0.0) 138
Description 138
Code Location 138
BVSS 138
Recommendation 139
Remediation Plan 139
4.44 (HAL-44) INCORRECT CODE COMMENTS - INFORMATIONAL(0.0) 140
Description 140
Code Location 140
BVSS 140
Recommendation 140
Remediation Plan 140
5 AUTOMATED TESTING 141
5.1 Automated Testing -- Overview 142
5.2 codeql 142
5.3 gosec 142

5.4 nancy 142

5.5 Fuzz Testing 143

Fuzz Harness 144

Summary 145

Other notes 145

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE
0.1	Document Creation	02/26/2023
0.2	Document Updates	03/31/2023
0.3	Document Updates	04/04/2023
0.4	Draft Review	04/04/2023
0.5	Draft Review	04/04/2023
1.0	Remediation Plan	10/16/2023
1.1	Remediation Plan Review	10/18/2023

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

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

ZetaChain engaged Halborn to conduct a security assessment 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 assessment covering the Solidity smart contracts was also performed with this assessment, and the resulting report should be viewed as a companion to this one.

1.2 ASSESSMENT SUMMARY

The team at Halborn was provided about four weeks for the engagement and assigned a full-time security engineer to verify the security of the project. 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 assessment 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 that were mostly addressed by the ZetaChain team before the project was 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 assessment:

- 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.

2. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

2.1 EXPLOITABILITY

Attack Origin (AO):

Captures whether the attack requires compromising a specific account.

Attack Cost (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

Attack Complexity (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

Metrics:

Exploitability Metric (m_E)	Metric Value	Numerical Value
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

2.2 IMPACT

Confidentiality (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

Integrity (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

Availability (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

Deposit (D):

Measures the impact to the deposits made to the contract by either users or owners.

Yield (Y):

Measures the impact to the yield generated by the contract for either users or owners.

Metrics:

Impact Metric (m_I)	Metric Value	Numerical Value
Confidentiality (C)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium: (Y:M)	0.5
	High: (Y:H)	0.75
	Critical (Y:H)	1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

2.3 SEVERITY COEFFICIENT

Reversibility (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

Scope (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

Coefficient (C)	Coefficient Value	Numerical Value
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

Severity	Score Value Range
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

2.4 SCOPE

1. IN-SCOPE TREE & COMMIT :

The assessment 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. REMEDIATION COMMITS :

- 95f28b04a7b452d21021a946d6b7fff3f4d8a81e
- 31b169aab1aaa3a6fd823ff1828a1dd5a3a037de
- 3f5d0782f73dee061d60bbde1106d25c1706a504
- 4f8333b6677f9deea34938222bb287fcfb97440f
- ee0ac1faf0c9d5fe97fd582d1be53dbf3d6c8926
- a5aa77681ced5ea20cacdcfb6bbd5bdb385bdd2c
- 37de781c228c08fa79ef3c1ce45da9d8a685e2a7
- 490f57435bd4fd1bfbc4a55ebece4b93376c2d76
- 007ff12dacf5142933bdce06ad2da724a992bd6e
- 1e1e73024d21716cce1b39b401027690c4a13083
- a1c06f4698d371f374a72cdd6e152209b8109dc2
- a3d936b7856ab2b77e5486de6b4c0177e6e90f23

3. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
4	5	6	15	14

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) ZETA SUPPLY DOES NOT TRACK ASSETS CORRECTLY	Critical (10)	SOLVED - 10/16/2023
(HAL-02) OVERFLOW IN ZETA BLOCK HEIGHT CAUSES EXPONENTIAL INCREASE IN GAS PRICE FOR ALL PENDING TRANSACTIONS	Critical (10)	SOLVED - 10/16/2023
(HAL-03) POSSIBLE DIVISION BY ZERO COULD CAUSE CHAIN HALT DUE TO PANIC	Critical (10)	RISK ACCEPTED
(HAL-04) BITCOIN TRANSACTIONS REQUIRE ONLY ONE CONFIRMATION	Critical (10)	SOLVED - 10/16/2023
(HAL-05) LACK OF MECHANISM TO LIMIT SUPPLY OF ZETA	High (7.8)	SOLVED - 10/16/2023
(HAL-06) PRICE MANIPULATION AND DENIAL-OF-SERVICE VIA UpdatePrices FUNCTION	High (7.8)	SOLVED - 10/16/2023
(HAL-07) ERROR CONDITION FOR KEY SIGNING IS UNCHECKED	High (7.8)	RISK ACCEPTED
(HAL-08) ITERATION OVER MAPS MAY BE A SOURCE OF NON-DETERMINISM	High (7.8)	SOLVED - 10/16/2023
(HAL-09) SYBIL ATTACK RISK DUE TO USE OF MEDIAN GAS VOTES FOR SETTING GAS PRICE	High (7.8)	RISK ACCEPTED
(HAL-10) MALICIOUS GAS PRICE VOTING: DENIAL-OF-SERVICE BY SETTING LARGE GAS PRICES FOR EVM NETWORKS	Medium (5.5)	SOLVED - 10/16/2023
(HAL-11) MALICIOUS GAS PRICE VOTING: DENIAL-OF-SERVICE OR PRICE MANIPULATION BY SETTING GAS PRICES FOR BITCOIN	Medium (5.5)	SOLVED - 10/16/2023
(HAL-12) INTEGER OVERFLOW BREAKS GRPC COMMUNICATION FOR LARGE BLOCK HEIGHTS	Medium (5.5)	SOLVED - 10/16/2023
(HAL-13) RELIANCE ON UNISWAPV2 POOLS FOR PRICES EXPOSES ZETACHAIN TO PRICE MANIPULATION RISK	Medium (5.5)	RISK ACCEPTED

(HAL-14) ARBITRARY MINTING OF ZETA VIA MintZetaToEVMAccount FUNCTION	Medium (5.5)	SOLVED - 10/16/2023
(HAL-15) SUPPORT FOR A TOKEN CANNOT BE REMOVED FROM THE PROTOCOL	Medium (5.5)	SOLVED - 10/16/2023
(HAL-16) USE OF VULNERABLE COSMOSSDK VERSION	Low (2.3)	SOLVED - 10/16/2023
(HAL-17) ValidateBasic INCOMPLETE FOR SOME MESSAGE TYPES	Low (2.3)	SOLVED - 10/16/2023
(HAL-18) CENTRALIZATION RISK	Low (2.3)	RISK ACCEPTED
(HAL-19) LACK OF UNIT TESTS	Low (2.3)	RISK ACCEPTED
(HAL-20) LACK OF FUZZ TESTS	Low (2.3)	RISK ACCEPTED
(HAL-21) BITCOIN TRANSACTIONS WITH LARGE NONCE VALUES MAY BE SENT REPEATEDLY	Low (2.3)	RISK ACCEPTED
(HAL-22) UNBOUNDED ARRAY IN Signers IN ChainNonces COULD CAUSE RESOURCE EXHAUSTION	Low (2.3)	SOLVED - 10/16/2023
(HAL-23) USE OF VULNERABLE DEPENDENCIES	Low (2.3)	SOLVED - 10/16/2023
(HAL-24) MNEMONIC PHRASES PRESENT IN CODEBASE	Low (2.3)	RISK ACCEPTED
(HAL-25) VULNERABLE TSS-LIB CONTAINS HASH COLLISION ISSUES	Low (2.3)	RISK ACCEPTED
(HAL-26) CALCULATION ERRORS IN getSatoshis FUNCTION FOR EXTREMELY SMALL OR LARGE FLOAT VALUES	Low (2.3)	SOLVED - 10/16/2023
(HAL-27) GAS LIMITS CANNOT BE CONFIGURED	Low (2.3)	RISK ACCEPTED
(HAL-28) VALIDATORS CANNOT EFFECTIVELY SET THE ZETA GAS PRICE FOR BITCOIN TRANSACTIONS	Low (2.3)	RISK ACCEPTED
(HAL-29) GO VERSIONS PRIOR TO 1.20.2 CONTAIN CRYPTOGRAPHIC ISSUES AND OTHER BUGS	Low (2.3)	RISK ACCEPTED

(HAL-30) ZETACLIENT AND ZETACORE TRACK BLOCK HEIGHT USING DIFFERENT TYPES	Low (2.3)	RISK ACCEPTED
(HAL-31) TYPE CONVERSION ISSUE FOR FIELD Decimals ON STRUCT MsgDeployFungibleCoinZRC20	Informational (0.0)	SOLVED - 10/16/2023
(HAL-32) USE OF DEPRECATED GO VERSION	Informational (0.0)	SOLVED - 10/16/2023
(HAL-33) UNUSED FIELD GasLimit ON STRUCT MsgDeployFungibleCoinZRC20	Informational (0.0)	SOLVED - 10/16/2023
(HAL-34) INTEGER OVERFLOW CONVERSION FOR GAS PRICE IN BITCOIN SIGNER	Informational (0.0)	ACKNOWLEDGED
(HAL-35) REFERENCE TO DEPRECATED ETHEREUM NETWORK	Informational (0.0)	ACKNOWLEDGED
(HAL-36) MESSAGE QueryBallotByIdentifierRequest RETURNS RESULTS FOR BALLOTS THAT DO NOT EXIST	Informational (0.0)	SOLVED - 10/16/2023
(HAL-37) DOCKER FILES USES A DIFFERENT GO VERSION THAN THE PROJECT	Informational (0.0)	ACKNOWLEDGED
(HAL-38) UPGRADING TO A MORE RECENT VERSION OF CosmosSDK COULD INCREASE PERFORMANCE	Informational (0.0)	ACKNOWLEDGED
(HAL-39) TESTING ENVIRONMENT IS USING OUTDATED BITCOIN DAEMON	Informational (0.0)	ACKNOWLEDGED
(HAL-40) GAS PRICE VOTER CLI INTERFACE USES THE WRONG TYPE	Informational (0.0)	SOLVED - 10/16/2023
(HAL-41) DOCKER IGNORE FILE SHOULD INCLUDE GIT FILES	Informational (0.0)	ACKNOWLEDGED
(HAL-42) TODOS IN CODEBASE	Informational (0.0)	ACKNOWLEDGED
(HAL-43) SPELLING MISTAKES IN CODE BASE	Informational (0.0)	ACKNOWLEDGED
(HAL-44) INCORRECT CODE COMMENTS	Informational (0.0)	ACKNOWLEDGED

FINDINGS & TECH DETAILS

4.1 (HAL-01) ZETA SUPPLY DOES NOT TRACK ASSETS CORRECTLY - CRITICAL(10)

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 assessment, 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 assessment, 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

```
// 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()))
    }

```

```
28      amountString := strings.TrimSuffix(bankRes.Supply.String(), "
↳ azeta") // Response supply string is in format like
↳ '200010000200000000000000000azeta'. Remove suffix
29      supply := new(big.Int)
30      _, ok := supply.SetString(amountString, 10)
31      if !ok {
32          panic("Problem parsing azeta supply string to big Int")
33      }
34      return supply
35  }
36
37  // use sm here to make use of zevmClient
38  func GetTotalZetaEVM(sm *SmokeTest) *big.Int {
39      // Taken from test_zeta_in_and_out.go
40      wzetaAddr := ethcommon.HexToAddress("0
↳ x5F0b1a82749cb4E2278EC87F8BF6B618dC71a8bf")
41      zevmClient := sm.zevmClient
42      wzeta, err := zevm.NewWZETA(wzetaAddr, zevmClient)
43      if err != nil {
44          panic(fmt.Sprintf("Problem calling newWZeta. Error: `%s`",
↳  err.Error()))
45      }
46      supply, err := wzeta.TotalSupply(&bind.CallOpts{}) // returns
↳ a big.Int (contracts/zevm/WZETA.go)
47      if err != nil {
48          panic(fmt.Sprintf("Problem querying total supply of wzeta
↳ in EVM. Error: `%s`", err.Error()))
49      }
50      return supply
51  }
52
53  func InvariantTotalSupplyIncreased(previousSupply *big.Int,
↳ totalCosmos *big.Int, totalEVM *big.Int) (newSupply *big.Int, err
↳ error) {
54      fmt.Printf("Total azeta in Cosmos: %d\n", totalCosmos)
55      fmt.Printf("Total Zeta in EVM: %d\n", totalEVM)
56      newSupply = new(big.Int)
57      newSupply = newSupply.Add(totalCosmos, totalEVM) //
↳ totalCosmos + totalEVM
58      fmt.Printf("Previous supply: %d\n", previousSupply)
59      fmt.Printf("New supply: %d\n", newSupply)
60      if newSupply == nil {
61          panic("New supply should never be nil in
↳ InvariantTotalSupplyIncreased")
```

```
62      }
63      if newSupply.Cmp(previousSupply) > 0 { // newSupply >
↳ previousSupply
64          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))
65      }
66      return newSupply, nil
67  }
68
69  func PostTestInvariantCheck(previousSupply *big.Int, totalCosmos *
↳ big.Int, totalEVM *big.Int) (newSupply *big.Int, err error) {
70      newSupply, err = InvariantTotalSupplyIncreased(previousSupply,
↳  totalCosmos, totalEVM)
71      if err != nil {
72          LoudPrintf(err.Error())
73          return newSupply, err
74      }
75      if newSupply == nil {
76          panic("New supply returned from
↳ InvariantTotalSupplyIncreased is nil")
77      }
78      return newSupply, nil
79  }
```

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

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

```
↳ ====================
 9     startTime := time.Now()
10     smokeTest.TestBitcoinSetup()
11     supply, totalCosmos, totalEVM = InvariantCheck(smokeTest,
↳ supply)
12
13     smokeTest.TestSetupZetaTokenAndConnectorContracts()
14     supply, totalCosmos, totalEVM = InvariantCheck(smokeTest,
↳ supply)
15     ...
16
17 ```{language=go caption="Function added to smoke test code in main
↳ .go}```
18 func InvariantCheck(sm *SmokeTest, previousSupply *big.Int) (
↳ newSupply *big.Int, totalCosmos *big.Int, totalEVM *big.Int) {
19     if previousSupply == nil {
20         panic("Previous supply should never be nil in
↳ InvariantCheck")
21     }
22     totalEVM = GetTotalZetaEVM(sm)
23     totalCosmos = GetTotalZetaCosmos()
24     newSupply, _ = PostTestInvariantCheck(previousSupply,
↳ totalCosmos, totalEVM)
25     return
26 }
```

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

```
1 2023-03-30 14:52:22 ========================================
2 2023-03-30 14:52:22 Initial supplies: EVM = 100000000010000000
3 2023-03-30 14:52:22 Cosmos = 2000100002000000000000000000
4 2023-03-30 14:52:22 Total = 2000100003000000000010000000
5 2023-03-30 14:52:22 ========================================
6 ...
7 2023-03-30 14:53:48 Total azeta in Cosmos:
↳ 2000100002000000000000000000
8 2023-03-30 14:53:48 Total Zeta in EVM: 100000000010000000
9 2023-03-30 14:53:48 Previous supply: 2000100003000000000010000000
10 2023-03-30 14:53:48 New supply: 2000100003000000000010000000
```

```
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: 20001010030000000010000000
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 20001010030000000010000000
2023-03-30 14:54:06 (azetaSupply 200010100200000000000000000 +
↳ wzetaSupply 100000000010000000)
2023-03-30 14:54:06 ========================================
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:C/Y:N/R:N/S:C (10)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by fixing the asset tracking.

Commit ID : 95f28b04a7b452d21021a946d6b7fff3f4d8a81e

4.2 (HAL-02) OVERFLOW IN ZETA BLOCK HEIGHT CAUSES EXPONENTIAL INCREASE IN GAS PRICE FOR ALL PENDING TRANSACTIONS - CRITICAL(10)

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

```
80 func (am AppModule) EndBlock(ctx sdk.Context, _ abci.
↳ RequestEndBlock) []abci.ValidatorUpdate {
81     //am.keeper.InitializeGenesisKeygen(sdk.WrapSDKContext(ctx))
82     am.keeper.ScrubGasPriceOfStuckOutTx(sdk.WrapSDKContext(ctx))
83     return []abci.ValidatorUpdate{}
84 }
```

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

```
31 func (k Keeper) ScrubUtility(ctx sdk.Context, store sdk.KVStore, p
↳  []byte) {
32     sendStore := prefix.NewStore(store, p)
33     iterator := sdk.KVStorePrefixIterator(sendStore, []byte{})
34     defer iterator.Close()
35     for ; iterator.Valid(); iterator.Next() {
36         var cctx types.CrossChainTx
37         k.cdc.MustUnmarshal(iterator.Value(), &cctx)
38         // if the status of send is pending, which means Finalized
↳ /Revert
39         if cctx.CctxStatus.Status == types.
↳ CctxStatus_PendingOutbound || cctx.CctxStatus.Status == types.
↳ CctxStatus_PendingRevert {
40             if ctx.BlockHeight()-int64(cctx.InboundTxParams.
↳ InboundTxFinalizedZetaHeight) > 100 { // stuck send
41                 var chainID int64
42                 currentOutTxParam := cctx.GetCurrentOutTxParam()
43                 chainID = currentOutTxParam.ReceiverChainId
44
45                 gasPrice, isFound := k.GetGasPrice(ctx, chainID)
46                 if !isFound {
47                     continue
48                 }
49                 mi := gasPrice.MedianIndex
50                 newGasPrice := big.NewInt(0).SetUint64(gasPrice.
↳ Prices[mi])
51                 oldGasPrice, ok := big.NewInt(0).SetString(
↳ currentOutTxParam.OutboundTxGasPrice, 10)
52                 if !ok {
53                     k.Logger(ctx).Error("failed to parse old gas
↳ price")
```

```
54                      continue
55                  }
56                  // do nothing if new gas price is even lower than
↳ old price
57                  if newGasPrice.Cmp(oldGasPrice) < 0 {
58                      continue
59                  }
60                  targetGasPrice := oldGasPrice.Mul(oldGasPrice, big
↳ .NewInt(4))
61                  targetGasPrice = targetGasPrice.Div(targetGasPrice
↳ , big.NewInt(3)) // targetGasPrice = oldGasPrice * 1.2
62                  // if current new price is not much higher; make
↳ it at least 20% higher
63                  // otherwise replacement tx will be rejected by
↳ the node
64                  if newGasPrice.Cmp(targetGasPrice) < 0 {
65                      newGasPrice = targetGasPrice
66                  }
67                  currentOutTxParam.OutboundTxGasPrice = newGasPrice
↳ .String()
68                  // No need to migrate as this function does not
↳ change the status of Send
69                  k.SetCrossChainTx(ctx, cctx)
70                  EmitCCTXScrubbed(ctx, cctx, chainID, oldGasPrice.
↳ String(), newGasPrice.String())
71              }
72          }
73      }
74 }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:C/Y:N/R:N/S:C (10)

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.

Remediation Plan:

SOLVED: The mechanism that multiplied the gas prices was removed by pull request 624.

4.3 (HAL-03) POSSIBLE DIVISION BY ZERO COULD CAUSE CHAIN HALT DUE TO PANIC - CRITICAL(10)

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

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

Listing 7: This function calls GetBondFactor

```
53 func GetBlockRewardComponents(ctx sdk.Context, bankKeeper types.
↳ BankKeeper, stakingKeeper types.StakingKeeper, emissionKeeper
↳ keeper.Keeper) (sdk.Dec, sdk.Dec, sdk.Dec) {
54     reservesFactor := GetReservesFactor(ctx, bankKeeper)
55     if reservesFactor.LTE(sdk.ZeroDec()) {
56         return sdk.ZeroDec(), sdk.ZeroDec(), sdk.ZeroDec()
57     }
```

```
58     bondFactor := GetBondFactor(ctx, stakingKeeper, emissionKeeper
↳ )
59     durationFactor := GetDurationFactor(ctx, emissionKeeper)
60     return reservesFactor, bondFactor, durationFactor
61 }
```

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

```
67 func GetBondFactor(ctx sdk.Context, stakingKeeper types.
↳ StakingKeeper, keeper keeper.Keeper) sdk.Dec {
68     targetBondRatio := sdk.MustNewDecFromStr(keeper.GetParams(ctx)
↳ .TargetBondRatio)
69     maxBondFactor := sdk.MustNewDecFromStr(keeper.GetParams(ctx).
↳ MaxBondFactor)
70     minBondFactor := sdk.MustNewDecFromStr(keeper.GetParams(ctx).
↳ MinBondFactor)
71
72     currentBondedRatio := stakingKeeper.BondedRatio(ctx)
73     // Bond factor ranges between minBondFactor (0.75) to
↳ maxBondFactor (1.25)
74     bondFactor := targetBondRatio.Quo(currentBondedRatio)
75     if bondFactor.GT(maxBondFactor) {
76         return maxBondFactor
77     }
78     if bondFactor.LT(minBondFactor) {
79         return minBondFactor
80     }
81     return bondFactor
82 }
```

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.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:C/Y:N/R:N/S:C (10)

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, it is recommended 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.

Remediation Plan:

RISK ACCEPTED: The ZetaChain team has accepted the risk of the issue.

4.4 (HAL-04) BITCOIN TRANSACTIONS REQUIRE ONLY ONE CONFIRMATION - CRITICAL(10)

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.

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

```
261     chain := ob.chain.ChainId
262     outTxID := fmt.Sprintf("%d-%d", chain, nonce)
263     ob.logger.Info().Msgf("IsSendOutTxProcessed %s", outTxID)
264
265     res, found := ob.submittedTx[outTxID]
266     if !found {
267         return false, false, nil
268     }
269     if res.Confirmations == 0 {
270         return true, false, nil
271     } else if res.Confirmations > 0 { // FIXME: use configured
 ↳ block confirmation
272         amountInSat, _ := big.NewFloat(res.Amount * 1e8).Int(nil)
273         zetaHash, err := ob.zetaClient.PostReceiveConfirmation(
274             sendHash,
275             res.TxID,
276             uint64(res.BlockIndex),
277             amountInSat,
278             common.ReceiveStatus_Success,
279             ob.chain,
280             nonce,
281             common.CoinType_Gas,
282         )
283         if err != nil {
284             ob.logger.Error().Err(err).Msgf("error posting to zeta
 ↳  core")
285         } else {
286             ob.logger.Info().Msgf("Bitcoin outTx confirmed:
 ↳ PostReceiveConfirmation zeta tx: %s", zetaHash)
287         }
288         return true, true, nil
289     }
290     return false, false, nil
291 }
```

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.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:C/D:C/Y:N/R:N/S:C (10)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by applying the recommended fix.

Commit ID : https://github.com/zeta-chain/node/pull/529/commits/4f8333b6677f9deea34938222bb287fcfb97440f

4.5 (HAL-05) LACK OF MECHANISM TO LIMIT SUPPLY OF ZETA - HIGH (7.8)

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.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:M/Y:N/R:N/S:C (7.8)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by limiting total supply.

Commit ID : 3f5d0782f73dee061d60bbde1106d25c1706a504

4.6 (HAL-06) PRICE MANIPULATION AND DENIAL-OF-SERVICE VIA UpdatePrices FUNCTION - HIGH (7.8)

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.

```
80 func (k Keeper) UpdatePrices(ctx sdk.Context, chainID int64, cctx
 ↳ *types.CrossChainTx) error {
81     chain := k.zetaObserverKeeper.GetParams(ctx).
 ↳ GetChainFromChainID(chainID)
82     medianGasPrice, isFound := k.GetMedianGasPriceInUint(ctx,
 ↳ chain.ChainId)
83     if !isFound {
84         return sdkerrors.Wrap(types.ErrUnableToGetGasPrice, fmt.
 ↳ Sprintf(" chain %d | Identifiers : %s ", cctx.GetCurrentOutTxParam
 ↳ ().ReceiverChainId, cctx.LogIdentifierForCCTX()))
85     }
86     cctx.GetCurrentOutTxParam().OutboundTxGasPrice =
 ↳ medianGasPrice.String()
87     gasLimit := sdk.NewUint(cctx.GetCurrentOutTxParam().
 ↳ OutboundTxGasLimit)
88     outTxGasFee := gasLimit.Mul(medianGasPrice)
89
90     // the following logic computes outbound tx gas fee, and
 ↳ convert into ZETA using system uniswapv2 pool wzeta/gasZRC20
91     gasZRC20, err := k.fungibleKeeper.
 ↳ QuerySystemContractGasCoinZRC4(ctx, big.NewInt(chain.ChainId))
92     if err != nil {
93         return sdkerrors.Wrap(err, "UpdatePrices: unable to get
 ↳ system contract gas coin")
94     }
95     outTxGasFeeInZeta, err := k.fungibleKeeper.
 ↳ QueryUniswapv2RouterGetAmountsIn(ctx, outTxGasFee.BigInt(),
 ↳ gasZRC20)
96     if err != nil {
97         return sdkerrors.Wrap(err, "UpdatePrices: unable to
 ↳ QueryUniswapv2RouterGetAmountsIn")
98     }
99     feeInZeta := types.GetProtocolFee().Add(math.NewUintFromBigInt
 ↳ (outTxGasFeeInZeta))
100
101    // swap the outTxGasFeeInZeta portion of zeta to the real gas
 ↳ ZRC20 and burn it
102    coins := sdk.NewCoins(sdk.NewCoin(config.BaseDenom, sdk.
 ↳ NewIntFromBigInt(feeInZeta.BigInt())))
103    err = k.bankKeeper.MintCoins(ctx, types.ModuleName, coins)
104    if err != nil {
105        return sdkerrors.Wrap(err, "UpdatePrices: unable to mint
```

```
 ↳ coins")
106     }
107     amounts, err := k.fungibleKeeper.
 ↳ CallUniswapv2RouterSwapExactETHForToken(ctx, types.
 ↳ ModuleAddressEVM, types.ModuleAddressEVM, outTxGasFeeInZeta,
 ↳ gasZRC20)
108     if err != nil {
109         return sdkerrors.Wrap(err, "UpdatePrices: unable to
 ↳ CallUniswapv2RouterSwapExactETHForToken")
110     }
111     ctx.Logger().Info("gas fee", "outTxGasFee", outTxGasFee, "
 ↳ outTxGasFeeInZeta", outTxGasFeeInZeta)
112     ctx.Logger().Info("CallUniswapv2RouterSwapExactETHForToken", "
 ↳ zetaAmountIn", amounts[0], "zrc20AmountOut", amounts[1])
113     err = k.fungibleKeeper.CallZRC20Burn(ctx, types.
 ↳ ModuleAddressEVM, gasZRC20, amounts[1])
114     if err != nil {
115         return sdkerrors.Wrap(err, "UpdatePrices: unable to
 ↳ CallZRC20Burn")
116     }
117
118     cctx.ZetaFees = cctx.ZetaFees.Add(feeInZeta)
119
120     if cctx.ZetaFees.GT(cctx.InboundTxParams.Amount) && cctx.
 ↳ InboundTxParams.CoinType == common.CoinType_Zeta {
121         return sdkerrors.Wrap(types.ErrNotEnoughZetaBurnt, fmt.
 ↳ Sprintf("feeInZeta(%s) more than zetaBurnt (%s) | Identifiers : %s
 ↳  ", cctx.ZetaFees, cctx.InboundTxParams.Amount, cctx.
 ↳ LogIdentifierForCCTX()))
122     }
123     cctx.GetCurrentOutTxParam().Amount = cctx.InboundTxParams.
 ↳ Amount.Sub(cctx.ZetaFees)
124
125     return nil
126 }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:M/Y:N/R:N/S:C (7.8)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by implementing necessity checks.

Commit ID : 3f5d0782f73dee061d60bbde1106d25c1706a504

4.7 (HAL-07) ERROR CONDITION FOR KEY SIGNING IS UNCHECKED - HIGH (7.8)

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

```
415 func TestKeysign(tssPubkey string, tssServer *tss.TssServer) error
 ↳  {
416     log.Info().Msg("trying keysign...")
417     data := []byte("hello meta")
418     H := crypto.Keccak256Hash(data)
419     log.Info().Msgf("hash of data (hello meta) is %s", H)
420
421     keysignReq := keysign.NewRequest(tssPubkey, []string{base64.
 ↳ StdEncoding.EncodeToString(H.Bytes())}, 10, nil, "0.14.0")
422     ksRes, err := tssServer.KeySign(keysignReq)
423     if err != nil {
424         log.Warn().Msg("keysign fail")
425     }
426     signature := ksRes.Signatures
427     // [{cyP8i/UuCVfQKDsLr1kpg09/CeIHje1FU6GhfmyMD5Q= D4jXTH3/
 ↳ CSgCg+9kLjhhfnNo3ggy9DTQSlloe3bbKAs= eY++
 ↳ Z2LwsuKG1JcghChrsEJ4u9grLloaaFZNtXI3Ujk= AA==}]
428     // 32B msg hash, 32B R, 32B S, 1B RC
429     log.Info().Msgf("signature of helloworld... %v", signature)
430
431     if len(signature) == 0 {
432         log.Info().Msgf("signature has length 0, skipping verify")
433         return fmt.Errorf("signature has length 0")
434     }
435     verifySignature(tssPubkey, signature, H.Bytes())
```

```
436     if verifySignature(tssPubkey, signature, H.Bytes()) {
437         return nil
438     }
439     return fmt.Errorf("verify signature fail")
440 }
```

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. ")
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:M/Y:N/R:N/S:C (7.8)

Recommendation:

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

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.8 (HAL-08) ITERATION OVER MAPS MAY BE A SOURCE OF NON-DETERMINISM - HIGH (7.8)

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.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:M/Y:N/R:N/S:C (7.8)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by implementing necessity checks.

Commit ID : 31b169aab1aaa3a6fd823ff1828a1dd5a3a037de

4.9 (HAL-09) SYBIL ATTACK RISK DUE TO USE OF MEDIAN GAS VOTES FOR SETTING GAS PRICE - HIGH (7.8)

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

```
119
120 func (k msgServer) GasPriceVoter(goCtx context.Context, msg *types
 ↳ .MsgGasPriceVoter) (*types.MsgGasPriceVoterResponse, error) {
121     ctx := sdk.UnwrapSDKContext(goCtx)
122
123     validators := k.StakingKeeper.GetAllValidators(ctx)
124     if !IsBondedValidator(msg.Creator, validators) {
125         return nil, sdkerrors.Wrap(sdkerrors.ErrorInvalidSigner,
 ↳ fmt.Sprintf("signer %s is not a bonded validator", msg.Creator))
126     }
127     chain := k.zetaObserverKeeper.GetParams(ctx).
 ↳ GetChainFromChainID(msg.ChainId)
128     if chain == nil {
129         return nil, sdkerrors.Wrap(types.ErrUnsupportedChain, fmt.
 ↳ Sprintf("ChainID : %d ", msg.ChainId))
130     }
131
132     gasPrice, isFound := k.GetGasPrice(ctx, chain.ChainId)
133     if !isFound {
134         gasPrice = types.GasPrice{
135             Creator:     msg.Creator,
136             Index:       strconv.FormatInt(chain.ChainId, 10), //
 ↳ TODO : Not needed index set at keeper
137             ChainId:     chain.ChainId,
138             Prices:      []uint64{msg.Price},
139             BlockNums:   []uint64{msg.BlockNumber},
140             Signers:     []string{msg.Creator},
141             MedianIndex: 0,
142         }
143     } else {
144         signers := gasPrice.Signers
145         exist := false
146         for i, s := range signers {
147             if s == msg.Creator { // update existing entry
148                 gasPrice.BlockNums[i] = msg.BlockNumber
149                 gasPrice.Prices[i] = msg.Price
150                 exist = true
151                 break
152             }
153         }
```

```
154         if !exist {
155             gasPrice.Signers = append(gasPrice.Signers, msg.
 ↳ Creator)
156             gasPrice.BlockNums = append(gasPrice.BlockNums, msg.
 ↳ BlockNumber)
157             gasPrice.Prices = append(gasPrice.Prices, msg.Price)
158         }
159         // recompute the median gas price
160         mi := medianOfArray(gasPrice.Prices)
161         gasPrice.MedianIndex = uint64(mi)
162     }
163     k.SetGasPrice(ctx, gasPrice)
164     chainIDBigINT := big.NewInt(chain.ChainId)
165     gasUsed, err := k.fungibleKeeper.SetGasPrice(ctx,
 ↳ chainIDBigINT, big.NewInt(int64(gasPrice.Prices[gasPrice.
 ↳ MedianIndex])))
166     if err != nil {
167         return nil, err
168     }
169
170     // reset the gas count
171     k.ResetGasMeterAndConsumeGas(ctx, gasUsed)
172
173     return &types.MsgGasPriceVoterResponse{}, nil
174 }
```

Listing 15

```
1 export PRICE="123456789"
2 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
```

Listing 16

```
1 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"
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:M/Y:N/R:N/S:C (7.8)

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.

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted risk the of the issue.

4.10 (HAL-10) MALICIOUS GAS PRICE VOTING: DENIAL-OF-SERVICE BY SETTING LARGE GAS PRICES FOR EVM NETWORKS - MEDIUM (5.5)

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.

BVSS:

AO:A/AC:L/AX:L/C:M/I:M/A:M/D:M/Y:N/R:P/S:C (5.5)

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, it is recommended using a TWAP mechanism rather than calculating the gas price based on the median value of posted prices.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by implementing suggested changes.

Commit ID : ee0ac1faf0c9d5fe97fd582d1be53dbf3d6c8926

4.11 (HAL-11) MALICIOUS GAS PRICE VOTING: DENIAL-OF-SERVICE OR PRICE MANIPULATION BY SETTING GAS PRICES FOR BITCOIN - MEDIUM (5.5)

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="92233720368547758O7"
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.

BVSS:

AO:A/AC:L/AX:L/C:M/I:M/A:M/D:M/Y:N/R:P/S:C (5.5)

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, it is recommended using a TWAP mechanism rather than calculating the gas price based on the median value of posted prices.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by implementing suggested changes.

Commit ID : ee0ac1faf0c9d5fe97fd582d1be53dbf3d6c8926

4.12 (HAL-12) INTEGER OVERFLOW BREAKS GRPC COMMUNICATION FOR LARGE BLOCK HEIGHTS - MEDIUM (5.5)

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

```
1 package main
2
3 import (
4     "context"
```

```
5     "fmt"
6     "math"
7     "strconv"
8     "strings"
9
10    "google.golang.org/grpc"
11
12    "github.com/cosmos/cosmos-sdk/codec"
13    //banktypes "github.com/cosmos/cosmos-sdk/x/bank/types"
14    crosschaintypes "github.com/zeta-chain/zetacore/x/crosschain/
↳ types"
15    //crosschaintypes "github.com/zeta-chain/zeta-node/x/
↳ crosschain/types"
16 )
17
18 func queryState(height uint64) (string, error) {
19    // Create a connection to the gRPC server.
20    grpcConn, err := grpc.Dial(
21        "127.0.0.1:9090",    // your gRPC server address.
22        grpc.WithInsecure(), // The Cosmos SDK doesn't support any
↳  transport security mechanism.
23        // This instantiates a general gRPC codec which handles
↳ proto bytes. We pass in a nil interface registry
24        // if the request/response types contain interface instead
↳  of 'nil' you should pass the application specific codec.
25        grpc.WithDefaultCallOptions(grpc.ForceCodec(codec.
↳ NewProtoCodec(nil).GRPCCodec())),
26    )
27    if err != nil {
28        return "", err
29    }
30    defer grpcConn.Close()
31
32    // This creates a gRPC client to query the service.
33    crosschainClient := crosschaintypes.NewQueryClient(grpcConn)
34    res, err := crosschainClient.ZEVMGetBlock(
35        context.Background(),
36        //&crosschaintypes.QueryZEVMGetBlockRequest{Height: height
↳ },
37        &crosschaintypes.QueryZEVMGetBlockByNumberRequest{Height:
↳ height},
38    )
39    if err != nil {
40        return "", err
```

```
41      }
42
43      return fmt.Sprintf("Block Number: %s\nBlock Hash: %s\n", res.
↳ Number, res.Hash), nil
44  }
45
46  func main() {
47      var latestBlock uint64 = 0
48      fmt.Printf("Querying the max int64 value (%s).\nThis will
↳ cause an error\n", uint(math.MaxInt64))
49      res, err := queryState(uint64(math.MaxInt64))
50      if err != nil {
51          // We expect to get here
52          fmt.Printf("Error occurred in call 1: %s\n", err)
53
54          // Extract the latest block number. It will be the last
↳ value in the error string
55          words := strings.Split(err.Error(), " ")
56          latestBlock, err = strconv.ParseUint(words[len(words)-1],
↳ 10, 64)
57          if err != nil {
58              panic("Could not parse block from error string")
59          }
60      }
61      fmt.Println(res)
62
63      fmt.Printf("Querying the max int64 value + 1 (%s). This will
↳ overflow and get the latest block\n", uint64(math.MaxInt64+1))
64      res, err = queryState(uint64(math.MaxInt64) + 1)
65      if err != nil {
66          fmt.Printf("Error occurred in call 2: %s\n", err)
67      }
68      fmt.Println(res)
69
70      if latestBlock == 0 {
71          panic("latestBlock is (still) 0. This should not happen.")
72      }
73
74      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)
75      res, err = queryState(latestBlock)
76      if err != nil {
```

```
77          fmt.Printf("Error occurred in call 3: %s\n", err)
78      }
79      fmt.Printf(res)
80  }
```

Listing 21: Output demonstrates overflow

```
1  Querying the max int64 value (%!s(uint=9223372036854775807)).
2  This will cause an error
3  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
4
5  Querying the max int64 value + 1 (%!s(uint64=9223372036854775808))
↳  . This will overflow and get the latest block
6  Block Number: 0x8000000000000000
7  Block Hash: 0
↳  x141e0e146ea4ecc707918883cd8b9fd1fe775dd10191fc08586787baea441c97
8
9  Querying the latest block as given in first request(%!s(uint64
↳  =13698)).
10 This will match the results from the previous request. This
↳  demonstrates that when the code overflows, it returns the latest
↳  block
11 Block Number: 0x3582
12 Block Hash: 0
↳  x141e0e146ea4ecc707918883cd8b9fd1fe775dd10191fc08586787baea441c97
```

BVSS:

AO:A/AC:L/AX:L/C:M/I:M/A:M/D:M/Y:N/R:P/S:C (5.5)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by implementing necessity checks.

Commit IDs :

- a5aa77681ced5ea20cacdcfb6bbd5bdb385bdd2c
- 37de781c228c08fa79ef3c1ce45da9d8a685e2a7

4.13 (HAL-13) RELIANCE ON UNISWAPV2 POOLS FOR PRICES EXPOSES ZETACHAIN TO PRICE MANIPULATION RISK - MEDIUM (5.5)

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.

BVSS:

AO:A/AC:L/AX:L/C:M/I:M/A:M/D:M/Y:N/R:P/S:C (5.5)

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).

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.14 (HAL-14) ARBITRARY MINTING OF ZETA VIA MintZetaToEVMAccount FUNCTION - MEDIUM (5.5)

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.

BVSS:

AO:A/AC:L/AX:L/C:M/I:M/A:M/D:M/Y:N/R:P/S:C (5.5)

Recommendation:

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

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by implementing suggested changes.

Commit ID : 490f57435bd4fd1bfbc4a55ebece4b93376c2d76

4.15 (HAL-15) SUPPORT FOR A TOKEN CANNOT BE REMOVED FROM THE PROTOCOL - MEDIUM (5.5)

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)
```

```
10      //
11      //if msg.Creator != types.AdminAddress {
12      //  return nil, sdkerrors.Wrap(sdkerrors.ErrUnauthorized, "
↳ only admin can remove foreign coin")
13      //}
14      //index := msg.Name
15
16      //_, found := k.GetForeignCoins(ctx, index)
17      //if !found {
18      //  return nil, sdkerrors.Wrapf(sdkerrors.ErrInvalidRequest, "
↳ foreign coin not found")
19      //}
20      //k.RemoveForeignCoins(ctx, index)
21      // TODO : FIX THIS MSG
22      return &types.MsgRemoveForeignCoinResponse{}, nil
23  }
```

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  //}
```

BVSS:

AO:A/AC:L/AX:L/C:M/I:M/A:M/D:M/Y:N/R:P/S:C (5.5)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by implementing suggested change.

4.16 (HAL-16) USE OF VULNERABLE COSMOSSDK VERSION - LOW (2.3)

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.

```
1 module github.com/zeta-chain/zetacore
2
3 go 1.19
4
5 require (
6     github.com/cosmos/cosmos-sdk v0.46.8
```

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

Recommendation:

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

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by upgrading the Cosmos SDK version.

Commit ID : 007ff12dacf5142933bdce06ad2da724a992bd6e

4.17 (HAL-17) ValidateBasic INCOMPLETE FOR SOME MESSAGE TYPES - LOW (2.3)

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

```
322 type MsgCreateTSSVoter struct {
323     Creator string `protobuf:"bytes,1,opt,name=creator,proto3"
 ↳ json:"creator,omitempty"`
324     Chain   string `protobuf:"bytes,3,opt,name=chain,proto3" json
 ↳ :"chain,omitempty"`
325     Address string `protobuf:"bytes,4,opt,name=address,proto3"
 ↳ json:"address,omitempty"`
326     Pubkey  string `protobuf:"bytes,5,opt,name=pubkey,proto3" json
 ↳ :"pubkey,omitempty"`
327 }
```

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

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

```
44          return sdkerrors.Wrapf(sdkerrors.ErrInvalidAddress, "
↳ invalid creator address (%s)", err)
45      }
46      return nil
47  }
```

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by using the ValidateBasic function.

4.18 (HAL-18) CENTRALIZATION RISK - LOW (2.3)

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",
        },
        {
```

```
18              PolicyType: Policy_Type_deploy_fungible_coin,
19              Address:    "
↳ zeta1afk9zr2hn2jsac63h4hm60vl9z3e5u69gndzf7c99cqge3vzwjzsxn0x73",
20          },
21      }
22      return NewParams(observerParams, adminPolicy)
23  }
```

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

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.

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.19 (HAL-19) LACK OF UNIT TESTS - LOW (2.3)

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)

```
11 // Mint ZETA (gas token) to the given address
12 // TODO balanceCoinAfter != expCoin , be replicated in an unit
↳ test
13 func (k *Keeper) MintZetaToEVMAccount(ctx sdk.Context, to sdk.
↳ AccAddress, amount *big.Int) error {
14     balanceCoin := k.bankKeeper.GetBalance(ctx, to, config.
↳ BaseDenom)
15     coins := sdk.NewCoins(sdk.NewCoin(config.BaseDenom, sdk.
↳ NewIntFromBigInt(amount)))
16     // Mint coins
17     if err := k.bankKeeper.MintCoins(ctx, types.ModuleName, coins)
↳ ; err != nil {
18         return err
19     }
20
21     // Send minted coins to the receiver
22     if err := k.bankKeeper.SendCoinsFromModuleToAccount(ctx, types
```

```
↳ .ModuleName, to, coins); err != nil {
23          return err
24      }
25
26      // Check expected receiver balance after transfer
27      balanceCoinAfter := k.bankKeeper.GetBalance(ctx, to, config.
↳ BaseDenom)
28      expCoin := balanceCoin.Add(coins[0])
29
30      if ok := balanceCoinAfter.IsEqual(expCoin); !ok {
31          return sdkerrors.Wrapf(
32              types.ErrBalanceInvariance,
33              "invalid coin balance - expected: %v, actual: %v",
34              expCoin, balanceCoinAfter,
35          )
36      }
37
38      return nil
39 }
```

Example 2: Error-checking is absent in the 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

```
1  func TestKeeper_GetBallot(t *testing.T) {
2      k, ctx := SetupKeeper(t)
3      identifier := "0
↳ x9ea007f0f60e32d58577a8cf25678942d2b10791c2a34f48e237b76a7e998e4d"
4      k.SetBallot(ctx, &types.Ballot{
5          Index:            "",
6          BallotIdentifier: identifier,
7          VoterList:        nil,
8          ObservationType:  0,
9          BallotThreshold:  sdk.Dec{},
10         BallotStatus:     0,
11     })
12
13     k.GetBallot(ctx, identifier)
14 }
```

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.

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

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.

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.20 (HAL-20) LACK OF FUZZ TESTS - LOW (2.3)

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.

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

Recommendation:

For a project providing bridging functionality, fuzz testing should be considered a requirement. It is recommended 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.

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.21 (HAL-21) BITCOIN TRANSACTIONS WITH LARGE NONCE VALUES MAY BE SENT REPEATEDLY - LOW (2.3)

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

```
211 chain := GetTargetChain(send)
212 nonce := send.GetCurrentOutTxParam().OutboundTxTssNonce
213 outTxID := fmt.Sprintf("%s-%d-%d", send.Index, send.
 ↳ GetCurrentOutTxParam().ReceiverChainId, nonce) // should be the
 ↳ outTxID?
214
215 // FIXME: config this schedule; this value is for localnet fast
 ↳ testing
216 if nonce%1 == bn%1 && !outTxMan.IsOutTxActive(outTxID) {
217     outTxMan.StartTryProcess(outTxID)
218     fmt.Printf("chain %s: Sign outtx %s with value %d\n", chain,
 ↳ send.Index, send.GetCurrentOutTxParam().Amount)
219     go signer.TryProcessOutTx(send, outTxMan, outTxID, chainClient
 ↳ , co.bridge)
220 }
```

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

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

```
 ↳ decode to address err %v", send.Index, err)
181         return
182     }
183     fmt.Printf("BTC TryProcessOutTx: %s, value %d to %s\n", send.
 ↳ Index, send.GetCurrentOutTxParam().Amount.BigInt(), toAddr)
184     defer func() {
185         outTxMan.EndTryProcess(outTxID)
186     }()
187     btcClient, ok := chainclient.(*BitcoinChainClient)
188     if !ok {
189         signer.logger.Error().Msgf("chain client is not a bitcoin
 ↳ client")
190         return
191     }
192
193     logger := signer.logger.With().
194         Str("sendHash", send.Index).
195         Logger()
196
197     myid := zetaBridge.keys.GetAddress().String()
198
199     // Early return if the send is already processed
200     // FIXME: handle revert case
201     included, confirmed, _ := btcClient.IsSendOutTxProcessed(send.
 ↳ Index, int(send.GetCurrentOutTxParam().OutboundTxTssNonce), common
 ↳ .CoinType_Gas)
202     if included || confirmed {
203         logger.Info().Msgf("CCTX already processed; exit signer")
204         return
205     }
```

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

```
260 func (ob *BitcoinChainClient) IsSendOutTxProcessed(sendHash string
 ↳ , nonce int, _ common.CoinType) (bool, bool, error) {
261     chain := ob.chain.ChainId
262     outTxID := fmt.Sprintf("%d-%d", chain, nonce)
263     ob.logger.Info().Msgf("IsSendOutTxProcessed %s", outTxID)
264
```

```
265     res, found := ob.submittedTx[outTxID]
266     if !found {
267         return false, false, nil
268     }
```

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

Listing 33

```
568 func (ob *BitcoinChainClient) observeOutTx() {
569     ticker := time.NewTicker(2 * time.Second)
570     for {
571         select {
572         case <-ticker.C:
573             trackers, err := ob.zetaClient.
 ↳ GetAllOutTxTrackerByChain(ob.chain)
574             if err != nil {
575                 ob.logger.Error().Err(err).Msg("error
 ↳ GetAllOutTxTrackerByChain")
576                 continue
577             }
578             for _, tracker := range trackers {
579                 outTxID := fmt.Sprintf("%d-%d", tracker.ChainId,
 ↳ tracker.Nonce)
580                 ob.logger.Info().Msgf("tracker outTxID: %s",
 ↳ outTxID)
581                 for _, txHash := range tracker.HashList {
582                     hash, err := chainhash.NewHashFromStr(txHash.
 ↳ TxHash)
583                     if err != nil {
584                         ob.logger.Error().Err(err).Msg("error
 ↳ NewHashFromStr")
585                         continue
586                     }
587                     getTxResult, err := ob.rpcClient.
 ↳ GetTransaction(hash)
588                     if err != nil {
589                         ob.logger.Warn().Err(err).Msg("error
 ↳ GetTransaction")
590                         continue
591                     }
```

```
592                     if getTxResult.Confirmations >= 0 {
593                         ob.submittedTx[outTxID] = *getTxResult
594                     }
595                 }
596             }
597         case <-ob.stop:
598             ob.logger.Info().Msg("observeOutTx stopped")
599             return
600         }
601     }
602 }
```

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.

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

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.

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.22 (HAL-22) UNBOUNDED ARRAY IN Signers IN ChainNonces COULD CAUSE RESOURCE EXHAUSTION - LOW (2.3)

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.

```
103 func (k msgServer) NonceVoter(goCtx context.Context, msg *types.
 ↳ MsgNonceVoter) (*types.MsgNonceVoterResponse, error) {
104     ctx := sdk.UnwrapSDKContext(goCtx)
105
106     validators := k.StakingKeeper.GetAllValidators(ctx)
107     if !IsBondedValidator(msg.Creator, validators) {
108         return nil, sdkerrors.Wrap(sdkerrors.ErrorInvalidSigner,
 ↳ fmt.Sprintf("signer %s is not a bonded validator", msg.Creator))
109     }
110
```

```
111     chain := msg.Chain
112     chainNonce, isFound := k.GetChainNonces(ctx, chain)
113     //if isDuplicateSigner(msg.Creator, chainNonce.Signers) {
114     //  return nil, sdkerrors.Wrap(sdkerrors.ErrorInvalidSigner,
 ↳ fmt.Sprintf("signer %s double signing!!", msg.Creator))
115     //}
116     if isFound {
117         chainNonce.Signers = append(chainNonce.Signers, msg.
 ↳ Creator)
118         chainNonce.Nonce = msg.Nonce
119     } else if !isFound {
120         chainNonce = types.ChainNonces{
121             Creator: msg.Creator,
122             Index:   msg.Chain,
123             Chain:   msg.Chain,
124             Nonce:   msg.Nonce,
125             Signers: []string{msg.Creator},
126         }
127     } else {
128         return nil, sdkerrors.Wrap(sdkerrors.ErrInvalidRequest,
 ↳ fmt.Sprintf("chainNonce vote msg does not match state: %v vs %v",
 ↳ msg, chainNonce))
129     }
130
131     //if hasSuperMajorityValidators(len(chainNonce.Signers),
 ↳ validators) {
132     //  chainNonce.FinalizedHeight = uint64(ctx.BlockHeader().
 ↳ Height)
133     //}
134
135     k.SetChainNonces(ctx, chainNonce)
136     return &types.MsgNonceVoterResponse{}, nil
137 }
```

```
> 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

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by implementing necessity checks.

4.23 (HAL-23) USE OF VULNERABLE DEPENDENCIES - LOW (2.3)

Description:

In addition to the specific issues with the Cosmos SDK 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.
...
```

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by updating vulnerable dependencies.

Commit ID : 007ff12dacf5142933bdce06ad2da724a992bd6e

4.24 (HAL-24) MNEMONIC PHRASES PRESENT IN CODEBASE - LOW (2.3)

Description:

Bash scripts in the repository contain hard-coded mnemonic phrases corresponding to Cosmos wallets. In general, it is recommended 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

```
1 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"
2 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"
3 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"
4 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"
5 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"
6 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"
7 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"
8 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"
 9 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"
10 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"
11 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"
```

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

Recommendation:

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

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.25 (HAL-25) VULNERABLE TSS-LIB CONTAINS HASH COLLISION ISSUES - LOW (2.3)

Description:

During the assessment, 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

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

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

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.

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.26 (HAL-26) CALCULATION ERRORS IN getSatoshis FUNCTION FOR EXTREMELY SMALL OR LARGE FLOAT VALUES - LOW (2.3)

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

```
16 func getSatoshis(btc float64) (int64, error) {
17     // The amount is only considered invalid if it cannot be
↳ represented
18     // as an integer type.  This may happen if f is NaN or +-
↳ Infinity.
19     switch {
20     case math.IsNaN(btc):
21         fallthrough
22     case math.IsInf(btc, 1):
23         fallthrough
24     case math.IsInf(btc, -1):
25         return 0, errors.New("invalid bitcoin amount")
```

```
26      }
27      return round(btc * satoshiPerBitcoin), nil
28  }
```

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by modifying the function.

Commit ID : 1e1e73024d21716cce1b39b401027690c4a13083

4.27 (HAL-27) GAS LIMITS CANNOT BE CONFIGURED - LOW (2.3)

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

```
19 func (k Keeper) setupChainGasCoinAndPool(ctx sdk.Context, c string
↳ , gasAssetName string, symbol string, decimals uint8) (ethcommon.
↳ Address, error) {
20     name := fmt.Sprintf("%s-%s", gasAssetName, c)
21     chainName := common.ParseChainName(c)
22     chain := k.zetaobserverKeeper.GetParams(ctx).
↳ GetChainFromChainName(chainName)
23     if chain == nil {
24         return ethcommon.Address{}, zetaObserverTypes.
↳ ErrSupportedChains
25     }
26
27     transferGasLimit := big.NewInt(21_000)
28     if chain.IsEVMChain() {
29         transferGasLimit = big.NewInt(21_000)
30     } else if chain.IsBitcoinChain() {
31         transferGasLimit = big.NewInt(100) // 100B for a typical
↳ tx
32     }
```

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

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.

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.28 (HAL-28) VALIDATORS CANNOT EFFECTIVELY SET THE ZETA GAS PRICE FOR BITCOIN TRANSACTIONS - LOW (2.3)

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

```
func (ob *BitcoinChainClient) WatchGasPrice() {
    gasTicker := time.NewTicker(time.Duration(config.BitcoinConfig
 ↳ .WatchGasPricePeriod) * time.Second)
    for {
        select {
        case <-gasTicker.C:
            err := ob.PostGasPrice()
            if err != nil {
                ob.logger.Error().Err(err).Msg("PostGasPrice error
 ↳  on " + ob.chain.String())
                continue
            }
        case <-ob.stop:
            ob.logger.Info().Msg("WatchGasPrice stopped")
            return
        }
    }
}
```

zetaclient/bitcoin_client.go Lines 322-356

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

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

```
328         _, err = ob.zetaClient.PostGasPrice(ob.chain, 1000, "100",
 ↳  uint64(bn))
329         if err != nil {
330             ob.logger.Err(err).Msg("PostGasPrice:")
331             return err
332         }
333         return nil
334     }
335     // EstimateSmartFee returns the fees per kilobyte (BTC/kb)
 ↳ targeting given block confirmation
336     feeResult, err := ob.rpcClient.EstimateSmartFee(1, &btcjson.
 ↳ EstimateModeConservative)
337     if err != nil {
338         return err
339     }
340     if feeResult.Errors != nil || feeResult.FeeRate == nil {
341         return fmt.Errorf("error getting gas price: %s", feeResult
 ↳ .Errors)
342     }
343     gasPrice := big.NewFloat(0)
344     gasPriceU64, _ := gasPrice.Mul(big.NewFloat(*feeResult.FeeRate
 ↳ ), big.NewFloat(1e8)).Uint64()
345     bn, err := ob.rpcClient.GetBlockCount()
346     if err != nil {
347         return err
348     }
349     _, err = ob.zetaClient.PostGasPrice(ob.chain, gasPriceU64, "
 ↳ 100", uint64(bn))
350     if err != nil {
351         ob.logger.Err(err).Msg("PostGasPrice:")
352         return err
353     }
354     _ = feeResult
355     return nil
356 }
```

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

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.

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.

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.29 (HAL-29) GO VERSIONS PRIOR TO 1.20.2 CONTAIN CRYPTOGRAPHIC ISSUES AND OTHER BUGS - LOW (2.3)

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

```
1 module github.com/zeta-chain/zetacore
2
3 go 1.19
```

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

Recommendation:

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

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.30 (HAL-30) ZETACLIENT AND ZETACORE TRACK BLOCK HEIGHT USING DIFFERENT TYPES - LOW (2.3)

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

```
37 // ZetaCoreBridge will be used to send tx to ZetaCore.
38 type ZetaCoreBridge struct {
39     logger              zerolog.Logger
40     blockHeight         int64
41     accountNumber       uint64
42     seqNumber           uint64
43     grpcConn            *grpc.ClientConn
44     httpClient          *retryablehttp.Client
45     cfg                 config.ClientConfiguration
```

```
46     keys                *Keys
47     broadcastLock       *sync.RWMutex
48     ChainNonces         map[string]uint64 // FIXME: Remove this?
49     lastOutTxReportTime map[string]time.Time
50 }
```

zetaclient/broadcast.go

Listing 47

```
20 // Broadcast Broadcasts tx to metachain. Returns txHash and error
21 func (b *ZetaCoreBridge) Broadcast(gaslimit uint64, msgs ...stypes
↳ .Msg) (string, error) {
22     b.broadcastLock.Lock()
23     defer b.broadcastLock.Unlock()
24     var err error
25     blockHeight, err := b.GetZetaBlockHeight()
26     if err != nil {
27         return "", err
28     }
29
30     if int64(blockHeight) > b.blockHeight {
31         b.blockHeight = int64(blockHeight)
32         accountNumber, seqNumber, err := b.
↳ GetAccountNumberAndSequenceNumber()
33         if err != nil {
34             return "", err
35         }
36         b.accountNumber = accountNumber
37         if b.seqNumber < seqNumber {
38             b.seqNumber = seqNumber
39         }
40     }
```

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.

BVSS:

AO:A/AC:L/AX:L/C:L/I:L/A:M/D:M/Y:N/R:F/S:C (2.3)

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.

Remediation Plan:

RISK ACCEPTED: The ZetaChain team accepted the risk of this issue.

4.31 (HAL-31) TYPE CONVERSION ISSUE FOR FIELD Decimals ON STRUCT MsgDeployFungibleCoinZRC20 - INFORMATIONAL (0.0)

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 {
```

```
20          _, err := k.setupChainGasCoinAndPool(ctx, msg.ForeignChain
↳ , msg.Name, msg.Symbol, uint8(msg.Decimals))
21          if err != nil {
22              return nil, sdkerrors.Wrapf(err, "failed to
↳ setupChainGasCoinAndPool")
23          }
24      } else {
25          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          }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation:

It is recommended 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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by changing data type.

4.32 (HAL-32) USE OF DEPRECATED GO VERSION - INFORMATIONAL (0.0)

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

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

contrib/localnet/orchestrator/Dockerfile

Listing 51

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

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation:

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

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by updating the go version.

4.33 (HAL-33) UNUSED FIELD GasLimit ON STRUCT MsgDeployFungibleCoinZRC20 - INFORMATIONAL (0.0)

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 {
```

```
33                  return err
34              }
35
36              clientCtx, err := client.GetClientTxContext(cmd)
37              if err != nil {
38                  return err
39              }
40              fmt.Printf("CLI address: %s\n", clientCtx.
↳ GetFromAddress().String())
41              msg := types.NewMsgDeployFungibleCoinZRC20(
42                  clientCtx.GetFromAddress().String(),
43                  argERC20,
44                  argForeignChain,
45                  uint32(argDecimals),
46                  argName,
47                  argSymbol,
48                  common.CoinType(argCoinType),
49              )
```

x/fungible/keeper/msg_server_deploy_fungible_coin_zrc_4.go

Listing 53

```
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 {
20         _, err := k.setupChainGasCoinAndPool(ctx, msg.ForeignChain
↳ , msg.Name, msg.Symbol, uint8(msg.Decimals))
21         if err != nil {
22             return nil, sdkerrors.Wrapf(err, "failed to
↳ setupChainGasCoinAndPool")
23         }
24     } else {
25         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          }
```

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.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by implementing suggestion.

4.34 (HAL-34) INTEGER OVERFLOW CONVERSION FOR GAS PRICE IN BITCOIN SIGNER - INFORMATIONAL (0.0)

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, it is recommended 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.

```
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)
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by changing data type.

4.35 (HAL-35) REFERENCE TO DEPRECATED ETHEREUM NETWORK - INFORMATIONAL (0.0)

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"
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation:

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

Remediation Plan:

ACKNOWLEDGED: The ZetaChain team acknowledged the issue.

4.36 (HAL-36) MESSAGE QueryBallotByIdentifierRequest RETURNS RESULTS FOR BALLOTS THAT DO NOT EXIST - INFORMATIONAL (0.0)

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.

```
43 func (k Keeper) BallotByIdentifier(goCtx context.Context, req *
↳ types.QueryBallotByIdentifierRequest) (*types.
↳ QueryBallotByIdentifierResponse, error) {
44     if req == nil {
45         return nil, status.Error(codes.InvalidArgument, "invalid
↳ request")
46     }
47     ctx := sdk.UnwrapSDKContext(goCtx)
48     voter, _ := k.GetBallot(ctx, req.BallotIdentifier)
49     return &types.QueryBallotByIdentifierResponse{Ballot: &voter},
↳  nil
50 }
```

```
> zetacored q observer show-ballot 0
ballot:
  BallotThreshold: "0.00000000000000000"
  ballot_identifier: ""
  ballot_status: BallotFinalized_SuccessObservation
  index: ""
  observation_type: EmptyObserverType
  voter_list: []
  votes: []
> zetacored q observer show-ballot 999999999999999999999999
ballot:
  BallotThreshold: "0.00000000000000000"
  ballot_identifier: ""
  ballot_status: BallotFinalized_SuccessObservation
  index: ""
  observation_type: EmptyObserverType
  voter_list: []
  votes: []
> zetacored q observer show-ballot 6c80546cdd972567769fbc747ecea44f528edba020ab36170013237a401cdb8f
ballot:
  BallotThreshold: "0.00000000000000000"
  ballot_identifier: ""
  ballot_status: BallotFinalized_SuccessObservation
  index: ""
  observation_type: EmptyObserverType
  voter_list: []
  votes: []
```

Figure 2: Results when querying Ballots that do not exist

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

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.

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by implementing suggested change.

4.37 (HAL-37) DOCKER FILES USES A DIFFERENT GO VERSION THAN THE PROJECT - INFORMATIONAL (0.0)

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

```
1 module github.com/zeta-chain/zetacore
2
3 go 1.19
```

Dockerfile

Listing 58

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

contrib/localnet/orchestrator/Dockerfile

Listing 59

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

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation:

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

Remediation Plan:

ACKNOWLEDGED: The ZetaChain team acknowledged the issue.

4.38 (HAL-38) UPGRADING TO A MORE RECENT VERSION OF CosmosSDK COULD INCREASE PERFORMANCE - INFORMATIONAL (0.0)

Description:

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

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

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.

Remediation Plan:

ACKNOWLEDGED: The ZetaChain team acknowledged the issue.

4.39 (HAL-39) TESTING ENVIRONMENT IS USING OUTDATED BITCOIN DAEMON - INFORMATIONAL (0.0)

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

```
80 ...
81   bitcoin:
82     image: ruimarinho/bitcoin-core:22 # version 23 is not working
↳ with btcd 0.22.0 due to change in createwallet rpc
83     container_name: bitcoin
84     hostname: bitcoin
85     ...
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

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.

Remediation Plan:

ACKNOWLEDGED: The ZetaChain team acknowledged the issue.

4.40 (HAL-40) GAS PRICE VOTER CLI INTERFACE USES THE WRONG TYPE - INFORMATIONAL (0.0)

Description:

The command-line interface for gas-price-voter parses the price parameter to an Int value. However, it is defined as a Uint in MsgGasPriceVoter. Later, the value is converted to a uint64 type. However, the use of ParseInt (instead of ParseUint) means that the code will throw an error when a gas price higher than MAX_INT64 is submitted as the value for 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

```
77 func CmdGasPriceVoter() *cobra.Command {
78     cmd := &cobra.Command{
79         Use:   "gas-price-voter [chain] [price] [supply] [
↳ blockNumber]",
80         Short: "Broadcast message gasPriceVoter",
81         Args:  cobra.ExactArgs(4),
82         RunE: func(cmd *cobra.Command, args []string) error {
83             argsChain, err := strconv.ParseInt(args[0], 10, 64)
84             if err != nil {
85                 return err
86             }
87             argsPrice, err := strconv.ParseInt(args[1], 10, 64)
88             if err != nil {
89                 return err
90             }
91             argsSupply := args[2]
92
93             argsBlockNumber, err := strconv.ParseInt(args[3], 10,
↳ 64)
94             if err != nil {
```

```
95                  return err
96              }
97              clientCtx, err := client.GetClientTxContext(cmd)
98              if err != nil {
99                  return err
100             }
101
102             msg := types.NewMsgGasPriceVoter(clientCtx.
 ↳ GetFromAddress().String(), argsChain, uint64(argsPrice),
 ↳ argsSupply, uint64(argsBlockNumber))
103             if err := msg.ValidateBasic(); err != nil {
104                 return err
105             }
106             return tx.GenerateOrBroadcastTxCLI(clientCtx, cmd.
 ↳ Flags(), msg)
107         },
108     }
109
110     flags.AddTxFlagsToCmd(cmd)
111
112     return cmd
113 }
```

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation:

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

Remediation Plan:

SOLVED: The ZetaChain team solved the issue by using ParseUint.

Commit ID : a3d936b7856ab2b77e5486de6b4c0177e6e90f23

4.41 (HAL-41) DOCKER IGNORE FILE SHOULD INCLUDE GIT FILES - INFORMATIONAL (0.0)

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.

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation:

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

Remediation Plan:

ACKNOWLEDGED: The ZetaChain team acknowledged the issue.

4.42 (HAL-42) TODOS IN CODEBASE - INFORMATIONAL (0.0)

Description:

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

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

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.

Remediation Plan:

ACKNOWLEDGED: The ZetaChain team acknowledged the issue.

4.43 (HAL-43) SPELLING MISTAKES IN CODE BASE - INFORMATIONAL (0.0)

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

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

zetaclient/bitcoin_client.go L358, various mistakes:

Listing 63: Spelling mistakes

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

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation:

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

Remediation Plan:

ACKNOWLEDGED: The ZetaChain team acknowledged the issue.

4.44 (HAL-44) INCORRECT CODE COMMENTS - INFORMATIONAL (0.0)

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).

BVSS:

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:F/S:C (0.0)

Recommendation:

Fix the comments to reflect the code.

Remediation Plan:

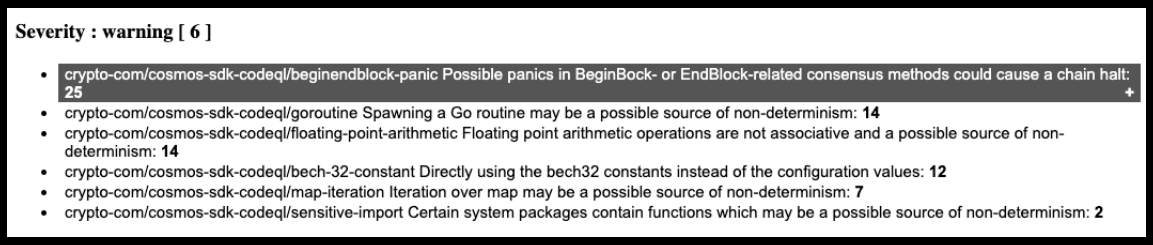
ACKNOWLEDGED: The ZetaChain team acknowledged the issue.

AUTOMATED TESTING

5.1 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.

5.2 codeql



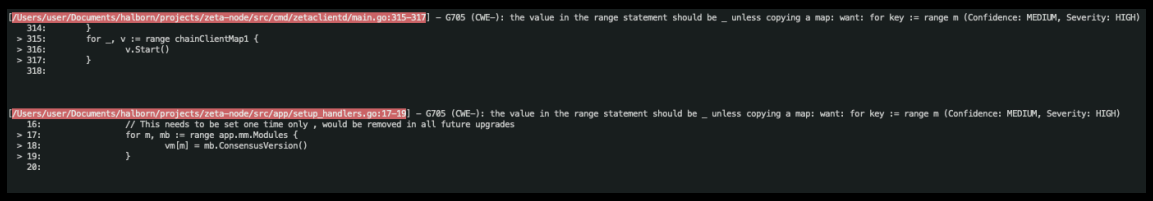
Severity : warning [6]

- 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

5.3 gosec

The following as an excerpt from running the tool gosec:



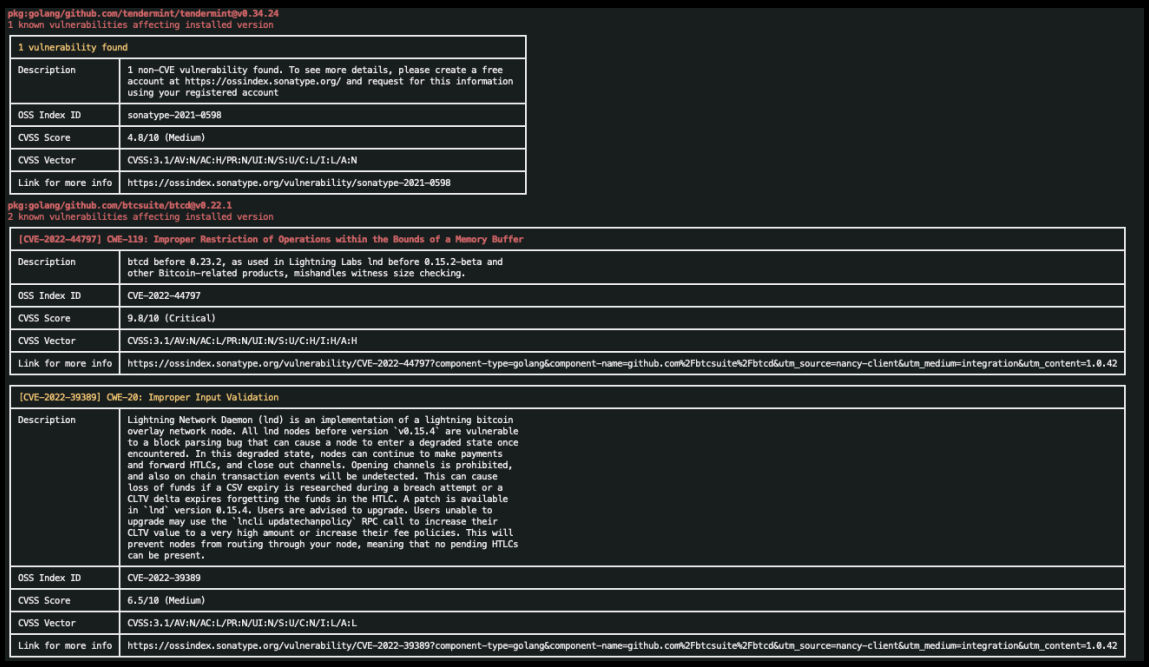
```
[/Users/user/Documents/halborn/projects/zeta-node/src/cmd/zetaclientd/main.go:315-317] - G705 (CWE-): the value in the range statement should be _ unless copying a map: want: for key := range m (Confidence: MEDIUM, Severity: HIGH)
   314:         }
 > 315:         for _, v := range chainClientMap1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

5.4 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

5.5 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

```
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