Zellic

ZetaChain

Smart Contract Security Assessment

June 30, 2023

Prepared for:

Tad Tobar

ZetaChain

Prepared by:

William Bowling and Syed Faraz Abrar

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for ZetaChain from March 15th to April 21st, 2023. During this engagement, Zellic reviewed ZetaChain's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Can a malicious attacker use the cross-chain communication logic to steal funds or disrupt the zEVM's operations?
- Can the cross-chain communication logic be exploited to mint free ZETA tokens?
- Is there a risk of malicious validators double signing their votes to manipulate the vote threshold and cause unintended outcomes?

1.2 Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.3 Results

During our assessment on the scoped ZetaChain contracts, we discovered 13 findings. Of the 13 findings, seven were of critical impact, four were of high impact, one was of medium impact, and the remaining finding was informational in nature.

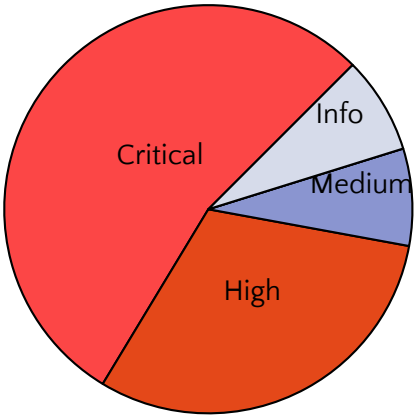
Additionally, Zellic recorded its notes and observations from the assessment for ZetaChain's benefit in the Discussion section (4) at the end of the document.

Based on our assessment, it is our opinion that the project is not yet ready for production, and we recommend a comprehensive re-audit before deployment. Our conclu-

sion is based on the significant number of bugs identified and the changes required to bring the codebase up to the necessary level of quality and security. We advise conducting additional testing and remediation before proceeding with any production release.

Breakdown of Finding Impacts

Impact Level	Count
Critical	7
High	4
Medium	1
Low	0
Informational	1



2 Introduction

2.1 About ZetaChain

ZetaChain is the foundational, public blockchain that enables omnichain, generic smart contracts and messaging between any blockchain. It solves the problems of cross-chain and multichain and aims to open the crypto and global financial ecosystem to anyone. ZetaChain envisions and supports a truly fluid, multichain crypto ecosystem, where users and developers can move between and appreciate the benefits of any blockchain: payments, DeFi, liquidity, games, art, social graphs, performance, security, privacy, and so on.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

ZetaChain Modules

Repository	https://github.com/zeta-chain/zeta-node
Version	zeta-node: `62dcc22756e3233ed611b058ca2c3c14319757c9`
Programs	• Zetanode • Zetaclient
Type	Cosmos
Platform	Cosmos-SDK

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of eight person-weeks. The assessment was conducted over the course of five calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

William Bowling, Engineer
vakzz@zellic.io

Syed Faraz Abrar, Engineer
faith@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

March 13, 2023	Kick-off call
March 15, 2023	Start of primary review period
April 21, 2023	End of primary review period
April 21, 2023	Draft report delivered

3 Detailed Findings

3.1 Any `ZetaSent` events are processed regardless of what contract emits them

- **Target**: evm_hooks.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The main method through which funds are intended to be bridged over from the zEVM to another chain is by calling the `send()` function in the zEVM's ZetaConnectorZEVM contract. This function emits a `ZetaSent` event, which is intended to be processed by the `crosschain` module's `PostTxProcessing()` hook.

It is crucial that this function checks to ensure that any `ZetaSent` event it picks up originated from the ZetaConnectorZEVM contract. Otherwise, any malicious attacker can deploy their own contract on the zEVM and emit arbitrary `ZetaSent` events to send arbitrary amounts of ZETA without actually holding any ZETA.

Impact

Inside the `PostTxProcessing()` hook, we see the following:

```
func (k Keeper) PostTxProcessing(
    ctx sdk.Context,
    msg core.Message,
    receipt *ethtypes.Receipt,
) error {
    target := receipt.ContractAddress
    if msg.To() ≠ nil {
        target = *msg.To()
    }
    for _, log := range receipt.Logs {
        eZRC20, err := ParseZRC20WithdrawalEvent(*log)
        if err == nil {
            if err := k.ProcessZRC20WithdrawalEvent(ctx, eZRC20, target,
    ""); err ≠ nil {
                return err
```

```
            }
        }
        eZeta, err := ParseZetaSentEvent(*log)
        if err == nil {
            if err := k.ProcessZetaSentEvent(ctx, eZeta, target, ""); err
    != nil {
                return err
            }
        }
    }
    return nil
}
```

The `receipt` parameter of this function contains information about transactions that occur on the zEVM. This function iterates through all logs (i.e., emitted events) in each receipt and attempts to parse the events as `Withdrawal` or `ZetaSent` events.

However, there is no check to ensure that these events originate from the ZetaConnectorZEVM contract. This allows a malicious attacker to deploy their own contract on the zEVM, which would allow them to emit arbitrary `ZetaSent` events, and thus gain access to ZETA tokens that they otherwise should not have access to.

The prototype of the `ZetaSent` event is as follows:

```
event ZetaSent(
    address sourceTxOriginAddress,
    address indexed zetaTxSenderAddress,
    uint256 indexed destinationChainId,
    bytes destinationAddress,
    uint256 zetaValueAndGas,
    uint256 destinationGasLimit,
    bytes message,
    bytes zetaParams
);
```

It is important to note that `Withdrawal` events are not affected by this bug. The `ProcessZRC20WithdrawalEvent()` function checks and ensures that the event was emitted from a whitelisted ZRC20 token contract address.

Recommendations

Add a check to ensure that `ZetaSent` events are only processed if they are emitted from the ZetaConnectorZEVM contract.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit 8a988ae9.

3.2 Bonded validators can trigger reverts for successful transactions

- **Target**: keeper_out_tx_tracker.go
- **Category**: Coding Mistakes
- **Severity**: Critical
- **Likelihood**: High
- **Impact**: Critical

Description

A single bonded validator has the ability to add or remove transactions from the out tracker, as the only check is that they are bonded.

```
func (k msgServer) AddToOutTxTracker(goCtx context.Context, msg
    *types.MsgAddToOutTxTracker) (*types.MsgAddToOutTxTrackerResponse,
    error) {
    ctx := sdk.UnwrapSDKContext(goCtx)

    // Zellic: this is the only relevant check
    validators := k.StakingKeeper.GetAllValidators(ctx)
    if !IsBondedValidator(msg.Creator, validators) && msg.Creator
    ≠ types.AdminKey {
        return nil, sdkerrors.Wrap(sdkerrors.ErrorInvalidSigner,
    fmt.Sprintf("signer %s is not a bonded validator", msg.Creator))
    }

    // [ ... ]
}

func (k msgServer) RemoveFromOutTxTracker(goCtx context.Context, msg
    *types.MsgRemoveFromOutTxTracker)
    (*types.MsgRemoveFromOutTxTrackerResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)

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

```
    k.RemoveOutTxTracker(ctx, msg.ChainId, msg.Nonce)
    return &types.MsgRemoveFromOutTxTrackerResponse{}, nil
}
```

Impact

This allows a malicious validator to remove an entry from the out transaction tracker and replace it with another one. One way to exploit this would be to

1. Initiate a Goerli->Goerli message sending some ZETA by calling `ZetaConnectorEth.send` on the Goerli chain.
2. After processing the incoming events, a new transaction will be signed, sending the ZETA back to the Goerli chain in `signer.TryProcessOutTx` and then adding to the outgoing transaction tracker.
3. The malicious validator can then remove that transaction using `tx crosschain remove-from-out-tx-tracker 1337 nonce` and add a different transaction that has previously failed (any failed hash will do) using the original nonce.
4. Then, `observeOutTx` will pick up this fake transaction from the tracker and add it to `ob.outTXConfirmedReceipts` and `ob.outTXConfirmedTransaction`.
5. Next, `IsSendOutTxProcessed` is run using this fake receipt and `PostReceiveConfirmation` is called, marking that status as `ReceiveStatus_Failed`.
6. The flow then continues on to revert the cross-chain transactions (CCTXs) and return the ZETA even though the original transaction went through, causing more ZETA to be transferred than was originally sent.

Here is what the attacker's ZETA balance would look like when performing the above attack:

```
90000000000000000000 // initial balance
89000000000000000000 // balance after triggering ZetaConnectorEth.send
89799999999799398194 // balance after receiving funds from the deleted
    out tracker tx
90399999999398194581 // balance after receiving the revert funds
```

Recommendations

Consider whether a single validator should be able to remove transactions from the out tracker or whether it could be done via a vote. If it is unnecessary, then the feature should be removed.

The `observeOutTx` method could be hardened to ensure that the sender of the transaction is the correct threshold signature scheme (TSS) address and that the nonce of the transaction matches the expected value. This does not prevent a malicious validator from removing legitimate transactions from the tracker and locking up funds.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit's 24d4f9eb and 8222734c.

3.3 Sending ZETA to a Bitcoin network results in BTC being sent instead

- **Target**: btc_signer.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

There are three different types of coin that can be sent via outgoing transactions, which are `CoinType_Zeta`, `CoinType_Gas`, and `CoinType_ERC20`.

The observer will call `go signer.TryProcessOutTx(send, outTxMan, outTxID, chainClient, co.bridge)` on each of the current out transactions, and it is up to the signer implementation for each chain to handle the different coin types. The `EVMSigner` correctly handles all the coin types, but the `BTCSigner` assumes that all the transactions are of type `CoinType_Gas`.

```
func (signer *BTCSigner) TryProcessOutTx(send *types.CrossChainTx,
    outTxMan *OutTxProcessorManager, outTxID string, chainclient
    ChainClient, zetaBridge *ZetaCoreBridge) {
    // [ ... ]

    // Zellic: - incorrect assumption of CoinType_Gas here
    included, confirmed, _ := btcClient.IsSendOutTxProcessed(send.Index,
    int(send.GetCurrentOutTxParam().OutboundTxTssNonce),
    common.CoinType_Gas)
    if included || confirmed {
        logger.Info().Msgf("CCTX already processed; exit signer")
        return
    }

    // [ ... ]
```

Impact

If you try to send ZETA to a Bitcoin chain using `ZetaConnectorZEVM.send` on the zEVM, it will generate an outgoing CCTX with a coin type of `CoinType_Zeta` and an `Amount` of the ZETA that was burnt. This will then get picked up by the `BTCSigner` and processed as if it was a `CoinType_Gas`, which directly sends `Amount / 1e8` (the BTC gas coin has decimals of 8 in zEVM) of BTC to the receiver.

This allows someone to burn a tiny fraction of a ZETA (1/1e10) and receive one BTC in return.

Recommendations

The `BTCSigner` should reject any transactions that are not of type `CoinType_Gas`. The `EvmHooks` could check to ensure that the destination chain supports `CoinType_Zeta` and could reject any transactions before they reach the inbound tracker.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit 630c515f.

3.4 Race condition in Bitcoin client leads to double spend

- **Target**: btc_signer.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The Bitcoin client is used to watch for cross-chain transactions as well as to relay transactions to and from the Bitcoin chain. There are numerous functions in the client, but the relevant functions are described below:

1. `IsSendOutTxProcessed()` - Checks the `ob.submittedTx[outTxID]` to see whether the transaction in question has already been submitted for relaying.
2. `startSendScheduler()` - Runs every three seconds. This function gets all pending CCTX and checks if they have already been submitted with `IsSendOutTxProcessed()`. If the CCTX has not been submitted, it will call `TryProcessOutTx()`.
3. `TryProcessOutTx()` - Signs and broadcasts a CCTX, then adds it to a tracker in the `x/crosschain` module with `AddTxHashToOutTxTracker()`.
4. `observeOutTx()` - Runs every two seconds. It queries for all transactions that have been added to the tracker in the `x/crosschain` module and adds them to `ob.submittedTx[outTxID]`.

Impact

The bug here occurs due to the racy check in `IsSendOutTxProcessed()`. More specifically, the following scenario would lead to the bug:

1. First, `startSendScheduler()` runs and gets a pending CCTX. It checks that the CCTX has not been processed (i.e., has not been added to `ob.submittedTx[]`, so `IsSendOutTxProcessed()` returns false), and thus calls `TryProcessOutTx()`.
2. Then, `TryProcessOutTx()` signs the CCTX and broadcasts it, then adds it to the tracker in the `x/crosschain` module.
3. After, `startSendScheduler()` runs again before `observeOutTx()` is able to run. The CCTX is in the `x/crosschain` module tracker but not yet in `ob.submittedTx[]` since `observeOutTx()` has not run yet. Therefore, `TryProcessOutTx()` is called again.

4. Then `TryProcessOutTx()` runs, signs, broadcasts, and adds the same CCTX to the tracker in the `x/crosschain` module.
5. Finally, `observeOutTx()` runs and adds (or in this case, overwrites) the CCTX to `ob.submittedTx[]`.

The bug occurs in step 3. Since `observeOutTx()` is responsible for adding the CCTX to the `ob.submittedTx[]` map, the intention is for `observeOutTx()` to run before `startSendScheduler()` runs again. Due to the racy nature of the code though, this does not happen, and thus the bug is triggered.

The bug triggers with the current smoke tests by modifying the following line of code in `bitcoin_client.go` to make `observeOutTx()` run every 30 seconds.

```
func (ob *BitcoinChainClient) observeOutTx() {
    ticker := time.NewTicker(30 * time.Second)

    // [ ... ]
}
```

Recommendations

A naive fix for this bug is to modify `IsSendOutTxProcessed()` to make it query for pending CCTXs in the `x/crosschain` module's tracker instead. This will prevent this issue from occurring, as `startSendScheduler()` and `TryProcessOutTx()` run synchronously.

Although the above fix is sufficient for this specific issue, we find it important to note that the code here is multithreaded and accesses `ob.submittedTx[]` asynchronously without any locking involved. Additionally, `ob.submittedTx[]` is often out of sync with the tracker in the `x/crosschain` module. Code like this is prone to similar bugs, and it is especially prone to bugs being introduced in the future. Because of this, it is our recommendation that the ZetaChain team do a thorough refactoring of the code to introduce synchronization between the functions. This would eliminate the racy nature of the code and make it less likely for bugs to be introduced in the future.

Remediation

This issue has been acknowledged by ZetaChain.

3.5 Not waiting for minimum number of block confirmations results in double spend

- **Target**: btc_client.go
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: Critical
- **Impact**: Critical

Description

Forks of length 1 (that is, a reorganization of one block in the blockchain) happen semifrequently in the Bitcoin chain. This occurs when two miners mine a winning nonce at nearly the same time. When this occurs, each full node will consider the first block it sees (from either miner) to be the best block for that block height. This would mean that for a short period of time, nodes will be divided on which block should be part of the canonical chain.

Some nodes will continue with block A, while the others will continue with block B. The way the nodes come to consensus on which chaintip to follow is by waiting to see which chaintip pulls ahead of the other by adding another block. When this occurs, all nodes that are not on this chaintip will reorganize to the longest chaintip.

Note that forks of length greater than 1 can also occur, but the probability of it occurring goes down as the length goes up. In Satoshi's Bitcoin whitepaper, it is recommended that applications wait for six block confirmations after a transaction before considering it to be part of the canonical chain (i.e., confirmed and irreversible). This assumes that a malicious attacker who is attempting to construct a malicious chaintip has access to ~10% of the total hashing power of all nodes on the chain.

Impact

In the Bitcoin client, there is a state variable for the amount of block confirmations that the code must wait before considering a transaction as confirmed.

```
type BitcoinChainClient struct {
    // [ ... ]
    confCount int64 // must wait this many blocks to be considered
    "confirmed"
    // [ ... ]
}
```

However, this variable is not used anywhere in the code. The client assumes that

any transaction it sees in new blocks are confirmed, and it will create and broadcast CCTXs immediately. This causes an issue, because if the Bitcoin chain reorganizes at any point in time after the CCTX has been created, the Bitcoin transaction will revert, but funds will have already been sent across to the zEVM.

To demonstrate this in the local testing environment, we used the `invalidateblock` RPC call. The steps for the attack are as follows:

1. Send 1 BTC from the `smoketest` wallet to the Bitcoin TSS address `bcrt1q7cj32g6scwdaa5sq08t7dqn7jf7ny9lrqhgrwz`.
2. Mine a block using the `generatetoaddress` RPC.
3. Confirm that the transaction was included, either by checking the client logs for the CCTX or using a block explorer such as btc-rpc-explorer.
4. Use the `invalidateblock` RPC to invalidate the block that the transaction occurred in.

The above steps will result in a CCTX being generated for 1 BTC to be sent to the zEVM. However, due to the reorganization triggered in step 4, the 1 BTC that was sent in step 1 will remain in the `smoketest` wallet. Therefore, 1 BTC will essentially have been minted in the zEVM.

Recommendations

The Bitcoin client should wait for a minimum number of block confirmations before assuming that a block has been confirmed. The recommended number is six block confirmations according to the Bitcoin whitepaper.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit c276e903.

3.6 Multiple events in the same transaction causes loss of funds and chain halting

- **Target**: evm_hooks.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The `ProcessZetaSentEvent()` and `ProcessZRC20WithdrawalEvent()` functions are used to process `ZetaSent` and `Withdrawal` events respectively. These events are emitted by the ZetaConnectorZEVM contract.

These functions first use the parameters of the emitted event to create a new `MsgSendVoter` message. It then hashes this message and uses the hash as an index to create a new CCTX.

The relevant code in `ProcessZetaSentEvent()` is shown below:

```
func (k Keeper) PostTxProcessing(/* ... */) error {
    // [ ... ]

    for _, log := range receipt.Logs {
        // [ ... ]

        eZeta, err := ParseZetaSentEvent(*log)
        if err == nil {
            if err := k.ProcessZetaSentEvent(ctx, eZeta, target, ""); err
    != nil {
                return err
            }
        }
    }
    return nil
}

func (k Keeper) ProcessZetaSentEvent(ctx sdk.Context, event
    *contracts.ZetaConnectorZEVMZetaSent, contract ethcommon.Address,
    txOrigin string) error {
    // [ ... ]

    msg := zetacoretypes.NewMsgSendVoter("", contract.Hex(),
```

```
    senderChain.ChainId, txOrigin, toAddr, receiverChain.ChainId, amount,
    "", event.Raw.TxHash.String(), event.Raw.BlockNumber, 90000,
    common.CoinType_Zeta, "")
    sendHash := msg.Digest()
    cctx := k.CreateNewCCTX(ctx, msg, sendHash,
    zetacoretypes.CctxStatus_PendingOutbound, &senderChain,
    receiverChain)
    EmitZetaWithdrawCreated(ctx, cctx)
    return k.ProcessCCTX(ctx, cctx, receiverChain)
}
```

Impact

An issue arises if two or more events are emitted in the same transaction with the same parameters. To demonstrate this, let us assume that two identical `ZetaSent` events are emitted in the same transaction. If the parameters are the same, then the `sendHash` that is generated from hashing the `MsgSendVoter` message will be identical for both the events. When this happens, the CCTX that is created will be the same for both events, and thus the CCTX created for the second `ZetaSent` event will overwrite the CCTX created for the first `ZetaSent` event.

An example of a scenario in which this might occur is when a user wants to send 10,000 ZETA tokens to their own address on a different chain. One way they might do this is by opting to send 5,000 ZETA in two `ZetaSent` events. Since all other parameters would be the same, only the second `ZetaSent` event gets processed (the CCTX overwrites the first one). This causes the user to only receive 5,000 ZETA on the receiving chain, even though they originally sent 10,000 ZETA.

Additionally, the `ProcessCCTX()` function will increment the nonce twice in the above scenario. Ethereum enforces that nonces have to always increase by one after each transaction, so in the event that this issue occurs, all outgoing transactions to the receiving chain will begin to fail, halting the bridge in the process.

```
func (k Keeper) ProcessCCTX(ctx sdk.Context, cctx
    zetacoretypes.CrossChainTx, receiverChain *common.Chain) error {
    // [ ... ]

    err := k.UpdateNonce(ctx, receiverChain.ChainId, &cctx)
    if err != nil {
        return fmt.Errorf("ProcessWithdrawalEvent: update nonce failed:
    %s", err.Error())
    }
```

```
    // [ ... ]
}

func (k Keeper) UpdateNonce(ctx sdk.Context, receiveChainID int64, cctx
    *types.CrossChainTx) error {
    chain :=
    k.zetaObserverKeeper.GetParams(ctx).GetChainFromChainID(receiveChainID)

    nonce, found := k.GetChainNonces(ctx, chain.ChainName.String())
    if !found {
        return sdkerrors.Wrap(types.ErrCannotFindReceiverNonce,
    fmt.Sprintf("Chain(%s) | Identifiers : %s ",
    chain.ChainName.String(), cctx.LogIdentifierForCCTX()))
    }

    // SET nonce
    cctx.GetCurrentOutTxParam().OutboundTxTssNonce = nonce.Nonce
    nonce.Nonce++
    k.SetChainNonces(ctx, nonce)
    return nil
}
```

Recommendations

We recommend introducing an ever-increasing nonce within the ZetaConnectorZEVM smart contract. Whenever a new event is emitted by the smart contract, this nonce should be incremented. This means that every emitted event is distinct from all other emitted events, and thus each emitted event will cause the creation of a new CCTX, preventing this issue from occurring.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit 2fdec9ef.

3.7 Missing authentication when adding node keys

- **Target**: keeper_node_account.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Critical
- **Impact**: Critical

Description

The `SetNodeKeys` message allows a node to supply a public key that will be used for the TSS signing:

```
func (k msgServer) SetNodeKeys(goCtx context.Context, msg
    *types.MsgSetNodeKeys) (*types.MsgSetNodeKeysResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)
    addr, err := sdk.AccAddressFromBech32(msg.Creator)
    if err ≠ nil {
        return nil, sdkerrors.Wrap(sdkerrors.ErrInvalidRequest,
    fmt.Sprintf("msg creator %s not valid", msg.Creator))
    }
    _, found := k.GetNodeAccount(ctx, msg.Creator)
    if !found {
        na := types.NodeAccount{
            Creator: msg.Creator,
            Index: msg.Creator,
            NodeAddress: addr,
            PubkeySet: msg.PubkeySet,
            NodeStatus: types.NodeStatus_Unknown,
        }
        k.SetNodeAccount(ctx, na)
    } else {
        return nil, sdkerrors.Wrap(sdkerrors.ErrInvalidRequest,
    fmt.Sprintf("msg creator %s already has a node account", msg.Creator))
    }

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

The issue is that there are no authentication or verification checks in place to limit who can call it. As a result, anyone can call the function and add their public key to the list.

Impact

The list of node accounts is fetched in the `InitializeGenesisKeygen` and in the zeta client's `genNewKeysAtBlock` in order to determine the public keys that should be used for the TSS signing. If anyone is able to add their public key before the list is queried (for example, just before the block number that the new keys will be generated), they could potentially be able to control enough signatures to pass the threshold and sign transactions or otherwise create a denial of service where the TSS can no longer sign anything.

Recommendations

Adding node accounts should be a privileged operation, and only trusted keys should be able to be added.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit a246e64b.

3.8 Missing `nil` check when parsing client event

- **Target**: evm_client.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

One of the responsibilities of the zeta client is to watch for incoming transactions and handle any `ZetaSent` events emitted by the connector.

```
logs, err := ob.Connector.FilterZetaSent(&bind.FilterOpts{
    Start: uint64(startBlock),
    End: &tb,
    Context: context.TODO(),
}, []ethcommon.Address{}, []*big.Int{})

if err ≠ nil {
    return err
}
cnt, err := ob.GetPromCounter("rpc_getLogs_count")
if err ≠ nil {
    return err
}
cnt.Inc()

// Pull out arguments from logs
for logs.Next() {
    event := logs.Event
    ob.logger.Info().Msgf("TxBlockNumber %d Transaction Hash: %s Message
    : %s", event.Raw.BlockNumber, event.Raw.TxHash, event.Message)
    destChain
    := common.GetChainFromChainID(event.DestinationChainId.Int64())
    destAddr := clienttypes.BytesToEthHex(event.DestinationAddress)

    if strings.EqualFold(destAddr, con-
    fig.ChainConfigs[destChain.ChainName.String()].ZETATokenContractAddress)
    {
        ob.logger.Warn().Msgf("potential attack attempt: %s destination
    address is ZETA token contract address %s", destChain, destAddr)}
```

When fetching the destination chain, `common.GetChainFromChainID(event.DestinationChainId.Int64())` is used, which will return `nil` if the chain is not found.

```
func GetChainFromChainID(chainID int64) *Chain {
    chains := DefaultChainsList()
    for _, chain := range chains {
        if chainID == chain.ChainId {
            return chain
        }
    }
    return nil
}
```

Since a user is able to specify any value for the destination chain, if a nonsupported chain is used, then `destChain` will be `nil` and the following `destChain.ChainName` call will cause the client to crash.

Impact

As all the clients watching the remote chain will see the same events, a malicious user (or a simple mistake entering the chain) will cause all the clients to crash. If the clients automatically restart and try to pick up from the block they were up to (the default), then they will crash again and enter into an endless restart and crash loop. This will prevent any incoming or outgoing transactions on the remote chain from being processed, effectively halting that chain's integration.

Recommendations

There should be an explicit check to ensure that `destChain` is not nil and to skip the log if it is.

It would also be a good idea to have a recovery mechanism that can handle any blocks that cause the client to crash and skip them. This will help prevent the remote chain from being paused if a similar bug occurs again.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit 0dfbf8d7.

3.9 Case-sensitive address check allows for double signing

- **Target**: keeper_chain_nonces.go
- **Category**: Coding Mistakes
- **Severity**: High
- **Likelihood**: Low
- **Impact**: High

Description

The `IsDuplicateSigner()` function is used to check whether a given address already exists within a list of signers. It does this by doing a string comparison, which is case sensitive.

```
func isDuplicateSigner(creator string, signers []string) bool {
    for _, v := range signers {
        if creator == v {
            return true
        }
    }
    return false
}
```

This function is used in `CreateTSSVoter()`, which is the message handler for the `MsgCreateTSSVoter` message. This message is used by validators to vote on a new TSS.

```
func (k msgServer) CreateTSSVoter(goCtx context.Context, msg
    *types.MsgCreateTSSVoter) (*types.MsgCreateTSSVoterResponse, error) {
    // [ ... ]

    if isDuplicateSigner(msg.Creator, tssVoter.Signers) {
        return nil, sdkerrors.Wrap(sdkerrors.ErrorInvalidSigner,
    fmt.Sprintf("signer %s double signing!!", msg.Creator))
    }

    // [ ... ]

    // this needs full consensus on all validators.
    if len(tssVoter.Signers) == len(validators) {
        tss := types.TSS{
            Creator:  "",
            Index:  tssVoter.Chain,
```

```
            Chain: tssVoter.Chain,
            Address: tssVoter.Address,
            Pubkey: tssVoter.Pubkey,
            Signer: tssVoter.Signers,
            FinalizedZetaHeight: uint64(ctx.BlockHeader().Height),
        }
        k.SetTSS(ctx, tss)
    }

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

Impact

In Cosmos-based chains, addresses are alphanumeric, and the alphabetical characters in the address can either be all uppercase or all lowercase when represented as a string. This means that case-sensitive string comparisons, such as the one in `IsDuplicateSigner()`, can allow a single `creator` to pass the check twice — once for an all lowercase address, and once for an all uppercase version of the same address.

Due to the `len(tssVoter.Signers) == len(validators)` check, it is possible for a malicious actor to spin up multiple bonded validators and double sign with each of them. This would cause the check to erroneously pass, even though full consensus has not been reached, and allow the malicious actor to effectively force the vote to pass.

Recommendations

The `sdk.AccAddressFromBech32()` function can be used to convert a `string` address to an instance of a `sdk.AccAddress` type. Comparing two `sdk.AccAddress` types is the correct way to compare addresses in Cosmos-based chains, and it will fix this issue.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit 83d0106b.

3.10 No panic handler in Zetaclient may halt cross-chain communication

- **Target**: btc_signer.go
- **Category**: Coding Mistakes
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

The code under `zetaclient/` implements two separate clients — an EVM client for all EVM-compatible chains and a Bitcoin client for the Bitcoin chain. The clients are intended to relay transactions between chains as well as watch for cross-chain interactions (via emitted events).

Impact

In the event that a panic occurs in the `zetaclient` code, the client will simply crash. If a malicious actor is able to find a reliable way to cause panics, they can effectively halt all cross-chain communications by crashing all of the clients for that specific chain.

We discovered a bug in the Bitcoin client that can allow a malicious actor to achieve this; however, there may be numerous other ways to do this. The bug exists in the Bitcoin client's `TryProcessOutTx()` function.

```
func (signer *BTCSigner) TryProcessOutTx(send *types.CrossChainTx,
    outTxMan *OutTxProcessorManager, outTxID string, chainclient
    ChainClient, zetaBridge *ZetaCoreBridge) {
    // [ ... ]

    // FIXME: config chain params
    addr, err := btcutil.DecodeAddress(string(toAddr),
    config.BitconNetParams)

    if err != nil {
        logger.Error().Err(err).Msgf("cannot decode address %s ",
    send.GetCurrentOutTxParam().Receiver)
        return
    }

    // [ ... ]
}
```

Specifically, the call to `btcutil.DecodeAddress()` can panic if the `toAddr` provided to it is not a valid Bitcoin address. This is easily achieved by passing in an EVM-compatible address instead. The following stack trace is observed when the crash occurs:

```
zetaclient0 | panic: runtime error: index out of range [65533] with length
    256
zetaclient0 |
zetaclient0 | goroutine 12508 [running]:
zetaclient0 | github.com/btcsuite/btcutil/base58.Decode({0xc005e9f968,
    0x14})
zetaclient0 | ^^I/go/pkg/mod/github.com/btcsuite/btcutil@v1.0.3-
    0.20201208143702-a53e38424cce/base58/base58.go:58 +0x305
zetaclient0
    | github.com/btcsuite/btcutil/base58.CheckDecode({0xc005e9f968?,
    0xc001300000?})
zetaclient0 | ^^I/go/pkg/mod/github.com/btcsuite/btcutil@v1.0.3-
    0.20201208143702-a53e38424cce/base58/base58check.go:39 +0x25
zetaclient0 | github.com/btcsuite/btcutil.DecodeAddress({0xc005e9f968?,
    0xc0061a6de0?}, 0x458b080)
zetaclient0 | ^^I/go/pkg/mod/github.com/btcsuite/btcutil@v1.0.3-
    0.20201208143702-a53e38424cce/address.go:182 +0x2aa
zetaclient0 | github.com/zeta-
    chain/zetacore/zetaclient.(*BTCSigner).TryProcessOutTx(0xc004aed680,
    0xc006691680, 0xc00053aab0, {0xc00484edc0, 0x4a}, {0x32c9040?,
    0xc0050ba200}, 0xc000e9af00)
zetaclient0
    | ^^I/go/delivery/zeta-node/zetaclient/btc_signer.go:213 +0x893
zetaclient0 | created by github.com/zeta-
    chain/zetacore/zetaclient.(*CoreObserver).startSendScheduler
zetaclient0 | ^^I/go/delivery/zeta-
    node/zetaclient/zetacore_observer.go:224 +0x1045
```

Recommendations

The bug demonstrated above is in an external package that is not maintained by the ZetaChain team. Since it is not sustainable to go through and fix any such bugs that arise from the use of external packages, we recommend adding a panic handler to the Zetaclient code so that panics are handled gracefully and preferably logged, so they can be taken care of later.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit f2adb252.

3.11 Ethermint Ante handler bypass

- **Target**: app/ante/handler_options.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

It is possible to bypass the `EthAnteHandler` by wrapping the `ethermint.evm.v1.MsgEthereumTx` inside a `MsgExec` as described in https://jumpcrypto.com/bypassing-ethermint-ante-handlers/. These are responsible for numerous vital actions such as deducting the gas limit from the sender's account to limit the number computations a contract can perform.

Impact

It is possible to cause a complete chain halt by deploying a contract with an infinite loop and then calling it with a huge gas limit. Since the coins are not deducted from the senders account, the gas limit will be accepted and the EVM will get stuck in the loop.

The following steps can be performed to replicate this issue. First, create a new account to simulate a malicious user, then deploy the following contract to the zEVM:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;
contract Demo {
    function loop() external {
            while(true) {}
    }
}
```

Using the details of the malicious account (one can use `zetacored keys unsafe-export-eth-key` to get the private key) and the deployed contract, sign a transaction and get the hex bytes:

```
import web3
from web3 import Web3

account = "0x30b254F67cBaB5E6b12b92329b53920DE403aA02"
contract = "0x6da71267cd63Ec204312b7eD22E02e4E656E72ac"
```

```
private_key = "xxx"

loop_selector = "0xa92100cb"
loop_data={"data":loop_selector,"from": account, "gas":
    "0xFFFFFFFFFFFFFFF","gasPrice": "0x7","to": contract,"value": "0x0",
    "nonce": "0x0"}

w3 = web3.Web3(web3.HTTPProvider("http://localhost:9545"))
print(w3.eth.account.sign_transaction(transaction_dict=nop_data,
    private_key=private_key))
```

This can then be used to generate a `MsgEthereumTx` message, which we then remove the `ExtensionOptionsEthereumTx` and wrap it in a `MsgExec` using the authz grant mechanism:

```
zetacored tx evm raw [TX_HASH] --generate-only > /tmp/tx.json
sed -i 's/{"@type":"\/ethermint.evm.v1.ExtensionOptionsEthereumTx"}//g'
    /tmp/tx.json
zetacored tx --chain-id athens_101-1 --keyring-backend=test --from $hacker
    authz exec /tmp/tx.json --fees 20azeta --yes
```

Since the granter and the grantee are the same in this instance, the grant automatically passes, causing the inner message to be executed and putting the nodes in an infinite loop.

It is also possible to steal all the transaction fees for the current block by supplying a higher gas limit that is used. Since the gas was never paid for, when `RefundGas` is triggered, it will end up sending any gas that was collected from other transactions.

Recommendations

Consider adding a new Ante handler base on the `AuthzLimiterDecorator` that was used to fix the issue in EVMOS;

see https://github.com/evmos/evmos/blob/v12.1.2/app/ante/cosmos/authz.go#L58-L91.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit 3362b137.

3.12 Unbonded validators prevent the TSS vote from passing

- **Target**: keeper_tss_voter.go
- **Category**: Coding Mistakes
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Medium

Description

Bonded validators can cast a vote to add a new TSS by sending a `MsgCreateTSSVoter`. The issue is that there is a check to allow only bonded validators to vote, but for the vote to pass, the number of signers must be equal to the total number of validators (which includes unbonded/unbonding validators).

```
func (k msgServer) CreateTSSVoter(goCtx context.Context, msg
    *types.MsgCreateTSSVoter) (*types.MsgCreateTSSVoterResponse, error) {
    ctx := sdk.UnwrapSDKContext(goCtx)

    validators := k.StakingKeeper.GetAllValidators(ctx)

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

    // [ ... ]
    // this needs full consensus on all validators.
    if len(tssVoter.Signers) == len(validators) {
        tss := types.TSS{
            Creator: "",
            Index: tssVoter.Chain,
            Chain: tssVoter.Chain,
            Address: tssVoter.Address,
            Pubkey: tssVoter.Pubkey,
            Signer: tssVoter.Signers,
            FinalizedZetaHeight: uint64(ctx.BlockHeader().Height),
        }
        k.SetTSS(ctx, tss)
    }

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

Impact

If not every validator is a bonded validator, then it is impossible to add a new TSS as the vote can never pass. As anyone can become an unbonded validator, this would be easy to trigger and will likely happen in the course of normal operation as validators will unbond, putting them into an `unbonding` state.

It is also possible for a bonded validator to sign the vote, become unbonded and removed, and have the vote still count.

Recommendations

The vote should only be passed when the set of currently bonded validators have all signed it.

Remediation

This issue has been acknowledged by ZetaChain.

3.13 Code maturity

- **Target**: N/A
- **Category**: Coding Mistakes
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

Codebases that contain commented-out functions, dead code, TODOs, FIXMEs, and other similar elements can be challenging to maintain and audit for security vulnerabilities. These elements can make it easier for developers to introduce bugs into the codebase unknowingly, as they may not be aware of the intended functionality of the commented-out or unfinished code. Additionally, leaving these elements in the codebase can clutter the code and make it harder to understand and maintain over time.

Impact

The codebase contains a significant number of TODOs, FIXMEs, commented-out code, dead code, and empty functions. This impacts readability and makes the codebase harder to comprehend and maintain.

- msg_server_remove_foreign_coin.go - The `RemoveForeignCoin` handler is commented out and does nothing.
- observer_mapper.go - The `AddObserver` handler is commented out and does nothing.
- keeper_chain_nonces.go - The final else block is unreachable as both the true and false states for `isFound` and handled above.

There are 48 TODOs and FIXMEs in `x/*` and 37 in `zetaclient/*`, ranging from adding better error messages to correctly handling gas limits. By addressing these, the codebase will exhibit improved robustness, maintainability, and resilience against potential vulnerabilities, ultimately resulting in a more reliable and secure product for end users.

Recommendations

These issues should be fixed by completing or removing pending tasks, eliminating dead code and commented-out sections, and populating empty functions with appropriate logic or removing them altogether. This will not only enhance the code quality but also facilitate future audits and improve the system's overall security posture.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit 80f47883.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Missing validation

Code maturity is a critical aspect of high-assurance projects. By implementing checks, we can protect against potential mishaps, reduce the risk of lost funds or frozen protocols, and improve user experience. In addition, adding extra error messages can help clarify the internal mechanisms and reduce potential bugs that future developers might introduce while building on this project.

- `keeper_cross_chain_tx_vote_outbound_tx.go: VoteOnObservedOutboundTx` – There is no check that the `observationChain` is not `nil`, which will cause a panic. During the normal operation, the chain should always exist, but since the `IsAuthorized` check uses the `observationChain`, it is possible for anyone to trigger it by sending a `MsgVoteOnObservedOutboundTx` message. The panic is handled by the default recovery middleware but would be better to explicitly check for `nil` and return an error.
- `btc_signer.go: SignWithdrawTx` – The value of the out transaction is calculated with `remainingSatoshis - fees`, but there is no guarantee that `remainingSatoshis` is greater than the fees, which would result in a negative value and the transaction failing. It would be better to explicitly check for this case and return an error indicating what has happened.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 Component: BTC client

Flow: Observing incoming cross-chain transactions from the Bitcoin chain

Observation and processing of incoming cross-chain transactions starts in the `observeInTx()` function. This function runs every five seconds (configurable).

The logic in this function is very similar to the EVM client. One issue here is that the BTC client does not wait for a certain amount of blocks (i.e., block confirmations) like the EVM client does. This leads to a double-spending issue where a malicious attacker (or even users accidentally) are able to send BTC across chains without actually spending any BTC (see 3.5).

The code then loops through all transactions in all recent unobserved blocks, filters for the ones that are destined for the BTC TSS address, takes the value of the transaction and multiplies it by 1e8, and then creates and broadcasts a `MsgVoteOnObservedInboundTx` message to the `x/crosschain` module with that amount. The coin type is set to `CoinType_Gas` in this instance.

Flow: Observing outgoing cross-chain transactions to the Bitcoin chain

Functionally identical to the EVM client's implementation of observing outgoing transactions (see 5.3).

Flow: Watch unspent transaction outputs

The `WatchUTXOS` sets up a ticker that collects all the unspent transaction outputs on the TSS address. It does this by calling `ListUnspentMinMaxAddresses` in a loop, from one to the current block number in chunks of 500.

```
for i := minConfirmations; i < maxConfirmations; i += chunkSize {
        unspents, err := ob.rpcClient.ListUnspentMinMaxAddresses(i,
    i+chunkSize, addresses)
        if err ≠ nil {
            return err
        }
        utxos = append(utxos, unspents ... )
        ob.logger.Info().Msgf("btc: fetched %d utxos", len(unspents))
    }
```

This results in around 5,000 RPC calls per tick on the BTC testnet and around 1,600 on mainnet, as the current block numbers are 2.5 million and 800,000, respectively.

The transactions are then filtered to remove any pending outputs, then sorted by the amount. It then calls `housekeepPending` to remove any completed transactions from the list of pending transaction outputs.

5.2 Module: x/crosschain

Message: `AddToOutTxTracker`

The `AddToOutTxTracker` handler is responsible for adding transaction hashes to the output transaction tracker based on the chain and a nonce. It first ensures that the caller is a bonded validator or the admin address, then either adds a new entry into the tracker or updates the hash list if the nonce is already known.

The out tracker is used by the zeta clients to check and report the success of the outgoing transaction back to the zeta core.

The `ChainId` must be a known chain, the `Nonce` can be any `uint64`, and the `TxHash` can be any string.

Since any bonded validator is allowed to add a transaction to the tracker, it is possible for a malicious validator to trigger the revert flow even if the outgoing transaction was successful (see 3.2).

Message: `RemoveFromOutTxTracker`

The `RemoveFromOutTxTracker` handler is responsible for removing transaction hashes from the output transaction tracker. It first ensures that the caller is a bonded validator or the admin address, then removes the transaction from the tracker that has the corresponding chain ID and nonce.

The `ChainId` must be a known chain, and the `Nonce` can be any `uint64`.

Since any bonded validator is allowed to remove transactions from the tracker, it is possible for a malicious validator to trigger the revert flow, even if the outgoing transaction was successful (see 3.2).

Message: `CreateTSSVoter`

The `CreateTSSVoter` handler is responsible for adding new TSS entries via a voting mechanism. Only bonded validators are allowed to vote for a new TSS, and once all validators have voted, then the TSS is added. There are two issues with this handler. The first is that the `isDuplicateSigner` can be tricked into allowing two signatures from a node instead of one (see 3.9). The second is that although only bonded validators can vote, the check to see if the vote passes is looking at every validator, including unbonded ones (see 3.12).

Message: `SetNodeKeys`

The `SetNodeKeys` handler allows a node to specify a public key that should be used for TSS signing. There is no authentication on this method, which allows anyone to add a public key (see 3.7)

Message: `UpdatePermissionFlags`

The `UpdatePermissionFlags` handler allows an admin to update whether inbound transactions from foreign chains are enabled or not. The caller must have the `stop_inboun d_cctx` policy in order to change the setting.

Message: `MsgGasPriceVoter`

The `MsgGasPriceVoter` handler allows observers to vote on the gas prices for their chain, setting the price to the median value of the submitted values. It is only callable by observers that are in the mapping for the requested chain.

Message: `MsgNonceVoter`

The `MsgNonceVoter` handler allows observers to update the nonce for a chain. It is only callable by observers that are in the mapping for the requested chain. If the chain nonce already exists, then a signer will be added to the existing entry and the nonce replaced, otherwise a new entry is created with the specified nonce.

Message: `MsgVoteOnObservedOutboundTx`

The `MsgVoteOnObservedOutboundTx` handler is responsible for keeping track of the status of transactions that have been sent to a foreign chain and allowing observers to vote on its status. Once two thirds of the observers have voted, the ballot is finalized and the CCTX is either marked as completed or it is reverted in an attempt to return the funds.

Only valid observers for the sender chain are able to successfully vote on outbound transactions.

Message: `MsgVoteOnObservedInboundTx`

The `MsgVoteOnObservedInboundTx` handler is responsible for keeping track of whether an inbound transaction is legitimate by holding a ballot that the observers can vote on. If two thirds of the observers vote in the same way, then the ballot is considered finalized and a new CCTX is generated.

Only valid observers for the sender chain are able to successfully vote on inbound transactions.

Function: `PostTxProcessing()`

The `PostTxProcessing()` hook is used to process all `ZetaSent` and `Withdrawal` events that are emitted by contracts on the zEVM.

The processing of the `ZetaSent` event is done in the `ProcessZetaSentEvent()` function. It is identical to that of the `ZetaSent` event emitted by the connector contract on EVM-compatible chains, with one key difference — the code here fails to ensure that only `ZetaSent` events emitted by the zEVM connector contract are processed. This allows an attacker to deploy a contract that emits `ZetaSent` events with the same signature, which allows them to transfer ZETA tokens across chains without ever owning any ZETA. This essentially lets them mint free tokens out of nothing, which is a critical issue.

The processing of the `Withdrawal` event is done in the `ProcessZRC20WithdrawalEvent()` function. It is also very similar to the processing of the `ZetaSent` event above, except that it does check that the event is emitted by a whitelisted ZRC20 contract on the zEVM. It creates and broadcasts a `MsgVoteOnObservedInboundTx` message with the `value` field set to the event's `value` field, and the recipient set to the event's `to` field.

The format of the `Withdrawal` event is shown below:

```
event Withdrawal(address indexed from, bytes to, uint256 value,
    uint256 gasfee, uint256 protocolFlatFee)
```

5.3 Component: EVM client

Flow: Observing incoming cross-chain transactions from EVM-compatible chains

Observation and processing of incoming cross-chain transactions starts in the `ExternalChainWatcher()` function. This function runs the `observeInTX()` function every `chainconfig.BlockTime` seconds, up to a maximum of 24 seconds. For Ethereum, this would be 12 seconds.

The `observeInTX()` function has the bulk of the functionality. It first ensures that the chain the transaction is coming from has been whitelisted. All external chains are blacklisted by default. It then ensures to go back to the last confirmed block (i.e., it does not take the latest block on the tip of the chain as confirmed). This prevents potential issues where a chain reorganization would lead to a double spend.

It then looks for any emitted `ZetaSent` events from the last checked block to the latest confirmed block. It ensures to get emitted events only from the Connector contract, as otherwise anyone would be able to deploy their own contract and emit events of the same signature to reach this code path.

For each `ZetaSent` event, the code uses the `PostSend()` function to construct a `MsgVoteOnObservedInboundTx` message to broadcast to Zetacore. The coin type for the amount of coins being sent across chains is set to `CoinType_Zeta` for this case, which means `ZetaSent` events are only used to transfer ZETA across chains. This message will get picked up by the `x/crosschain` module. See 5.2 for more information on what happens then.

Next, the code does the same thing for `Deposited` events from the `ERC20Custody` contract, except this time the coin type is set to `CoinType_ERC20`.

Finally, one other thing users can do is transfer native tokens to a designated TSS address. When they do this, the transfer is noted as a "gas deposit". This is how users pay for gas in cross-chain transactions, and `observeInTX()` handles this case for both EVM- and Klaytn-based chains. The logic is largely the same — a `MsgVoteOnObservedInboundTx` message is constructed with the coin type set to `CoinType_Gas` and then subsequently broadcasted.

The inputs that are controllable in this flow are emitted event parameters. Specifically, for the `ZetaSent` event, a user would be able to fully control the following parameters:

```
event ZetaSent(
    address sourceTxOriginAddress, // tx.origin
    address indexed zetaTxSenderAddress, // msg.sender
```

```
    uint256 indexed destinationChainId, // CONTROLLED
    bytes destinationAddress, // CONTROLLED
    uint256 zetaValueAndGas, // CONTROLLED
    uint256 destinationGasLimit, // CONTROLLED
    bytes message, // CONTROLLED
    bytes zetaParams // CONTROLLED
);
```

We found one issue where, if the `destinationChainId` is set to a nonexistent chain, then a `nil pointer dereference` exception will be thrown inside `observeInTX()`.

The code also specifically checks to see if `destinationAddress` is set to the ZETA token contract address. If it is, the code logs a potential attack attempt, but it continues to run normally otherwise.

For the `Deposited` event, a user would be able to fully control the following parameters:

```
event Deposited(
    bytes recipient, // CONTROLLED
    IERC20 asset, // Partially controlled, must be whitelisted
    uint256 amount, // CONTROLLED,
    but must own at least this amount of tokens
    bytes message // CONTROLLED
);
```

None of the parameters are checked or used in the code. They are simply passed into the `MsgVoteOnObservedInboundTx` message that is later broadcasted.

For more information on how these parameters are used in the `x/crosschain` module, please see 5.2.

Flow: Observing outgoing cross-chain transactions to EVM-compatible chains

Observation and processing of incoming cross-chain transactions starts in the `observeOutTx()` function. This function currently runs every three seconds. It simply queries for the outgoing transaction tracker for a specific chain from the `x/crosschain` module. Afterwards, it loops through each transaction hash in the tracker, queries for the receipt and actual transaction using the hash, and then stores the receipt and transaction data into the `outTXConfirmedReceipts` and `outTXConfirmedTransaction` maps respectively, indexed by the nonce.

5.4 Module: x/fungible

Message: `DeployFungibleCoinZRC20`

The `DeployFungibleCoinZRC20` handler is responsible for deploying a ZRC20 token contract on the EVM. It does two different things depending on whether the coin being deployed has a type of `CoinType_Gas` or not.

If the type is `CoinType_Gas`, the handler sets up an ERC20 contract for the token and then sets up a Uniswap V2 pool for the token. The pool contains ZETA tokens and the gas token itself. It also adds 0.1 gas/0.1 zeta into the pool.

If the type is `CoinType_Zeta` or `CoinType_ERC20`, the handler simply deploys an ERC20 contract for the token on the EVM.

It is only callable by the Admin Policy account.

6 Audit Results

At the time of our audit, the code was not deployed to mainnet Ethereum.

During our audit, we discovered 13 findings. Of these, seven were of critical risk, four were of high risk, one was of medium risk, and one was a suggestion (informational). ZetaChain acknowledged all findings and implemented fixes.

6.1 Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.