Zellic

ZetaChain

Smart Contract Security Assessment

July 12, 2023

Prepared for:

Tad Tobar

ZetaChain

Prepared by:

Syed Faraz Abrar and William Bowling

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 June 26th to July 12th, 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:

- Have any security vulnerabilities been introduced as a result of the changes made between commits 62dcc227 and e27028fa?

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.

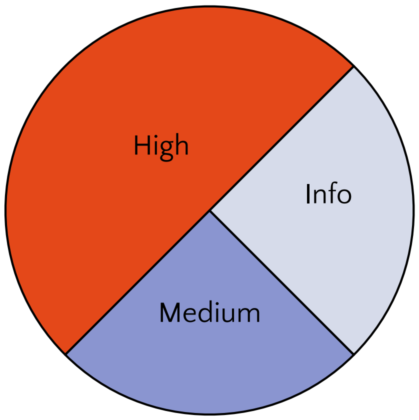
1.3 Results

During our assessment on the scoped ZetaChain modules, we discovered four findings. No critical issues were found. Two 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.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	2
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 modules.

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 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

Repositories	https://github.com/zeta-chain/zeta-node
	https://github.com/zeta-chain/protocol-contracts-audit
Versions	zeta-node: `e27028fa78b654878fa8546e152afa6cec1134e5`
	protocol-contracts-audit: `1a9a91d392a80e1cfcedfdf52d23191e7981e25`
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 three person-weeks. The assessment was conducted over the course of three 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:

Syed Faraz Abrar, Engineer
faith@zellic.io

William Bowling, Engineer
will@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

June 26, 2023	Start of primary review period
July 12, 2023	End of primary review period

3 Detailed Findings

3.1 Ethermint Ante handler bypass

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

Description

In commit `3362b13` a fix was added in order to prevent the Ethermint Ante handler from being bypassed (see https://jumpcrypto.com/writing/bypassing-ethermint-ante-handlers/).

The patch was based on the original fix implemented in Evmos, but the issue is that ZetaChain has the x/group module enabled, which allows for a new way of bypassing the Ante handler.

The `group.MsgSubmitProposal` allows for arbitrary messages to be run when the proposal is passed:

```
// MsgSubmitProposal is the Msg/SubmitProposal request type.
type MsgSubmitProposal struct {
    // group_policy_address is the account address of group policy.
    GroupPolicyAddress string
    `protobuf:"bytes,1,opt,name=group_policy_address,json=groupPolicyAddre
    ss,proto3" json:"group_policy_address,omitempty"`
    // proposers are the account addresses of the proposers.
    // Proposers signatures will be counted as yes votes.
    Proposers []string `protobuf:"bytes,2,rep,name=proposers,proto3"
    json:"proposers,omitempty"`
    // metadata is any arbitrary metadata to attached to the proposal.
    Metadata string `protobuf:"bytes,3,opt,name=metadata,proto3"
    json:"metadata,omitempty"`
    // messages is a list of `sdk.Msg`s that will be executed if the
    proposal passes.
    Messages []*types.Any `protobuf:"bytes,4,rep,name=messages,proto3"
    json:"messages,omitempty"`
    // exec defines the mode of execution of the proposal,
    // whether it should be executed immediately on creation or not.
    // If so, proposers signatures are considered as Yes votes.
    Exec Exec
    `protobuf:"varint,5,opt,name=exec,proto3,enum=cosmos.group.v1.Exec"
    json:"exec,omitempty"`
}
```

Since anyone can create a group with themselves as the only member, they can then submit a proposal with a message that will be executed immediately using the `Exec` option of `Try`. The `checkDisabledMsgs` function is only checking for `authz` messages, and so the group proposal will not be filtered:

```
func (ald AuthzLimiterDecorator) checkDisabledMsgs(msgs []sdk.Msg,
    isAuthzInnerMsg bool, nestedLvl int) error {
    if nestedLvl ≥ maxNestedMsgs {
        return fmt.Errorf("found more nested msgs than permited. Limit is
    : %d", maxNestedMsgs)
    }
    for _, msg := range msgs {
        switch msg := msg.(type) {
        case *authz.MsgExec:
            innerMsgs, err := msg.GetMessages()
            if err ≠ nil {
                return err
            }
            nestedLvl++
            if err := ald.checkDisabledMsgs(innerMsgs, true, nestedLvl);
    err ≠ nil {
                return err
            }
        case *authz.MsgGrant:
            authorization, err := msg.GetAuthorization()
            if err ≠ nil {
                return err
            }

            url := authorization.MsgTypeURL()
            if ald.isDisabledMsg(url) {
                return fmt.Errorf("found disabled msg type: %s", url)
            }
        default:
            url := sdk.MsgTypeURL(msg)
            if isAuthzInnerMsg && ald.isDisabledMsg(url) {
                return fmt.Errorf("found disabled msg type: %s", url)
            }
        }
    }
    return nil
}
```

Impact

Similar to the original finding in section 3.11 of the April 21st, 2023 report, this can be used to steal the transaction fees for the current block, and also to trigger an infinite loop, halting the entire chain.

Recommendations

A new case should be added to the `checkDisabledMsgs` method to check the `group.MsgSubmitProposal` message in the same way as the existing messages:

```
case *group.MsgSubmitProposal:
    innerMsgs, err := msg.GetMsgs()
    if err ≠ nil {
        return err
    }
    nestedLvl++
    if err := ald.checkDisabledMsgs(innerMsgs, true, nestedLvl); err
    ≠ nil {
        return err
    }
```

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit cd279b80.

3.2 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 Zetaclient is to watch for incoming transactions and handle any `ZetaSent` events emitted by the connector.

```
logs, err := connector.FilterZetaSent(&bind.FilterOpts{
    Start: uint64(startBlock),
    End: &tb,
    Context: context.TODO(),
}, []ethcommon.Address{}, []*big.Int{})
if err ≠ nil {
    ob.logger.ChainLogger.Warn().Err(err).Msgf("observeInTx:
    FilterZetaSent error:")
    return
}
// Pull out arguments from logs
for logs.Next() {
    event := logs.Event
    ob.logger.ExternalChainWatcher.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)
```

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 542eb37c.

3.3 Admin policy check will always fail

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

Description

The `AddToOutTxTracker` was changed from allowing bonded validators to call it to allowing an admin policy account or one of the current observers:

```go
func (k msgServer) AddToOutTxTracker(goCtx context.Context, msg
    *types.MsgAddToOutTxTracker) (*types.MsgAddToOutTxTrackerResponse,
    error) {
    ctx := sdk.UnwrapSDKContext(goCtx)
    chain :=
    k.zetaObserverKeeper.GetParams(ctx).GetChainFromChainID(msg.ChainId)
    if chain == nil {
        return nil, zetaObserverTypes.ErrSupportedChains
    }
    authorized := false
    if msg.Creator
    == k.zetaObserverKeeper.GetParams(ctx).GetAdminPolicyAccount
    (zetaObserverTypes.Policy_Type_out_tx_tracker) {
        authorized = true
    }
    ok, err := k.IsAuthorized(ctx, msg.Creator, chain)
    if err ≠ nil {
        return nil, err
    }
    if ok {
        authorized = true
    }
    if !authorized {
        return nil, sdkerrors.Wrap(types.ErrNotAuthorized,
    fmt.Sprintf("Creator %s", msg.Creator))
    }
```

The issue is that the admin account is unlikely to be an observer, and so the check to `IsAuthorized` will return an error and the function will return.

Impact

The admin policy will not work as expected and will be unable to add to the out tracker.

Recommendations

The function should be refactored to allow for either the admin or the observers to access it instead of returning early if the caller is not an observer.

Remediation

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

3.4 Incorrect method comments

- **Target**: keeper_tss.go
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Informational
- **Impact**: Informational

Description

The TSS keeper was changed so that there is only a single entry instead of storing it based on the index, but all of the comments still refer to the index:

```
// SetTSS set a specific tSS in the store from its index
func (k Keeper) SetTSS(ctx sdk.Context, tss types.TSS) {
    store := prefix.NewStore(ctx.KVStore(k.storeKey),
    types.KeyPrefix(types.TSSKey))
    b := k.cdc.MustMarshal(&tss)
    store.Set([]byte{0}, b)
}

// GetTSS returns a tSS from its index
func (k Keeper) GetTSS(ctx sdk.Context) (val types.TSS, found bool) {
    store := prefix.NewStore(ctx.KVStore(k.storeKey),
    types.KeyPrefix(types.TSSKey))

    b := store.Get([]byte{0})
    if b == nil {
        return val, false
    }

    k.cdc.MustUnmarshal(b, &val)
    return val, true
}
```

Impact

When reading the comments to understand the code, it is not clear that the index is now always zero and only a single TSS is stored.

Recommendations

Update the comments to reflect the new code.

Remediation

This issue has been acknowledged by ZetaChain, and a fix was implemented in commit 829ebf9a.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1 Review of changes

In this section, we summarize all the changes that were made to the codebase to remediate our findings from the previous audit.

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

This finding was fixed by introducing the ConnectorZEVM contract's address into the `SystemContract` structure. This address is now used in the EVM hooks to ensure that `ZetaSent` events are only processed if they are emitted by the ConnectorZEVM contract.

All nonrelevant code has been omitted from the excerpt below.

```
func (k Keeper) ProcessLogs(ctx sdk.Context, logs []*ethtypes.Log,
    emittingContract ethcommon.Address, txOrigin string) error {

    system, found := k.fungibleKeeper.GetSystemContract(ctx)
    connectorZEVMAddr := ethcommon.HexToAddress(system.ConnectorZevm) //
    Zellic: new ConnectorZEVM address

    for _, log := range logs {
        eZeta, err := ParseZetaSentEvent(*log, connectorZEVMAddr)
        if err == nil {
            if err := k.ProcessZetaSentEvent(ctx, eZeta,
    emittingContract, txOrigin); err ≠ nil {
                return err
            }
        }
    }
    return nil
}

func ParseZetaSentEvent(log ethtypes.Log, connectorZEVM ethcommon.Address)
```

```
    (*connectorzevm.ZetaConnectorZEVMZetaSent, error) {
    // [ ... ]

    // Zellic: new check to ensure events are emitted by this contract
    if event.Raw.Address != connectorZEVM {
        return nil, fmt.Errorf("ParseZetaSentEvent: event address %s does
    not match connectorZEVM %s", event.Raw.Address.Hex(),
    connectorZEVM.Hex())
    }
    return event, nil
}
```

3.2 — Bonded validators can trigger reverts for successful transactions

This finding was fixed by introducing a new Admin Policy account type. With this fix, although outgoing transactions can still be removed from the tracker, this is only allowed if the message creator is the Admin Policy account. Previously, any bonded validators could perform this action.

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

    // Zellic: new check added to ensure only the Admin Policy account can
    remove transactions from the out TX tracker
    if msg.Creator
    != k.zetaObserverKeeper.GetParams(ctx).GetAdminPolicyAccount
    (zetaObserverTypes.Policy_Type_out_tx_tracker) {
        return &types.MsgRemoveFromOutTxTrackerResponse{},
    zetaObserverTypes.ErrNotAuthorizedPolicy
    }

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

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

This finding was fixed by introducing a new check in the `ProcessZetaSentEvent()` function that checks to ensure that any external chains ZETA is being sent to has a ZETA-token contract deployed on it. Since the Bitcoin chain does not implement smart contracts, this check will prevent ZETA from being sent to the Bitcoin chain (and any other non-EVM-compatible chains).

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

    // Validation if we want to send ZETA to external chain, but there is
    no ZETA token.
    coreParams, found := k.zetaObserverKeeper.GetCoreParamsByChainID(ctx,
    receiverChain.ChainId)
    if !found {
        return zetacoretypes.ErrNotFoundCoreParams
    }
    if receiverChain.IsExternalChain()
    && coreParams.ZetaTokenContractAddress == "" {
        return zetacoretypes.ErrUnableToSendCoinType
    }

    // [ ... ]
}
```

3.4 — Race condition in Bitcoin client leads to double spend

This finding was fixed by introducing a new map to the Bitcoin client that now tracks all transaction hashes of successfully broadcasted transactions to the Bitcoin chain. This prevents the same transaction from being broadcasted twice, thus preventing the double spend.

```
// returns isIncluded, isConfirmed, Error
func (ob *BitcoinChainClient) IsSendOutTxProcessed(sendHash string, nonce
    int, _ common.CoinType, logger zerolog.Logger) (bool, bool, error) {
    outTxID := ob.GetTxID(uint64(nonce))
    logger.Info().Msgf("IsSendOutTxProcessed %s", outTxID)
    ob.mu.Lock()
    txnHash, broadcasted := ob.broadcastedTx[outTxID]
    res, mined := ob.minedTx[outTxID]
    ob.mu.Unlock()

    if !mined {
        if !broadcasted {
            return false, false, nil
        }

        // [ ... ]
    }

    // [ ... ]
    return true, true, nil
}
```

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

This finding was fixed by waiting for a configured amount of blocks before considering a block confirmed. In the testing environment, the Bitcoin client now waits for at least two blocks before considering a block as confirmed. On average, each block takes 10 minutes to be mined, so this would be 20 minutes for a block to be considered confirmed.

```
func GetCoreParams() CoreParamsList {
    params := CoreParamsList{
        CoreParams: []*CoreParams{
            // [ ... ]
            {
                ChainId: common.BtcRegtestChain().ChainId,
                ConfirmationCount: 2,
                // [ ... ]
            }},
    }

    // [ ... ]
}

func (ob *BitcoinChainClient) observeInTx() error {
    // [ ... ]

    // "confirmed" current block number
    confirmedBlockNum := cnt
    - int64(ob.GetChainConfig().CoreParams.ConfCount)
    if confirmedBlockNum < 0 || confirmedBlockNum > math2.MaxInt64 {
        return fmt.Errorf("skipping observer , confirmedBlockNum is
    negative or too large ")
    }

    // query incoming gas asset
    if confirmedBlockNum > lastBN {
        // [ ... ]
    }
    // [ ... ]
}
```

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

This finding was fixed by introducing an `index` to the emitted `ZetaSent` event. This is then used within the x/crosschain module to distinguish between multiple `ZetaSent` events within the same transaction. The ZetaChain team decided to add this `index` onto the `gasLimit` for each cross-chain transaction, which effectively turns the `gasLimit` into a unique identifier for each emitted event. Their reasoning behind this approach (as opposed to just adding the `index` to the `MsgVoteOnObservedInboundTx` message) was to not unnecessarily add another parameter to the `MsgVoteOnObservedInboundTx` message just for the purpose of computing a unique hash digest.

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

    // Bump gasLimit by event index (which is very unlikely to be larger
    than 1000) to always have different ZetaSent events msgs.
    msg := zetacoretypes.NewMsgSendVoter("", emittingContract.Hex(),
    senderChain.ChainId, txOrigin, toAddr, receiverChain.ChainId, amount,
    "", event.Raw.TxHash.String(), event.Raw.BlockNumber,
    90000+uint64(event.Raw.Index), common.CoinType_Zeta, "")
    // [ ... ]
}
```

3.7 — Missing authentication when adding node keys

This finding was fixed by removing the public `SetNodeKeys` message handler completely. This prevents anyone from supplying a public key to be used for TSS signing.

3.8 — Missing `nil` check when parsing client event

This finding was not fixed. See finding 3.2 of this report for more details.

3.9 — Case-sensitive address check allows for double signing

This finding was fixed by completely removing the `IsDuplicateSigner()` function that was responsible for the case-sensitive address checking.

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

This finding was fixed by adding panic handlers to the Bitcoin client's `fetchUTXOs()` and `TryProcessOutTx()` functions.

```
func (ob *BitcoinChainClient) fetchUTXOS() error {
    defer func() {
        if err := recover(); err != nil {
            ob.logger.WatchUTXOS.Error().Msgf("BTC fetchUTXOS: caught
    panic error: %v", err)
        }
    }()

    // [ ... ]
}

func (signer *BTCSigner) TryProcessOutTx(send *types.CrossChainTx,
    outTxMan *OutTxProcessorManager, outTxID string, chainclient
    ChainClient, zetaBridge *ZetaCoreBridge, height uint64) {
    defer func() {
        if err := recover(); err != nil {
            signer.logger.Error().Msgf("BTC TryProcessOutTx: %s, caught
    panic error: %v", send.Index, err)
        }
    }()

    // [ ... ]
}
```

3.11 — Ethermint Ante handler bypass

This finding was partially fixed, but another bypass was found due to the inclusion of the x/group module. See finding 3.1 of this report for more details.

3.12 — Unbonded validators prevent the TSS vote from passing

This finding was fixed by refactoring the `CreateTSSVoter()` function. It now only allows authorized node accounts (which are created in genesis) to vote on creating a new TSS key.

3.13 — Code maturity

This finding was acknowledged and partially fixed by removing the commented out code. The remaining FIXMEs and TODOs will continue to be worked on to improve the maturity of the code.

4.2 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#L69: VoteOnObservedOutboundTx` — When fetching the current TSS with `k.GetTSS(ctx)`, the returned `found` boolean is not checked. Even though it is almost always going to exist, it would still be best to check the boolean and return an error instead of panicking on a `nil` value.

5 Audit Results

At the time of our audit, the audited code was not deployed to production.

During our assessment on the scoped ZetaChain modules, we discovered four findings. No critical issues were found. Two were of high impact, one was of medium impact, and the remaining finding was informational in nature. ZetaChain acknowledged all findings and implemented fixes.

5.1 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.