Zellic

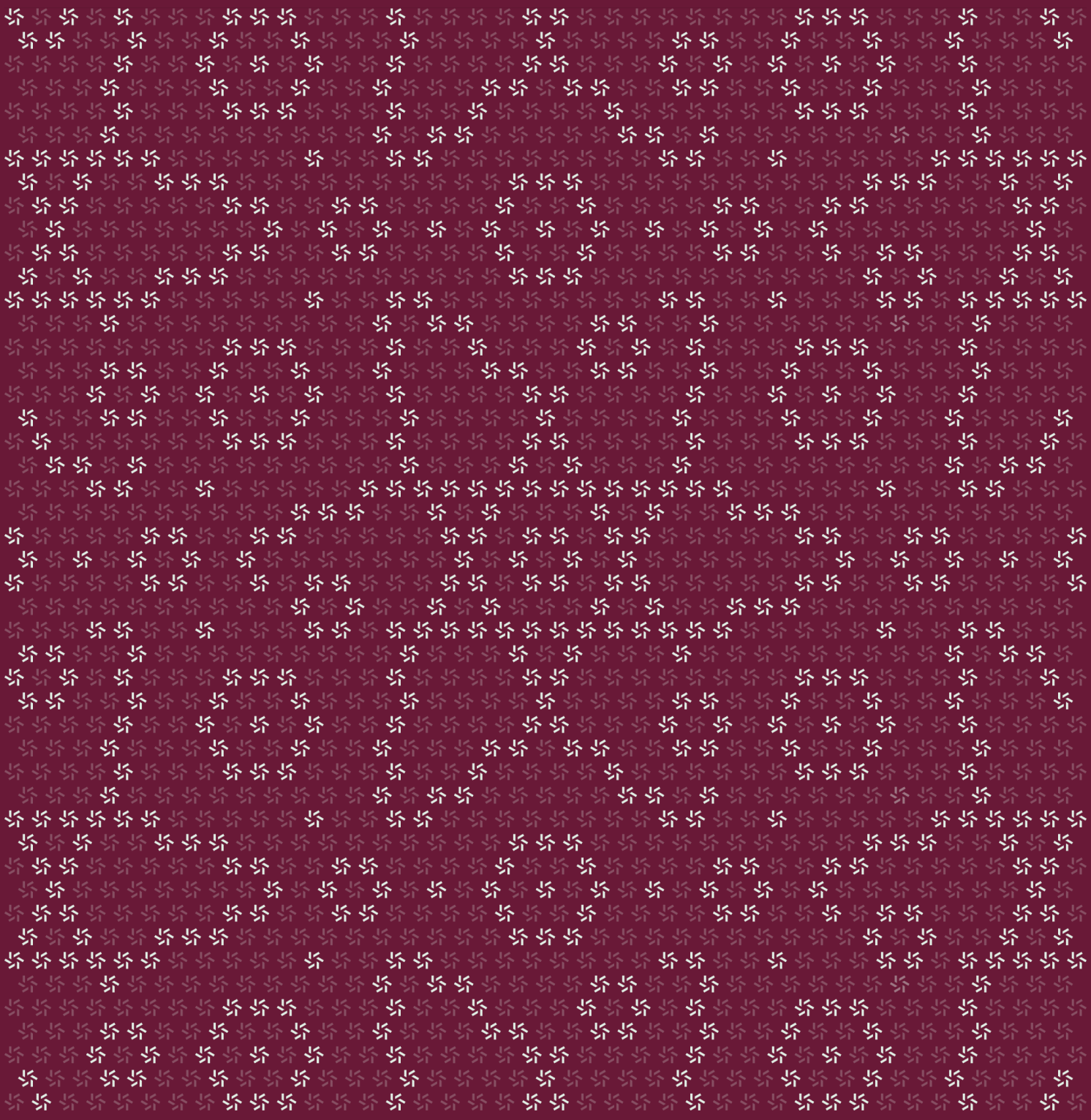
Prepared for
Tad Tobar
ZetaChain

Prepared by
Syed Faraz Abrar
Ayaz Mammadov
Zellic

November 14, 2023

ZetaChain

Smart Contract Security Assessment



Contents

About Zellic **4**

1. Executive Summary **4**

1.1. Goals of the Assessment 5

1.2. Non-goals and Limitations 5

1.3. Results 5

2. Introduction **6**

2.1. About ZetaChain 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. Multiple events in the same TX cause loss of funds 11

3.2. ZRC-20 paused status can be bypassed 14

3.3. No slippage limit set in Uniswap swap 19

3.4. Median gas-price threshold 21

3.5. ZetaChain pays gas costs for EVM-to-zEVM transfers 23

3.6. Ordering of steps for TSS funds migration is not enforced 25

3.7. The `MsgDeployFungibleCoinZRC20` message overwrites current ZRC-20 mapping 27

4. Discussion **28**

4.1. Lenient slash behaviour 29

4.2. Gas stability pool 30

5. Threat Model **30**

5.1. Module: x/crosschain 31

5.2. Module: x/fungible 32

5.3. Module: x/observer 35

6. Assessment Results **37**

6.1. Disclaimer 38

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 October 23rd to November 10th, 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:

- Could the new group 1 type policy admin group be able to exploit the protocol?
- Could maliciously crafted proofs be accepted by the new permissionless transaction validation code?
- Could an observer receive an incorrect amount of rewards due to logic errors in the new rewards distribution code?
- Could an attacker abuse the new gas stability pool mechanism to steal money from the protocol?
- Could an attacker abuse any of the newly implemented Messages to attack the protocol?

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 seven findings. One critical issue was found. One was of high impact, three were of medium impact, and two were of low impact.

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	1
High	1
Medium	3
Low	2
Informational	0

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 codebase 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, upgradability 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.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped modules itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

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: `161406eb7a936c6a9b7081f26017ce8545a0accb`
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 six 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

Ayaz Mammadov
Engineer
ayaz@zellic.io

2.5. Project Timeline

October 23, 2023	Kick-off call
October 23, 2023	Start of primary review period
November 13, 2023	End of primary review period

3. Detailed Findings

3.1. Multiple events in the same TX cause loss of funds

Target	EVMClient		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

When a user sends ZETA or ERC-20 tokens from an EVM chain to another chain, events are emitted on the EVM from specific contracts that are picked up by the ZetaChain observers. The observers then execute the `VoteOnObservedInboundTx()` message on ZetaChain with the details of the event. This is done through a ballot system, where the intention is for each event to have its own ballot.

The ballots are uniquely identified by their index, and the index for each ballot in this case is a hash of the `VoteOnObservedInboundTx` message (with the `Creator` set to `""`). If a ballot is not found for a specific hash, then it is created.

```
func (k msgServer) VoteOnObservedInboundTx(goCtx context.Context, msg
    *types.MsgVoteOnObservedInboundTx)
(/*...*/) {
    // [ ... ]

    index := msg.Digest()
    // Add votes and Set Ballot
    // GetBallot checks against the supported chains list before
    querying for Ballot
    ballot, isNew, err := k.zetaObserverKeeper.FindBallot(ctx, index,
    observationChain, observationType)
    if err != nil {
        return nil, err
    }

    // [ ... ]
}
```

The `MsgVoteOnObservedInboundTx` structure is as follows:

```
type MsgVoteOnObservedInboundTx struct {
    Creator string
    Sender string
```

```
    SenderChainId int64
    Receiver string
    ReceiverChain int64
    Amount github_com_cosmos_cosmos_sdk_types.Uint
    Message string
    InTxHash string
    InBlockHeight uint64
    GasLimit uint64
    CoinType common.CoinType
    TxOrigin string
    Asset string
}
```

Impact

Looking closely at the above structure, if multiple events are emitted in the same transaction with the same `Receiver`, `Amount`, and `Message`, the hash for every event would be the same. These events are usually uniquely identified by their log index in the transaction logs.

This leads to an issue where a smart contract on the EVM might trigger multiple cross-chain transfers in the same transaction but only one ballot for the very first event in the transaction. All subsequent calls to `VoteOnObservedInboundTx()` for each event will lead to the same ballot being voted on, rather than a new ballot being created for each event.

This would cause the sender to lose access to their funds.

An easy way to trigger this bug on the local developer network is to deploy a smart contract like the following on EVM (note the code is incomplete):

```
contract Test {
    ERC20Custody custody =
        ERC20Custody(0xD28D6A0b8189305551a0A8bd247a6ECa9CE781Ca);
    IERC20 usdt = IERC20(0xff3135df4F2775f4091b81f4c7B6359CfA07862a);
    address deployer = 0xE5C5367B8224807Ac2207d350E60e1b6F27a7ecC;

    function runTest() external {
        usdt.approve(address(custody), type(uint256).max);
        usdt.transferFrom(deployer, address(this), 10e6);

        for (int i = 0; i < 10; i++) {
            custody.deposit(abi.encodePacked(deployer), usdt, 1e6, "");
        }
    }
}
```

Initially, the deployer on the zEVM will have zero USDTZRC20 tokens. After calling `runTest()`, the deployer should have 10e6 USDTZRC20 tokens if everything works out correctly. In reality, the deployer will only have 1e6 tokens, as all the events emitted will generate the same hash and thus only the very first event will be processed.

Recommendations

We reported a similar issue previously, which applied to zEVM-to-EVM communication. That issue was fixed by adding each event's log index to the gas limit within the `MsgVoteOnObservedInboundTx` message.

Ideally, the message should be modified to contain a log-index field, but if that is not possible, the `gasLimit + event.Raw.Index` fix should be applied in the `evm_client`, within the `GetInboundVoteMsgForZetaSentEvent()` and `GetInboundVoteMsgForDepositedEvent()`.

Remediation

ZetaChain implemented a fix for this issue in Pull Request #1372 ↗. Specifically, the event index was implemented for inbound tx digests.

3.2. ZRC-20 paused status can be bypassed

Target	Fungible		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `MsgUpdateZRC20PausedStatus` message is used to pause a ZRC-20 contract on the zEVM. Once a contract is paused, any transaction that calls the contract and generates an event (such as a `Transfer` or `Approval` event) will cause the transaction to revert instead.

This message is intended to be used if a ZRC-20 contract is hacked or funds are stolen in some other way. In such a scenario, a paused status would prevent the attacker from transferring stolen funds across chains.

The `CheckPausedZRC20()` function is used to determine whether a transaction contains logs from a paused ZRC-20 contract. This is located in x/fungible/keeper/evm_hooks.go:

```
// PostTxProcessing is a wrapper for calling the EVM PostTxProcessing
    hook on the module keeper
func (h EVMHooks) PostTxProcessing(ctx sdk.Context, _ core.Message,
    receipt *ethtypes.Receipt) error {
    return h.k.CheckPausedZRC20(ctx, receipt)
}

// CheckPausedZRC20 checks the events of the receipt
// if an event is emitted from a paused ZRC20 contract it will revert the
    transaction
func (k Keeper) CheckPausedZRC20(ctx sdk.Context, receipt
    *ethtypes.Receipt) error {
    // [ ... ]

    // get non-duplicated list of all addresses that emitted logs
    // [ ... ]

    // check if any of the addresses are from a paused ZRC20 contract
    for _, address := range addresses {
        fc, found := k.GetForeignCoins(ctx, address.Hex())
        if found && fc.Paused {
            return cosmoserrors.Wrap(types.ErrPausedZRC20,
    address.Hex())
```

```
        }
    }

    return nil
}
```

This function is called during the `PostTxProcessing()` function, which is a hook introduced in Ethermint. The Evmos docs ↗ explain how this works:

> PostTxProcessing is only called after an EVM transaction finished successfully and delegates the call to underlying hooks. If no hook has been registered, this function returns with a nil error.

These docs, however, are unfortunately quite misleading. The `PostTxProcessing()` function is actually not called after an EVM transaction finishes successfully. It is only called when the underlying Ethermint code calls the `ApplyTransaction()` function. This is evident from a comment in the official Evmos ERC-20 module:

```
// Note that the PostTxProcessing hook is only called by sending an EVM
// transaction that triggers `ApplyTransaction`. A cosmos tx with a
// `ConvertERC20` msg does not trigger the hook as it only calls
    `ApplyMessage`.
func (k Keeper) PostTxProcessing(
    ctx sdk.Context,
    _ core.Message,
    receipt *ethtypes.Receipt,
) error { /* ... */ }
```

Both the `ApplyTransaction()` and `ApplyMessage()` functions will commit any state changes to the blockchain. However, only `ApplyTransaction()` will trigger the `PostTxProcessing()` hook.

Impact

Ethermint exposes a `CallEVM()` function that allows a Cosmos chain to call a smart contract within the Ethermint EVM. This code was ported into the ZetaChain repo, and it is evident that this function uses `ApplyMessage()` instead of `ApplyTransaction()` to commit state changes:

```
func (k Keeper) CallEVM(
    // [ ... ]
) (*evmtypes.MsgEthereumTxResponse, error) {
    // [ ... ]

    resp, err := k.CallEVMWithData(ctx, from, &contract, data, commit,
    noEthereumTxEvent, value, gasLimit)
    if err != nil {
        return resp, cosmoserrors.Wrapf(err, "contract call failed:
    method '%s', contract '%s'", method, contract)
    }
    return resp, nil
}

func (k Keeper) CallEVMWithData(
    // [ ... ]
) (*evmtypes.MsgEthereumTxResponse, error) {
    // [ ... ]

    res, err := k.evmKeeper.ApplyMessage(ctx, msg,
    evmtypes.NewNoOpTracer(), commit)
    if err != nil {
        return nil, err
    }

    // [ ... ]
}
```

Normally, this is okay, as `CallEVM()` is intended to be used to initiate privileged smart contract calls. However, ZetaChain actually has a code path that uses `CallEVM()` to call a user-defined smart contract. This code path is triggered when the user takes the following steps:

1. User triggers an ERC-20 deposit from an EVM chain to the zEVM. This is done using the deployed ERC20Custody contract's `deposit()` function on the EVM.
2. The observers pick up the `Deposited` event and then initiate a call on ZetaChain to deposit the tokens.
3. If the address being deposited into is detected as a contract account, the contract's `onCrossChainCall()` function is called.

This `onCrossChainCall()` code path is reached through the following functions:

```
// Observers call VoteOnObservedInboundTx()

VoteOnObservedInboundTx() -> HandleEVMDeposit() ->
```

```
    ZRC20DepositAndCallContract()
```

```
func (k Keeper) ZRC20DepositAndCallContract(
    // [ ... ]
) (*evmtypes.MsgEthereumTxResponse, bool, error) {
    // [ ... ]

    // check if the receiver is a contract
    // if it is, then the hook onCrossChainCall() will be called
    // if not, the zrc20 are simply transferred to the receiver
    acc := k.evmKeeper.GetAccount(ctx, to)
    if acc != nil && acc.IsContract() {
        context := systemcontract.ZContext{
            Origin: from,
            Sender: eth.Address{},
            ChainID: big.NewInt(senderChain.ChainId),
        }
        res, err := k.DepositZRC20AndCallContract(ctx, context,
    ZRC20Contract, to, amount, data)
        return res, true, err
    }

    // [ ... ]
}

func (k Keeper) DepositZRC20AndCallContract(
    // [ ... ]
) (*evmtypes.MsgEthereumTxResponse, error) {
    // [ ... ]

    return k.CallEVM( /* ... */ )
}
```

As seen above, when ZetaChain calls the `onCrossChainCall()` function, it uses `CallEVM()`, meaning the `PostTxProcessing()` hooks will not be triggered.

The ZetaChain developers are aware of this because they manually make a call to the `ProcessLogs()` function, which is also called through the x/crosschain module's `PostTxProcessing()` hook to handle any events emitted by the ConnectorZEVM contract. These events are used to process cross-chain transfers:

```
func (k Keeper) HandleEVMDeposit( /* ... */ ) (bool, error) {
    // [ ... ]

    if msg.CoinType == common.CoinType_Zeta {
        // [ ... ]
```

```
    } else {
        // [ ... ]

        evmTxResponse, contractCall,
    err := k.fungibleKeeper.ZRC20DepositAndCallContract( /* ... */ )
        if err != nil {
            // [ ... ]
        }

        // non-empty msg.Message means this is a contract call;
    therefore the logs should be processed.
        // a withdrawal event in the logs could generate cctxs for
    outbound transactions.
        if !evmTxResponse.Failed() && contractCall {
            logs := evmtypes.LogsToEthereum(evmTxResponse.Logs)
            if len(logs) > 0 {
                // [ ... ]

                err = k.ProcessLogs(ctx, logs, to, txOrigin)
                // [ ... ]
            }
        }
    }
    return false, nil
}
```

However, since `CheckPausedZRC20()` is not called here, this allows an attacker to transfer paused ZRC-20 tokens out of the zEVM by triggering a zero-amount token deposit from a supported EVM chain.

Recommendations

We recommend patching `CallEVMWithData()` to call the `PostTxProcessing()` hooks, similar to how the Teleport Network did here ↗.

Remediation

ZetaChain implemented a fix for this issue in Pull Request #31 ↗ by adding logic in the cosmos message handler that checks if the coin is paused. However, it is important to note that this check should be replicated for every instance of `CallEvm` and similar methods. We also recommend that this behavior be documented for future engineers when developing new handlers.

3.3. No slippage limit set in Uniswap swap

Target	system_contract		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

When performing swaps to pay for gas with ZRC-20s, the swap is executed with no minimum out parameter set. This can be seen in the invocation of the `CallEvm` function with the parameter `AmountInMax` set to `BigIntZero`.

```
func (k *Keeper) CallUniswapV2RouterSwapExactTokensForTokens(
    ctx sdk.Context,
    sender ethcommon.Address,
    to ethcommon.Address,
    amountIn *big.Int,
    inZRC4,
    outZRC4 ethcommon.Address,
    noEthereumTxEvent bool,
) (ret []*big.Int, err error) {
    ...
    res, err := k.CallEVM(
        ctx,
        *routerABI,
        sender,
        routerAddress,
        BigIntZero,
        big.NewInt(1000_000),
        true,
        noEthereumTxEvent,
        "swapExactTokensForTokens",
        amountIn,
        BigIntZero,
        []ethcommon.Address{inZRC4, wzetaAddr, outZRC4},
        to,
        big.NewInt(1e17),
    )
    ...
}
```

Impact

Users could be subject to sandwich attacks and as a result take on unfavorable prices, resulting in a loss of funds.

Recommendations

Add an external parameter that a user can specify `amountOutMin` to prevent such attacks.

Remediation

This issue has been acknowledged by ZetaChain.

The ZetaChain team stated that:

> We acknowledged the finding as an issue as it might cause the user to pay more fees than what needed for the actual gas of the transaction. However, we decided to not apply the recommendation, setting a AmountOutMin . The reason is if the swap of tokens fails during the gas payment for the revert transaction of the CCTX, the CCTX will become aborted. Aborting the CCTX would cause to refund the user on ZetaChain with the remaining amount. Although it prevents the slippage loss, we deemed it as a worse UX since the user would need to trigger a transaction to get the funds back on the source chain.

and that:

> We decided to keep the recommendation in our backlog for implementation in the future. As a remediation we plan to implement off-chain monitoring in order to check the liquidity Uniswap pools for gas payment.

3.4. Median gas-price threshold

Target	keeper_gas_price		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The price of gas is determined by taking a median of posted gas prices; however, there is no minimum threshold of posted prices. As a consequence, if there are few prices posted, the price could be manipulated.

```
// Only observer validators are authorized to broadcast this message.
func (k msgServer) GasPriceVoter(goCtx context.Context, msg
    *types.MsgGasPriceVoter) (*types.MsgGasPriceVoterResponse, error) {
        ...
        // recompute the median gas price
        mi := medianOfArray(gasPrice.Prices)
        // #nosec G701 always positive
        ...
}
```

Impact

An early malicious observer could be the first one to post and could post a completely manipulated price, which would then be considered the median, and they could quickly execute a transaction with said manipulated gas price for their own gain.

Recommendations

Set a minimum threshold of posted prices before setting the `gas_price`.

Remediation

This issue has been acknowledged by ZetaChain.

The ZetaChain team stated that:

We acknowledged the finding as an issue. However, we think the impacted is limited because the observer set is currently permissioned and the exploitation of the issue is limited since the sender of the transaction with the manipulated gas price would still need to pay for the gas on ZetaChain. We decided to not implement a remediation for the time being but we documented the issue in our backlog.

3.5. ZetaChain pays gas costs for EVM-to-zEVM transfers

Target	Fungible		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	Medium

Description

When an EVM-to-zEVM cross-chain transfer is initiated, the user initiating the transfer only pays gas fees for the EVM transaction. ZetaChain incurs the gas costs for any transaction initiated on the zEVM to finish the cross-chain transfer. This seems to be intended by ZetaChain.

However, when a user sends an ERC-20 token from the EVM to the zEVM, they are able to send the tokens to a smart contract on the zEVM. When this happens, the smart contract's `onCrossChainCall()` function is called by ZetaChain. In this scenario, ZetaChain sets the gas limit to `ZEVMGasLimitDepositAndCall`, which is one million. Again, it seems that this is intended by ZetaChain.

Impact

The issue here is that a malicious user can force ZetaChain to pay for very gas-intensive operations on the zEVM. The one-million gas limit applies per deposit, but a user is able to emit multiple `Deposited` events in a single transaction. If this is done on an EVM chain with very low gas fees like Polygon, the user might pay $1 in gas fees but then force ZetaChain to pay magnitudes more in gas fees on the zEVM.

Note that because of Finding 3.1. ↗, each event emitted will need to have a unique `message`, but that is easy to control. We note this in case the ZetaChain team would like to test this out.

We mark the impact as Medium because ZetaChain themselves control the total supply of the native ZETA token, so the costs they incur may not be as much of a concern.

Recommendations

Unfortunately, we do not think there is a good fix for this bug. Our recommendation is to either remove the ability for users to make cross-chain smart contract calls or revamp the code in order to let the user pay for the gas costs for the cross-chain smart contract call.

Remediation

This issue has been acknowledged by ZetaChain.

The ZetaChain team responded that they plan to fix it with the following statement:

> We acknowledged the finding as an issue. As a remediation, we are planning to limit the number of deposits that can be initiated from a single transaction on a source chain Each deposit can call a smart contract on zEVM with hard coded cap of gas limit 1M. External transaction that creates multiple deposit events will be rejected and refunded. There is currently no PR for the remediation.

3.6. Ordering of steps for TSS funds migration is not enforced

Target	Crosschain		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Low

Description

When a new TSS public key is successfully voted on, the old TSS public key is still in effect until `MsgUpdateTssAddress` is executed.

In order to migrate funds from the old TSS account to the new TSS account, ZetaChain introduced the `MsgMigrateTssFunds` message. This message will migrate funds from the current TSS account to the newest TSS account. After this step, `MsgUpdateTssAddress` can be executed to start using the new TSS account for cross-chain transactions.

Overall, the ordering of steps for this is as follows:

1. Observers use `MsgCreateTSSVoter` to vote on the new TSS public key.
2. After a successful vote, `MsgMigrateTssFunds` is used to migrate funds to the new TSS account.
3. Finally, `MsgUpdateTssAddress` is used to start using the new TSS account.

Impact

The issue here is that nothing enforces the above ordering of steps. The `MsgUpdateTssAddress` can be executed prior to `MsgMigrateTssFunds`, and when this happens, the `MsgMigrateTssFunds` would attempt to migrate funds from the current (new) TSS to itself, which would not work as intended.

ZetaChain will likely have another way to migrate the funds. However, the implementation of this message suggests that they are attempting to make this the only way to migrate the funds; therefore, we mark the impact as “Low”.

Recommendations

Implement a global flag of some sort to enforce that `MsgMigrateTssFunds` must be executed before `MsgUpdateTssAddress`.

Remediation

This was fixed in commit f34c26d3 ↗, by adding a key generation/migration process. Specifically one that enforces the order of operations such that a key migration cannot happen without the prerequisite steps. These steps have been outlined by the ZetaChain team as:

> The keygen process should follow the order Add node account → Generate new keys → Migrate Funds → Update TS

3.7. The `MsgDeployFungibleCoinZRC20` message overwrites current ZRC-20 mapping

Target	**Fungible**		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Low	**Impact**	Low

Description

The `MsgDeployFungibleCoinZRC20` message is used to deploy a new ZRC-20 contract and map it to an ERC-20 or gas token on a foreign chain.

In the ERC-20 token case, the code fails to check to ensure that the ERC-20 token contract address that is passed in (for the foreign chain) does not already have a mapping to a deployed ZRC-20 token contract.

In the gas-token case, the code fails to check to ensure that the foreign chain does not already have a ZRC-20 token contract mapped to it.

Impact

This issue would lead to an already mapped ZRC-20 contract being replaced by a newly deployed one, which would in turn brick the old contract as there currently exists no alternative method to modify the mapping.

Since this message requires a policy type 2 admin account (i.e., a multi-sig) to execute, we have decided that the likelihood of this occurring is low, and thus the impact is also “Low”.

Recommendations

In the gas-token case, implement a check in the message handler to ensure that the foreign chain’s gas token is not already mapped to a ZRC-20 token contract address.

In the ERC-20 token case, consider removing this functionality entirely, as it is handled by the `MsgWhitelistERC20` message already. Alternatively, consider adding a similar check to the above.

Remediation

ZetaChain implemented the suggested fix for this issue in Pull Request #13 ↗. The checks all follow the below structure.

```
// Check if gas coin already exists
_, found := k.GetGasCoinForForeignCoin(ctx, chainID)
if found {
    return ethcommon.Address{}, types.ErrForeignCoinAlreadyExist
}
```

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. Lenient slash behaviour

The reward system for observers is lenient. That is to say an observer is allowed to vote incorrectly on up to 50% of the votes, and they will still be rewarded; if they are wrong exactly 50% of the time, they get no rewards, but are not slashed; and ultimately if they get more than half of votes wrong, they are slashed.

While there is no security issue here, there is a lot of leeway for observers (e.g., in 1,000 votes, an observer is allowed to incorrectly vote on 499 of them). An alternative could be having a maximum number of incorrect guesses as a ratio of total guesses, depending on if this configuration is deemed insecure.

```
func DistributeObserverRewards(ctx sdk.Context, amount sdkmath.Int,
    keeper keeper.Keeper) error {
    ...
    if observerRewardUnits == 0 {
        finalDistributionList = append(finalDistributionList,
    &types.ObserverEmission{
            EmissionType: types.EmissionType_Slash,
            ObserverAddress: observerAddress.String(),
            Amount: sdkmath.ZeroInt(),
        })
        continue
    }
    if observerRewardUnits < 0 {
        slashAmount := keeper.GetParams(ctx).ObserverSlashAmount
        keeper.SlashObserverEmission(ctx, observerAddress.String(),
    slashAmount)
        finalDistributionList = append(finalDistributionList,
    &types.ObserverEmission{
            EmissionType: types.EmissionType_Slash,
            ObserverAddress: observerAddress.String(),
            Amount: slashAmount,
        })
        continue
    }
    // Defensive check
    if rewardPerUnit.GT(sdk.ZeroInt()) {
        rewardAmount :=
    rewardPerUnit.Mul(sdkmath.NewInt(observerRewardUnits))
```

```
            keeper.AddObserverEmission(ctx, observerAddress.String(),
        rewardAmount)
            finalDistributionList = append(finalDistributionList,
        &types.ObserverEmission{
                EmissionType: types.EmissionType_Rewards,
                ObserverAddress: observerAddress.String(),
                Amount: rewardAmount,
            })
        }
        ...
```

4.2. Gas stability pool

The gas stability pool is a mechanism that allows for the increase in gas fee (priority fee) if the CCTXs posted by the observers are not included in blocks (gas fee was too low). This is achieved by having a pool that is filled with surplus remains of CCTXs. The funding happens in the CCTX outbound voting in the finalizing vote.

```
func (k msgServer) VoteOnObservedOutboundTx(goCtx context.Context, msg
    *types.MsgVoteOnObservedOutboundTx)
    (*types.MsgVoteOnObservedOutboundTxResponse, error) {
    ...
    // Fund the gas stability pool with the remaining funds
    if err := k.FundGasStabilityPoolFromRemainingFees(ctx,
    *cctx.GetCurrentOutTxParam(), msg.OutTxChain); err != nil {
        log.Error().Msgf(
            "VoteOnObservedOutboundTx: CCTX: %s Can't fund the gas
    stability pool with remaining fees %s", cctx.Index, err.Error(),
        )
    }
    ...
}
```

The system to check whether or not transactions have gone through is the method `IterateAndUpdateCctxGasPrice` that is executed at the beginning of every block. It will check every epoch block and every interval second to query all `pendingCctxs` and increase gas prices.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the modules 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. Module: x/crosschain

Message: `MigrateTssFunds`

The `MigrateTssFunds` message handler is used to migrate TSS funds from the current TSS to the newest TSS in the store. New TSS details are added to the chain through the `CreateTSSVoter` message.

The code first ensures that that the caller is a policy type 2 admin account (i.e., a multi-sig). It then does the following checks:

1. The chain must have inbound disabled.
2. There can be no pending CCTXs for the chain where the TSS funds are being migrated.

It will then proceed to create and broadcast a CCTX to the specified chain that will migrate the TSS funds as required.

It is important to note that this function must be called after the vote in `CreateTSSVoter` passes but before `UpdateTssAddress` is executed. This is detailed as a finding here: Finding 3.6. ↗.

Message: `AddToInTxTracker`

The `AddToInTxTracker` message handler is used to add an inbound transaction into the inbound transaction tracker. Non-observer and non-admin accounts are able to execute this message as long as they provide a proof for the transaction.

The code first ensures that either the caller of the handler is a policy type 1 admin account or an observer account or that a proof was supplied with the message.

If a proof is supplied, the proof itself is verified. The verification process differs between Bitcoin and Ethereum; however, in either case, external libraries are used, so we decided to treat that code as a black box.

Finally, the transaction is added into the store if it has successfully been proved or if a policy type 1 admin account or observer account executed the message.

Message: `UpdateTssAddress`

The `UpdateTssAddress` message is used to update the stored TSS pubkey that is currently being used to the most recently voted TSS pubkey.

It first ensures that the caller is a policy type 2 admin account. Then it checks to ensure the TSS pubkey that is being set has already been generated (i.e., is in the store). Finally, it sets the TSS pubkey into the store for all foreign chains.

Note that the `MigrateTssFunds` message must be executed first prior to `UpdateTssAddress`. For more details, see the finding here: Finding 3.6. ↗.

5.2. Module: x/fungible

Message: `UpdateZRC20PausedStatus`

The `UpdateZRC20PausedStatus` message handler is used to pause and unpause ZRC-20 token contracts in the zEVM. Policy type 1 admin accounts are able to pause tokens, but unpausing requires a policy type 2 admin account (i.e., a multi-sig).

The code first ensures that the account executing the message has the required permissions. It then iterates through all foreign coins and modifies the pause status of the coin.

The pause status itself is checked in the Fungible module's `PostTxProcessing()` hook.

We found an issue that allows the pause status to be bypassed (i.e., an attacker is able to interact freely with a paused ZRC-20 token contract). The finding is detailed here: Finding 3.2. ↗.

Message: `UpdateZRC20WithdrawFee`

The `UpdateZRC20WithdrawFee` message handler is used to update the withdrawal fees and gas limits when withdrawing ZRC-20 coins. Only policy type 2 admin accounts are able to call this handler (i.e., a multi-sig).

The code ensures that the ZRC-20 exists and that a foreign coin is found for said ZRC-20 address (mapping must exist). Then the relevant functions are called to call the zEVM methods to update the respective properties.

The only thing to note is that there are no gas limits for the calls to the zEVM methods. This is not a security issue as ZRC-20 tokens can only be whitelisted/deployed by the admin team; therefore, they cannot endlessly consume gas. However, this is a future consideration in case features are added for users to add their own custom ZRC-20s.

Message: `UpdateSystemContract`

The `UpdateZRC20WithdrawFee` message handler is used to update the system contract. This is achieved by looping over all ZRC-20s updating the system contract address to the new system contract address and by setting the native gas coins and gas pool in the new system contracts.

Only policy type 2 admin accounts are able to call this handler (i.e., a multi-sig).

Then, global state variables are updated. The only thing to note is that if the system contract is not found, it will not revert; instead, the default global `systemContract` type is set to "". This should be kept in consideration for the future.

Message: `UpdateContractBytecode`

The `UpdateContractBytecode` message handler is used to update the bytecode of a contract. Only policy type 2 admin accounts are able to call this handler (i.e., a multi-sig).

The bytecode is sourced from an existing contract deployed on the chain. There are checks to ensure that the addresses exist.

Then, a check is enacted to ensure that the contract being updated is a delpoyed ZRC-20 or is the wzeta connect contract.

Message: `UpdateZRC20LiquidityCap`

The `UpdateZRC20LiquidityCap` message handler is used to update the liquidity cap of coins. Only policy type 2 admin accounts are able to call this handler (i.e., a multi-sig).

Apart from `ValidateBasic` checks, the ZRC-20 address is used to fetch the coin out of the foreign coin list, update it, and set it back into the foreign coins list.

This message emits no events and logs no errors if the coin is not found.

Message: `RemoveForeignCoins`

The `RemoveForeignCoins` message handler is used to remove ZRC-20s from the foreign coins list. Only policy type 2 admin accounts are able to call this handler (i.e., a multi-sig).

It checks if the coin is in the `ForeignCoin` mapping, and if it is, it removes it. Otherwise, it reverts.

This message does not emit any events if it succeeds.

Message: `DeployFungibleZRC20`

The `DeployFungibleZRC20` message handler is used to deploy a fungible ZRC-20. Only policy type 2 admin accounts are able to call this handler (i.e., a multi-sig).

It takes several parameters, such as the coin data, what type of coin it is (e.g., regular ZRC-20, the gas coin, native coin), and the gas limit.

It then deploys the contract. Depending on the type of coin deployed, other properties are changed. For example, if the coin is the `CoinGas`, that means that the system contract has to be updated to account for calculations when gas is involved.

An `EventZRC20Deployed` is emitted whenever this transaction is successfully executed.

Message: `WhitelistERC20`

The `WhitelistERC20` message handler is used to whitelist ERC-20 tokens such that they can be deposited and withdrawn. Only policy type 1 admins are able to call this message.

This message adds an ERC-20 address to the whitelisted coins in the zEVM and EVM contracts and then deploys a respective ZRC-20 and sends out a `CoinType_Cmd`, which is an admin message accepted by all observers and agreed upon to sign a TSS message. This is required as the `whitelist` function is `onlyTSS`.

We discovered an issue with this function — specifically how it interacts with `DeployFungibleZRC20`. Even if an a deployed ZRC-20 address is provided, it is not whitelisted. Instead, a new ZRC-20 contract is deployed and whitelisted.

Message: `AddToOutTxTracker`

The `AddToOutTxTracker` message handler is used add outgoing CCTXs to the TX tracker list for voting. Only observers and policy type 1 admin accounts are allowed to post arbitrary transactions to the tracker list. Any user can post transactions not in the tracker list if they have the proofs for said transactions (Ethereum/Bitcoin header). However, the transactions have to be in a block header that was previously agreed upon with a `MsgAddBlockHeader`.

The function `verifyProof` is called where the submitted data is used to fetch a block header from a store that has already passed the finalization process. Then, the relevant proof proving framework is called (geth/bitcoin_spv).

The `verifyProof` function verifies that a TX has been included in a header; however, it does not prove that the TX is related to ZetaChain. Therefore, another check `verifyOutTxBody` is ran that verifies that the TX sender/target is ZetaChain.

One issue was noticed when a new TxTracker is created via a proof: it immediately returns and does not set the proven property to true in the `hashlist`.

Message: `RemoveFromOutTxTracker`

The `RemoveFromOutTxTracker` message handler is used to remove transcations from the TxTracker. Only policy type 1 admin accounts are able to call this handler. The TX is deleted from the store; no event is emitted.

5.3. Module: x/observer

Message: `AddObserver`

The `AddObserver` message handler is used to add a new observer to the system. It first ensures that the caller is an admin policy group 2 account, which is a multi-sig. It then disables inbound messages and either

1. adds a new node account for the observer and resets the keygen for the TSS, OR
2. adds the observer address into all observer lists.

The functionality that it triggers depends on the `AddNodeAccountOnly` boolean flag in the message.

Message: `AddBlockHeader`

The `AddBlockHeader` message handler is used to add a new block header into the store. It initiates ballot voting for the block header in question, using the hash of the message as the ballot index.

Once the block header has been sufficiently voted on, it ensures the following:

1. The block header does not already exist in the store.
2. The block header timestamp is valid.
    - For Ethereum, no validation is necessary.
    - For Bitcoin, it checks the timestamp precision and ensures it is not more than two hours into the future compared to ZetaChain's time.
3. The block header's parent hash must not be `nil`.

It then finally inserts the new block header into the store.

Message: `UpdateCrosschainFlags`

The `UpdateCrosschainFlags` message is used to enable cross-chain related flags. These flags are

1. `IsInboundEnabled`, which determines whether inbound transactions are enabled
2. `IsOutboundEnabled`, which determines whether outbound transactions are enabled
3. `GasPriceIncreaseFlags`, which determines how much to increase the gas price by for long-term pending transactions
4. `BlockHeaderVerificationFlags`, which determines whether block-header transaction proofs are able to be verified for Ethereum and Bitcoin

A policy type 1 admin is able to modify the `BlockHeaderVerificationFlags` as well as disable inbound and outbound communication. Reenabling inbound or outbound communication and modifying the `GasPriceIncreaseFlags` require a policy type 2 admin account.

Message: `UpdateCoreParams`

The `UpdateCoreParams` message is used to update the core parameters for a specific foreign chain. The parameters are listed below:

```
type CoreParams struct {
    ConfirmationCount uint64
    GasPriceTicker uint64
    InTxTicker uint64
    OutTxTicker uint64
    WatchUtxoTicker uint64
    ZetaTokenContractAddress string
    ConnectorContractAddress string
    Erc20CustodyContractAddress string
    ChainId int64
    OutboundTxScheduleInterval int64
    OutboundTxScheduleLookahead int64
}
```

The chain that is being updated must be supported by ZetaChain, and the caller of the handler must be a policy type 2 admin account.

Message: `UpdateKeygen`

The `UpdateKeygen` message is used to update the block height of the keygen. The new block height must be at least 10 blocks ahead of the current block number in ZetaChain.

The code first ensures that this message is being executed by a policy type 1 admin account. It will then fetch the current keygen and add the pubkeys of all node accounts to the keygen. Finally, it updates the block height of the keygen, sets the status to “Pending”, and then sets the keygen into the store.

The “Pending” status is required to vote on a new TSS public key through the `CreateTSSVoter` message.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to production.

During our assessment on the scoped ZetaChain modules, we discovered seven findings. One critical issue was found. One was of high impact, three were of medium impact, and two were of low impact. 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 version reviewed during 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. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

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.