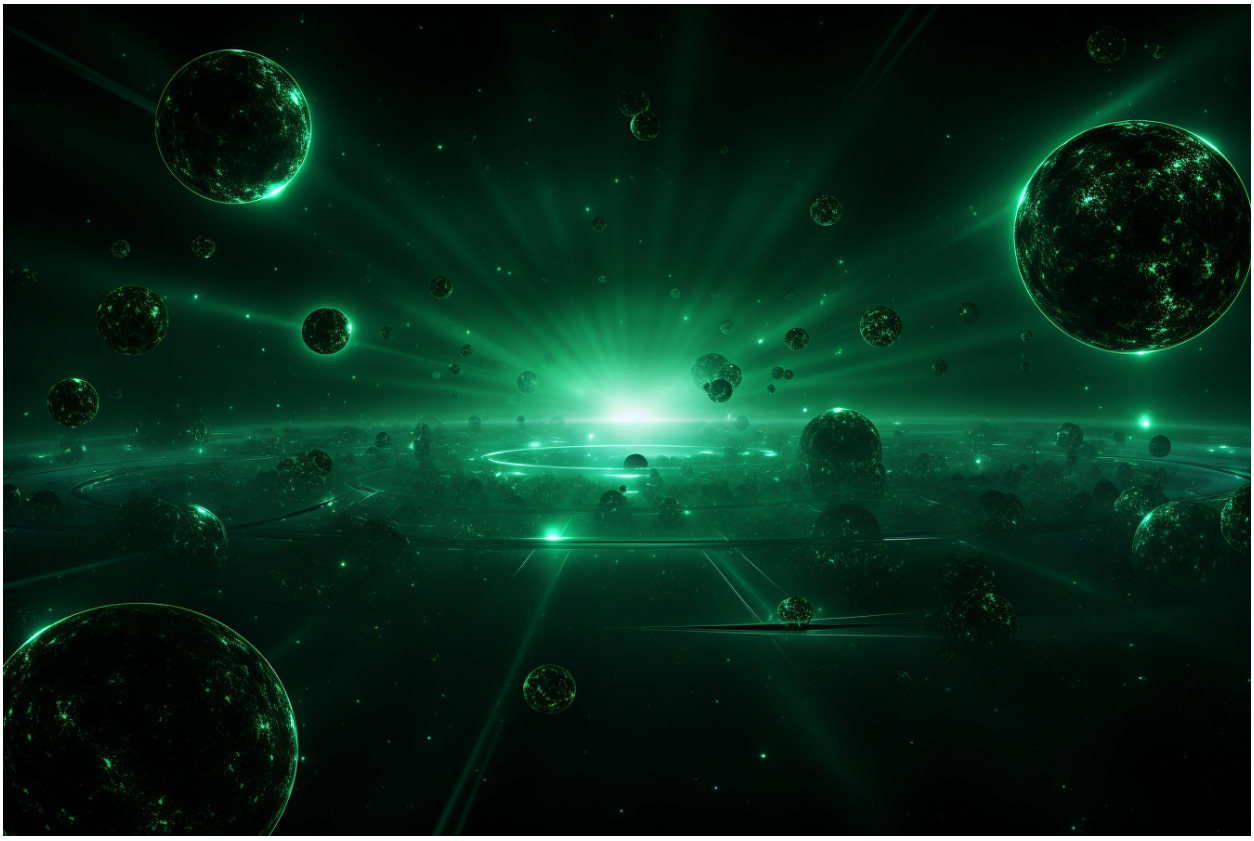
Collection & Classes of Vulnerabilities



In Project 101 and Testing 101 we discussed similarities, differences and architectural considerations that should be incorporated when conducting a security review of ZetaChain and/or other Cosmos ecosystem projects. In this section, we will talk about specific vulnerabilities and audit findings that have been reported in relation to ZetaChain and Cosmos projects. Furthermore, we will assign these vulnerabilities into classes where appropriate.

List of Past Audit Reports

- PeckShield Audit - Smart Contract - October 2022
- Veridise Audit - Smart Contract - October 2022
- QuantumBrief Audit - Contract - October 2022
- Zellic Audit - Contract June 2023
- Halborn - Smart contract March 2023
- Halborn - Deinal of service testing - July 2023
- Halborn - Zetanode Cosmos SDK - March 2023

Classes of Vulnerability

Please note that these classes of vulnerability are based on the ZetaChain Immunefi bug bounty platform (https://immunefi.com/bounty/zetachain)

- Protocol insolvency
- Cryptographic flaws that could result in leaking private key or allow forging signatures
  -Compromise of the “TSS Address” that leads to unauthorized transactions or complete private key exposure
- Unintended minting of ZETA token
- Loss of user funds by permanent freezing or direct theft
- Unintended permanent chain split requiring hard fork (network partition requiring hard fork)
- Tamper / manipulate blockchain history to add, invalidate, or change past transactions
- Temporary Consensus Failures
- Temporary Chain partition
- Cheap Denial of Service of Critical Functions of the blockchain
- Temporary freezing of funds

- High computation/load to zetaclientd & zetacored
- DoS to minority of validators
- Limited impact of denial of service

Audit Reports

Alpha Alert! The most recent publicly accessible audit for ZetaChain is the "Halborn's ZetaNode Audit". The Zeta team is diligently addressing issues identified in this audit.

- You can view the progress on specific resolutions in this pull request.
- To get an overview of ongoing issues labeled as 'bug', visit the ZetaChain node's issues page. By filtering for the 'bug' label, auditors can gain valuable insights into past bugs and can better determine areas of focus for testing.

To further streamline auditors knowledge of previous audit findings and potential areas of vulnerability, we have compiled previous public ZetaChain audit findings into this document.

Category of Vulnerability - Peckshield July 2022 - Deficient Protocol Contracts

Accomodation of Non-ERC20-Compliant Tokens

Relevant Contracts: `ZetaTokenConsumerUniV2`, `ZetaTokenConsumerUniV3`

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the transfer() routine and related idiosyncrasies from current widely-used token contracts. In particular, we use the popular token, i.e., ZRX, as our example. We show the related code snippet below. On its entry of transfer(), there is a check, i.e., if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]). If the check fails, it returns false. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: “Transfers _value amount of tokens to address _to, and

MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."

```
64    function transfer(address _to, uint _value) returns (bool) {
65        //Default assumes totalSupply can't be over max (2^256 - 1).
66        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67            balances[msg.sender] -= _value;
68            balances[_to] += _value;
69            Transfer(msg.sender, _to, _value);
70            return true;
71        } else { return false; }
72    }
```

```
74    function transferFrom(address _from, address _to, uint _value) returns (bool) {
75        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
              balances[_to] + _value >= balances[_to]) {
76            balances[_to] += _value;
77            balances[_from] -= _value;
78            allowed[_from][msg.sender] -= _value;
79            Transfer(_from, _to, _value);
80            return true;
81        } else { return false; }
82    }
```

Because of that, a normal call to transfer() is suggested to use the safe version, i.e., safeTransfer(), In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of approve()/transferFrom() as well, i.e., safeApprove()/safeTransferFrom(). In the following, we show the getZetaFromToken() routine in the ZetaTokenConsumerUniV2 contract. If the USDT token is supported as inputToken, the unsafe version of IERC20(inputToken).transferFrom(msg.sender, address(this), inputTokenAmount) (line 73) may revert as there is no return value in theUSDT token contract's transfer()/transferFrom() implementation (but the IERC20 interface expects a return value)!

```
64      function getZetaFromToken(
65          address destinationAddress,
66          uint256 minAmountOut,
67          address inputToken,
68          uint256 inputTokenAmount
69      ) external override returns (uint256) {
70          if (destinationAddress == address(0)  inputToken == address(0)) revert
                InvalidAddress();
71          if (inputTokenAmount == 0) revert InputCantBeZero();
72
73          bool success = IERC20(inputToken).transferFrom(msg.sender, address(this),
                inputTokenAmount);
74          if (!success) revert ErrorGettingZeta();
75          success = IERC20(inputToken).approve(address(uniswapV2Router), inputTokenAmount)
                ;
76          if (!success) revert ErrorGettingZeta();
77          ...
78      }
```

Recommendation:

Accommodate the above-mentioned idiosyncrasy about ERC20-related approve()/transfer()/transferFrom(). For the safe-version of approve(), there is a need to safeApprove () twice: the first one reduces the allowance to 0 and the second one sets the new allowance.

Category of Vulnerability - Peckshield July 2022 - Deficient Protocol Contracts

Trust Issue of Admin Keys

Relevant Contracts: Multiple Contracts

In the ZetaChain protocol contracts, there is a privileged manager account (tssAddress) that plays a critical role in governing and regulating the system-wide operations (e.g., set new connectorAddress to mint additional tokens into circulation, etc.). Our analysis shows that the privileged account needs to be scrutinized. In the following, we show the representative functions potentially affected by the privileges of the privileged account. Specifically, the privileged functions in the ZetaNonEth contract allow for the tssAddress to update the current connectorAddress, which can mint additional tokens into circulation.

```
34      function updateTssAndConnectorAddresses(address tssAddress_, address
            connectorAddress_) external {
35          if (msg.sender != tssAddressUpdater && msg.sender != tssAddress) revert
                CallerIsNotTssOrUpdater(msg.sender);
36          if (tssAddress_ == address(0)  connectorAddress_ == address(0)) revert
                InvalidAddress();

38          tssAddress = tssAddress_;
39          connectorAddress = connectorAddress_;
40      }

42      function mint(
43          address mintee,
44          uint256 value,
45          bytes32 internalSendHash
46      ) external override {
47          /**
48           * @dev Only Connector can mint. Minting requires burning the equivalent amount
                    on another chain
49           */
50          if (msg.sender != connectorAddress) revert CallerIsNotConnector(msg.sender);

52          _mint(mintee, value);
```

```
54          emit Minted(mintee, value, internalSendHash);
55      }
```

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged account may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation:

Promptly transfer the privileged account to the intended DAO-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the in-tended trustless nature and high-quality distributed governance.

Category of Vulnerability - Verdise July 2022 - Deficient Smart Contract

Possible Event Reordering via Reentrancy in getEthFromZeta

Relevant Contracts: `ZetaTokenConsumerUniV3.strategy.sol`

The call to (bool sent,) = destinationAddress.call {value: amountOut}(""); can potentially reenter, allowing a malicious user to potentially manipulate the order of the event ZetaExchangedForEth. This could disrupt any application relying on the consistent order of events.

```
function getEthFromZeta(
  address destinationAddress,
  uint256 minAmountOut,
  uint256 zetaTokenAmount
) external override returns (uint256) {
  ...
  (bool sent, ) = destinationAddress.call{value: amountOut}("");
  if (!sent) revert ErrorSendingETH();
  emit ZetaExchangedForEth(zetaTokenAmount, amountOut);
  return amountOut;
}

```

Attack Scenario:

A malicious user can use the fallback function of destinationAddress to call getEthFromZeta again. The event from this second transaction will emit before the event from the first transaction. The following Foundry test demonstrates the attack. Please note that some details have been elided for space.

```
contract Reentrant is Initial {
  function testGetEthFromZeta() public {
    AttackHelper attackHelper = new AttackHelper(address(zetaV3), address(zetaToken))
    ;
    zetaTokenMint(address(attackHelper), 100 ether);
    vm.startPrank(address(attackHelper));
    zetaToken.approve(address(zetaV3), 20);
    vm.stopPrank();
    vm.startPrank(attacker);
    zetaToken.approve(address(zetaV3), 20);
    zetaV3.getEthFromZeta(address(attackHelper), 1, 1);
    vm.stopPrank();
  }
}

```

```
contract AttackHelper is Test {
  ZetaTokenConsumerUniV3 zetaV3;
  ZetaEth zetaToken;
  bool private firstTime = true;
  constructor(address zetaV3Addr, address zetaTokenAddr) {
    zetaV3 = ZetaTokenConsumerUniV3(payable(zetaV3Addr));
    zetaToken = ZetaEth(zetaTokenAddr);
  }
  receive() external payable {
    if (firstTime) {
      firstTime = false;
      zetaV3.getEthFromZeta(address(this), 10, 10);
    }
  }
}
```

Recommendation:

Move destinationAddress.call {value: amountOut } (""); after emit ZetaExchangedForEth(zetaTokenAmount, amountOut); to prevent reordering.

Category of Vulnerability - **Verdise July 2022** - **Deficient Smart Contract**

Possible Event Reordering via Reentrancy in getZetaFromToken

Relevant Contracts: `ZetaTokenConsumerUniV2/3.strategy.sol`

In the contracts ZetaTokenConsumerUniV2.strategy.sol and ZetaTokenConsumerUniV3.strategy.sol , the calls to external contracts bool success = IERC20(inputToken).transferFrom (msg.sender, address(this), inputTokenAmount); and success = IERC20(inputToken).approve (uniswapV2RouterAddress, inputTokenAmount); in getZetaFromToken can potentially reenter, al-lowing a malicious user to potentially manipulate the order of the event TokenExchangedForZeta.

This could disrupt any application relying on the consistent order of events.

```
function getZetaFromToken(
  address destinationAddress,
  uint256 minAmountOut,
  address inputToken,
  uint256 inputTokenAmount
) external override {
  if (destinationAddress == address(0) ||
      inputToken == address(0)) revert InvalidAddress();
  if (inputTokenAmount == 0) revert InputCantBeZero();
  bool success = IERC20(inputToken).transferFrom(msg.sender, address(this),
    inputTokenAmount);
  if (!success) revert ErrorGettingZeta();
  success = IERC20(inputToken).approve(uniswapV2RouterAddress, inputTokenAmount);
  if (!success) revert ErrorGettingZeta();
  ...
  emit TokenExchangedForZeta(inputToken, inputTokenAmount, amountOut);
}
```

Recommendation:

Either move the emitting of the event before the potentially reentrant calls or add a reentrancy guard.

Category of Vulnerability - **Verdise July 2022** - **Deficient Smart Contract**

Allow Updating of MAX_DEADLINE for TokenConsumers

Relevant Contracts: `ZetaTokenConsumerUniVx.strategy.sol` **(versions 2 and 3)**

The deadline value MAX_DEADLILNE is used in calls to Uniswap to set the deadline for computing a swap. MAX_DEADLINE is set to 100 by default. While this value is likely sufficiently large given past Ethereum block times (https://ycharts.com/indicators/ethereum_average_-block_time), the contracts offer no means of updating this value if the need/desire arises.

Recommendation:

Allow the maximum deadline to be adjusted. One possible option is to make the deadline configurable such that users can opt to pay lower gas fees in exchange for a risk of timed out transactions.

Category of Vulnerability - **Verdise July 2022** - **Deficient Smart Contract**

Use Separate ZetaToken Interface

Relevant Contracts: `ZetaConnector.non-eth.sol`

The ZetaToken interface is defined at the top of the ZetaConnector.non-eth.sol file. However, if the interface of ZetaToken.non-eth.sol is changed without changing this ZetaToken interface, a runtime error could occur.

```
interface ZetaToken is IERC20 {
  function burnFrom(address account, uint256 amount) external;
  function mint(
    address mintee,
    uint256 value,
    bytes32 internalSendHash
  ) external;
}
```

Recommendation:

Define ZetaToken as an interface in a separate file and ensure that ZetaNonEth inherits from ZetaToken.Furthermore, we recommend renaming ZetaToken to IZetaToken to make it obvious that it is an interface.

Category of Vulnerability - Verdise July 2022 - Gas Efficiency

Save Gas by Replacing keccak(constant) with a Constant

Relevant Contracts: ZetaInteractor.sol

The function _isValidChainId uses the value keccak256(new bytes(0)) , which is recomputed every time the function is called and thus costs extra gas on every call.

```
function _isValidChainId(uint256 chainId) internal view returns (bool) {
  return (keccak256(interactorsByChainId[chainId]) != keccak256(new bytes(0)));
}
```

Recommendation:

Replace keccack256(new bytes(0)) with a constant that is computed once for the contract.

Category of Vulnerability - Verdise July 2022 - Informational

Protocol Must Sanitize Inputs to onReceive and onRevert

Relevant Contracts: `ZetaTokenConnector.eth.sol`, `ZetaTokenConnector.non-eth.sol`

The calls to onRevert and onReceive can only be (successfully) called by the tssAddress, which represents the validators collectively. The validators forward messages from other blockchains to these two functions. We note that no input sanitization is done within the smart contracts to prevent a malicious user from sending arbitrary information here that will be forwarded by the validators. For instance, there is no built-in verification that the zetaTxSenderAddress for onRevert ever actually attempted a send in the first place, or even if they did, that the zetaValueAndGas matches.

```
function onReceive(
  bytes calldata zetaTxSenderAddress,
  uint256 sourceChainId,
  address destinationAddress,
  uint256 zetaValueAndGas,
  bytes calldata message,
  bytes32 internalSendHash
) external override whenNotPaused onlyTssAddress {}

function onRevert(
  address zetaTxSenderAddress,
  uint256 sourceChainId,
  bytes calldata destinationAddress,
  uint256 destinationChainId,
  uint256 zetaValueAndGas,
  bytes calldata message,
  bytes32 internalSendHash
) external override whenNotPaused onlyTssAddress {}
```

Recommendation:

Most of this sanitization can and should be performed at the protocol level. We suggest that developers carefully consider and document which inputs to these functions are “verified” by the protocol and what “verification” means in this context.

Category of Vulnerability - Verdise July 2022 - Informational

A malicious User Can Prevent ZetaRevert Event

Relevant Contracts: `ZetaTokenConsumerUniV3.strategy.sol`

In onRevert , the call to onZetaRevert is called on the contract at zetaTxSenderAddress which can be altered by the user. Therefore, a user can guarantee that the event ZetaRevert will
never be called for a send they initiated by making onZetaRevert revert (which would revert the
whole call to onRevert).

```
function onRevert(
  address zetaTxSenderAddress,
  uint256 sourceChainId,
  bytes calldata destinationAddress,
  uint256 destinationChainId,
  uint256 zetaValueAndGas,
  bytes calldata message,
  bytes32 internalSendHash
) external override whenNotPaused onlyTssAddress
{
  ...
  if (message.length > 0) {
    ZetaReceiver(zetaTxSenderAddress).onZetaRevert(...);
  }
  emit ZetaReverted(...);
}
```

```
function onRevert(
  address zetaTxSenderAddress,
  uint256 sourceChainId,
  bytes calldata destinationAddress,
  uint256 destinationChainId,
  uint256 zetaValueAndGas,
  bytes calldata message,
  bytes32 internalSendHash
) external override whenNotPaused onlyTssAddress
{
  ...
  if (message.length > 0) {
    ZetaReceiver(zetaTxSenderAddress).onZetaRevert(...);
  }
  emit ZetaReverted(...);
}
```

Recommendation:

Protocol designers should be aware that users can affect this event and either (1) make the protocol robust to such a possibility or (2) remove the ability for a malicious actor to make the call to onRevert revert.

Category of Vulnerability - **Verdise July 2022** - **Deficient Protocol / Informational**

Potential Protocol Concerns When onRevert Reverts

Relevant Contracts: `ZetaConnector.eth.sol`, `ZetaConnector.non-eth.sol`

On the event that onRevert reverts, we see three possibilities for how the protocol can handle this:

1. Ignore the fact it reverted and continue on.
2. Automatically rerun the transaction until it completes successfully (or some gas limit is
   hit).
3. Allow a user to request that a reverted onRevert be rerun.

Option (1) is not desirable as users may lose the ability to roll back important application logic
which should be rolled back on revert. Option (2) could be dangerous depending on who pays
for the gas — if the tssAddress holder must pay for gas costs, this could be very expensive.
Option (3) could have the same gas concerns from (2) if a user can continually request a rerun.

Recommendation:

We recommend the protocol avoid option (1) given the potential vulnerability for users. If option (2) is selected, the protocol should ensure that reruns are only performed while the user's gas costs can cover additional tries. For option (3), the protocol should perform the same gas checks as (2) as well as ensure that only the appropriate user can request a rerun.

Category of Vulnerability - **Verdise July 2022** - **Gas**

Save Gas by Declaring Addresses as Immutable and External Contracts as Local Variables

Relevant Contracts: `ZetaConnector.base.sol`, `ZetaInteractor.sol`

A number of variables which are immutable are not declared as so, which costs more gas than necessary when these functions are invoked. Below we give three optimizations in the code where gas can be saved by declaring variables immutable.

```
contract ZetaConnectorBase is ConnectorErrors, Pausable {
  address public zetaToken;
  ...
}
```

Recommendation:

Replace address public zetaToken with address public immutable zetaToken as the zetaToken address should be immutable.

```
abstract contract ZetaInteractor is Ownable, ZetaInteractorErrors {
  uint256 internal immutable currentChainId;
  ZetaConnector public connector;
  ...
}
```

Recommendation:

Replace ZetaConnector public connector with ZetaConnector public immutable connector as the ZetaConnector instance should be immutable.

```
contract ZetaTokenConsumerUniV2 is ZetaTokenConsumer, ZetaTokenConsumerUniV2Errors {
  uint256 internal constant MAX_DEADLINE = 100;
  address public uniswapV2RouterAddress;
  ...
  IUniswapV2Router02 internal uniswapV2Router;

  function getZetaFromEth(address destinationAddress, uint256 minAmountOut) external
    payable override {
    ...
    uint256[] memory amounts = uniswapV2Router.swapExactETHForTokens{...}(...);
    ...
  }

  function getZetaFromToken(...) external override {
    ...
    uint256[] memory amounts = uniswapV2Router.swapExactTokensForTokens(...);
    ...
  }

  function getEthFromZeta(...) external override {
    ...
    uint256[] memory amounts = uniswapV2Router.swapExactTokensForETH(...);
    ...
  }

  function getTokenFromZeta(...) external override {
    ...
    uint256[] memory amounts = uniswapV2Router.swapExactTokensForTokens(...);
    ...
  }
}
```

Recommendation:

Replace address public uniswapV2RouterAddress with address public immutable uniswapV2RouterAddress as the address uniswapV2RouterAddress should be immutable. Then, remove the declaration of uniswapV2Router and replace each occurence of it in the code with the dynamic cast, i.e., IUniswapRouter02(uniswapV2RouterAddress).

Category of Vulnerability - **QuantumBrief July 2022** - **Deficient Smart Contract**

The USDT contract doesn’t comply with the ERC20 standard

The USDT contract doesn’t comply with the ERC20 standard. The approve() and transfer() functions don’t return a boolean value indicating the success or failure of the

operation.

```
bool success = IERC20(inputToken).transferFrom(msg.sender, address(this),
inputTokenAmount);

if (!success) revert ErrorGettingZeta();
```

If inputToken is USDT, the transaction will always fail.

Recommendation:

Use safeTransferFrom() and safeApprove() from openZeppelin's SafeERC20 library.

Category of Vulnerability - QuantumBrief July 2022 - Informational

Interfaces should be named using the “I” character as the first letter by convention.

```
import "./interfaces/ConnectorErrors.sol";

import "./interfaces/ZetaInterfaces.sol";
```

Recommendation:

Add “I” before the interface name “IZetaInterfaces” and “IConnectorErrors”

Category of Vulnerability - QuantumBrief July 2022 - Gas

Variables that are set in constructor and never change like the zetaToken address should be immutable to save gas

```
    address public zetaToken;
```

Category of Vulnerability - Zellic April 2023 - Coding Mistakes

Any ZetaSent events are processed regardless of what contract emits them

Relevant Contracts: `evm_hooks.go`

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 PostTxPorcessing() 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.

Link to Audit Report:

https://drive.google.com/file/d/1TjLkNn9MobjGTupukJBxnpxr0DKvj_6V/view?usp=drive_link&utm_source=immunefi

Category of Vulnerability - **Zellic April 2023** - **Coding Mistakes**

Bonded validators can trigger reverts for successful transactions

Relevant Contracts: `keeper_out_tx_tracker.go`

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
```

Recommendation:

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.

Category of Vulnerability - Zellic April 2023 - Coding Mistakes

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

Relevant Contracts: `btc_signer.go`

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.

Category of Vulnerability - **Zellic April 2023** - **Coding Mistakes**

Race condition in Bitcoin client leads to double spend

Relevant Contracts: `btc_signer.go`

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 IsSendOutTxProces sed(). 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.

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.

Category of Vulnerability - **Zellic April 2023** - **Coding Mistakes**

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

Relevant Contracts: `btc_client.go`

Forks of length 1 (that is, a reorganization of one block in the blockchain) happen semi-frequently 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:

```
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.

Category of Vulnerability - **Zellic April 2023** - **Coding Mistakes**

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

Relevant Contracts: `evm_hooks.go`

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

Category of Vulnerability - Zellic April 2023 - Coding Mistakes

Missing authentication when adding node keys

Relevant Contracts: `keeper_node_account.go`

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 zetaclient'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.

Category of Vulnerability - **Zellic April 2023** - **Coding Mistakes**

Missing nil check when parsing client event

Relevant Contracts: `evm_client.go`

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.

Category of Vulnerability - Zellic April 2023 - Coding Mistakes

Case-sensitive address check allows for double signing

Relevant Contracts: `keeper_chain_nonces.go`

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.

Category of Vulnerability - **Zellic April 2023** - **Coding Mistakes**

No panic handler in Zetaclient may halt cross-chain communication

Relevant Contracts: `btc_signer.go`

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.

Category of Vulnerability - Zellic April 2023 - Coding Mistakes

Ethermint Ante handler bypass

Relevant Contracts: `app/ante/handler_options.go`

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.

Category of Vulnerability - **Zellic April 2023** - **Coding Mistakes**

Unbonded validators prevent the TSS vote from passing

Relevant Contracts: `keeper_tss_voter.go`

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.

Category of Vulnerability - **Zellic April 2023** - **Informational**

Code maturity

Relevant Contracts: Multiple Contracts

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.

Category of Vulnerability - **Halborn March 2023** - **Deficient Smart Contract**

Recieve Function is not Restricted to WETH

Relevant Contracts: `ConnectorZEVM.sol#L74`

During the code review, It has been noticed that a potential issue with the implementation of the receive function in the smart contract. The receive function is not restricted to WETH (Wrapped Ether) token transfers, which
could lead to unexpected token transfers to your smart contract.
The unrestricted receive function may allow users to send native tokens to the contract. This could result in the undesired accumulation of tokens within the contract, making them permanently locked and inaccessible to
the intended recipients.

Listing 1

```
1     receive() external payable {}
```

Proof of Concept:

1. Send native token to ConnectorZEVM contract
2. Native token sent to ConnectorZEVM contract will be locked

Recommendation:

To address this issue, we recommend implementing a check within the receive function to ensure that only WETH token transfers are allowed. This can be achieved by comparing the address of the sender or the
transferred token with the known WETH contract address. This modification will prevent accidental or malicious transfers of unsupported tokens to your smart contract, ensuring that only the intended token is accepted.

Category of Vulnerability - **Halborn March 2023** - **Deficient Smart Contract**

Fee-On-Transfer Deflationary Tokens are not Supported

Relevant Contracts: `ERC20Custody.sol#LL126-L152`

During the audit, it was discovered that the smart contract does not support Fee-On-Transfer or deflationary tokens. These token types implement a fee mechanism, where a percentage of each token transfer is either burned, redistributed to token holders, or allocated to a specific address (e.g., a treasury or liquidity pool). The absence of support for these tokens may limit the smart contract's compatibility with various token projects and ecosystems.

Listing 2

```
    function deposit(bytes calldata recipient, IERC20 asset,
↳ uint256 amount, bytes calldata message) external {
        if (paused) {
            revert IsPaused();
        }
        if (!whitelisted[asset]) {
            revert NotWhitelisted();
        }
        if (address(zeta) != address(0)) {
            zeta.transferFrom(msg.sender, TSSAddress, zetaFee);
        }
        asset.transferFrom(msg.sender, address(this), amount);
        emit Deposited(recipient, asset, amount, message);
    }

    function withdraw(address recipient, IERC20 asset, uint256
↳ amount) external {
        if (paused) {
            revert IsPaused();
        }
        if (msg.sender != TSSAddress) {
            revert InvalidSender();
        }
        if (!whitelisted[asset]) {
            revert NotWhitelisted();
        }
        IERC20(asset).transfer(recipient, amount);
        emit Withdrawn(recipient, asset, amount);
    }
```

Proof of Concept:

1. Assume transfer fee to be 5% and ERC20Custody.sol has 200 token.
2. TSS Address deposit 100 tokens. Now, ERC20Custody.sol has 295 tokens.
3. TSS Address calls the method to withdraw 100 tokens.

4. ERC20Custody.sol ends up having 195 tokens.

Recommendation:

Consider checking the before and after balance of token transfer on the deposit and withdraw functions.

Category of Vulnerability - **Halborn March 2023** - **Deficient Smart Contract**

Absence of SafeTransfer/SafeTransferFrom in Token Transfers

Relevant Contracts: `ERC20Custody.sol#LL126-L152`

The given smart contract does not utilize the SafeTransfer or SafeTransferFrom functions for handling token transfers. These functions are part of the OpenZeppelin SafeERC20 library and are designed to provide additional safety checks and error handling when working with ERC20 tokens. Without these safety measures in place, the smart contract
may encounter unexpected issues or revert without providing clear error messages during token transfers, which can lead to confusion and difficulty in diagnosing problems.

Listing 3

```
    function deposit(bytes calldata recipient, IERC20 asset,
↳ uint256 amount, bytes calldata message) external {
        if (paused) {
            revert IsPaused();
        }
        if (!whitelisted[asset]) {
            revert NotWhitelisted();
        }
        if (address(zeta) != address(0)) {
            zeta.transferFrom(msg.sender, TSSAddress, zetaFee);
        }
        asset.transferFrom(msg.sender, address(this), amount);
        emit Deposited(recipient, asset, amount, message);
    }

    function withdraw(address recipient, IERC20 asset, uint256
↳ amount) external {
```

```
        if (paused) {
            revert IsPaused();
        }
        if (msg.sender != TSSAddress) {
            revert InvalidSender();
        }
        if (!whitelisted[asset]) {
            revert NotWhitelisted();
        }
        IERC20(asset).transfer(recipient, amount);
        emit Withdrawn(recipient, asset, amount);
    }
```

Proof of Concept:

Listing 4

```
contract NoRevertToken {


    // --- Token ---
    function transfer(address dst, uint wad) external returns (bool) {
        return transferFrom(msg.sender, dst, wad);
    }
    function transferFrom(address src, address dst, uint wad) virtual public returns (bool) {
        if (balanceOf[src] < wad) return false;
                        // insufficient src bal
        if (balanceOf[dst] >= (type(uint256).max - wad)) return false; // dst bal too high

        if (src != msg.sender && allowance[src][msg.sender] != type(uint).max) {
            if (allowance[src][msg.sender] < wad) return false;
        // insufficient allowance
            allowance[src][msg.sender] = allowance[src][msg.sender] - wad;
        }

        balanceOf[src] = balanceOf[src] - wad;
        balanceOf[dst] = balanceOf[dst] + wad;

        emit Transfer(src, dst, wad);
        return true;
    }

}
```

Recommendation:

To mitigate potential issues and enhance the safety of the smart contract, we recommend incorporating the use of SafeTransfer and SafeTransferFrom functions from

the OpenZeppelin SafeERC20 library.

Category of Vulnerability - **Halborn March 2023** - **Deficient Smart Contract**

ZRC20 Lacks Resistance to ERC20 Race Condition Issue

Relevant Contracts: `ZRC20.sol`

During the code review, It has been noticed that the ZRC20 implementation is not resistant to the well-known ERC20 race condition issue, also known as the allowance front-running problem. This issue occurs when a user
attempts to update their token allowance for a spender while the spender is simultaneously trying to use the existing allowance. If the spender's transaction is confirmed before the user's allowance update, the user might inadvertently grant the spender a higher allowance than intended.

Listing 5

```
    function _approve(address owner, address spender, uint256 amount) internal virtual {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }
```

Proof of Concept:

1. Alice initiates the approve(Bob, 500) function, granting Bob the permission to utilize 500 tokens.
2. Subsequently, Alice reconsiders and calls approve(Bob, 1000), which amends Bob's spending allowance to 1000 tokens.
3. Bob observes the transaction and swiftly invokes transferFrom(Alice, X, 500) before its mining completion.

4. If Bob’s transaction is mined before Alice’s, Bob will successfully transfer 500 tokens. Once Alice’s transaction has been mined, Bob can proceed to call transferFrom(Alice, X, 1000).

Recommendation:

To mitigate the race condition issue and enhance the security and reliability of the ZRC20 token implementation, we recommend implementing the increaseAllowance and decreaseAllowance functions, instead of only using
the approve function for modifying allowances. These functions allow users to update allowance values without first having to reduce them to zero, thus reducing the likelihood of encountering race condition issues.

Category of Vulnerability - Halborn March 2023 - Deficient Smart Contract

Insecure use of tx.origin in the Send Function

Relevant Contracts: `ConnectorZEVM.sol#83`

Listing 6

```
    function send(ZetaInterfaces.SendInput calldata input)
↳ external {
        // transfer wzeta to "fungible" module, which will be
↳ burnt by the protocol post processing via hooks.
        require(WZETA(wzeta).transferFrom(msg.sender, address(this
↳ ), input.zetaValueAndGas) == true, "wzeta.transferFrom fail");
        WZETA(wzeta).withdraw(input.zetaValueAndGas);
        (bool sent,) = FUNGIBLE_MODULE_ADDRESS.call{value: input.
↳ zetaValueAndGas}("");
```

```
        require(sent, "Failed to send Ether");
        emit ZetaSent(
            tx.origin,
            msg.sender,
            input.destinationChainId,
            input.destinationAddress,
            input.zetaValueAndGas,
            input.destinationGasLimit,
            input.message,
            input.zetaParams
        );
    }
```

Proof of Concept:

Listing 7

```
pragma solidity >=0.7.0 <0.9.0;
interface ConnectorZEVM {
    function send(ZetaInterfaces.SendInput calldata input)
↳ external
}

contract AttackerWallet {
    address payable owner;

    constructor() {
        owner = payable(msg.sender);
    }

    receive() external payable {
        ConnectorZEVM(msg.sender).send(input);
    }
}
```

Recommendation:

To mitigate the risks associated with the use of tx.origin, it is advised to replace tx.origin with msg.sender in the ZetaSent event emission. msg.sender provides a more secure way of identifying the direct caller of
the function, reducing the possibility of attacks.

Category of Vulnerability - Halborn March 2023 - Deficient Smart Contract

Fee Definition does not have an Upper Bound

Relevant Contracts: `ERC20Custody.sol#L70`

The current implementation of the smart contract does not define an upper bound for the fee structure. Without an upper limit in place, there is a possibility that the fees could be set to disproportionately high
values, either due to manipulation, incorrect configuration, or a bug in the smart contract. This can lead to a negative user experience, as users may be charged excessive fees for interacting with the contract.

Listing 8

```
1     function updateZetaFee(uint256 _zetaFee) external {
2         if (msg.sender != TSSAddress) {
3             revert InvalidSender();
4         }
5         if (_zetaFee == 0) {
6             revert ZeroFee();
7         }
8         zetaFee = _zetaFee;
9     }
```

Recommendation:

To address the issue and enhance the fee structure's security and predictability, we recommend defining an upper bound for the fee rates. This can be done by implementing a maximum fee rate value and a corresponding
validation check when setting the fee rate.

Category of Vulnerability - Halborn March 2023 - Deficient Smart Contract

Zetasent Event Parameters are not Validated on the Contract

Relevant Contracts: `ConnectorZEVM.sol#L85-L90`

Listing 9

```
contract ZetaConnectorZEVM is ZetaInterfaces{
    address public wzeta;
    address public constant FUNGIBLE_MODULE_ADDRESS = payable(0
↳ x735b14BB79463307AAcBED86DAf3322B1e6226aB);

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

    constructor(address _wzeta) {
        wzeta = _wzeta;
    }

    // the contract will receive ZETA from WETH9.withdraw()
    receive() external payable {}
```

```
22
23      function send(ZetaInterfaces.SendInput calldata input)
↳ external {
24          // transfer wzeta to "fungible" module, which will be
↳ burnt by the protocol post processing via hooks.
25          require(WZETA(wzeta).transferFrom(msg.sender, address(this
↳ ), input.zetaValueAndGas) == true, "wzeta.transferFrom fail");
26          WZETA(wzeta).withdraw(input.zetaValueAndGas);
27          (bool sent,) = FUNGIBLE_MODULE_ADDRESS.call{value: input.
↳ zetaValueAndGas}("");
28          require(sent, "Failed to send Ether");
29          emit ZetaSent(
30              tx.origin,
31              msg.sender,
32              input.destinationChainId,
33              input.destinationAddress,
34              input.zetaValueAndGas,
35              input.destinationGasLimit,
36              input.message,
37              input.zetaParams
38          );
39      }
```

Recommendation:

Consider validating all related parameters on the function.

Category of Vulnerability - **Halborn March 2023** - **Deficient Smart Contract**

Potential Blocking of Withdraw Operations due to High Gas Fees

Relevant Contracts: `ZRC20.sol#L152`

In the provided code snippet, the withdraw function requires the user to transfer a gas fee (gasFee) and a protocol flat fee (PROTOCOL_FLAT_FEE) to the FUNGIBLE_MODULE_ADDRESS before proceeding with the withdrawal. During times of network congestion or high gas prices, the total fees (gasFee + PROTOCOL_FLAT_FEE) might become prohibitively expensive for some users. As a result, these users might be unable to withdraw their tokens, effectively blocking them

from accessing their funds. The potential for blocked withdraw operations due to high gas fees can result in a negative user experience and may erode trust in the smart contract. Users who are unable to withdraw their tokens might be forced to wait for lower gas prices, which could take an indefinite amount of time, or they might have to pay exorbitant fees to access their funds.

Listing 10

```
    function withdraw(bytes memory to, uint256 amount) external
↳ override returns (bool) {
        (address gasZRC20, uint256 gasFee)= withdrawGasFee();
        require(IZRC20(gasZRC20).transferFrom(msg.sender,
↳ FUNGIBLE_MODULE_ADDRESS, gasFee+PROTOCOL_FLAT_FEE), "transfer gas
↳ fee failed");

        _burn(msg.sender, amount);
        emit Withdrawal(msg.sender, to, amount, gasFee,
↳ PROTOCOL_FLAT_FEE);
        return true;
    }
```

Recommendation:

Consider adjusting the required gas fees based on the current gas price, network conditions, or other factors, ensuring that users are not overcharged during high gas price periods.

Category of Vulnerability - **Halborn March 2023** - **Deficient Smart Contract**

Missing Slippage/Min-Return Check on CrossChainCall Function

Relevant Contracts: `SystemContract.sol#L48-L59`

The contracts are missing slippage checks, which can lead to being vulnerable to sandwich attacks. Sandwich attacks are prevalent in the decentralized finance (DeFi) sector. These attacks occur when an adversary observes a trade involving assets X and Y and preemptively purchases asset Y. Following the execution of the victim's trade, the attacker sells the acquired amount of asset Y, capitalizing on the inflated price. The

attacker profits by exploiting the knowledge of an impending trade that will increase the asset's price, ultimately causing a financial loss for the protocol.
Lacking appropriate slippage checks, trades may be executed at unfavorable prices, resulting in the acquisition of fewer tokens than the fair market value dictates.

Listing 11

```
    function depositAndCall(
        address zrc20,
        uint256 amount,
        address target,
        bytes calldata message
    ) external {
        if (msg.sender != FUNGIBLE_MODULE_ADDRESS) revert
↳ CallerIsNotFungibleModule();
        if (target == FUNGIBLE_MODULE_ADDRESS || target == address
↳ (this)) revert InvalidTarget();
```

```

        IZRC20(zrc20).deposit(target, amount);
        zContract(target).onCrossChainCall(zrc20, amount, message)
↳ ;
    }
```

Recommendation:

Without appropriate slippage checks, trades may be executed at unfavorable prices, resulting in the acquisition of fewer tokens than the fair market value dictates. To mitigate this risk, we recommend implementing minimum return amount checks in the following manner:

- Introduce a function parameter to be determined by the transaction initiator, ensuring that the received amount exceeds the specified parameter.