Veridise. **Auditing Report**

Hardening Blockchain Security with Formal Methods

FOR

ZetaChain: Smart Contracts

Veridise Inc.
July 15, 2022

► **Prepared For:**

ZetaChain
https://www.zetachain.com/

► **Prepared By:**

Ben Mariano
Bryan Tan
Hongbo

► **Contact Us:** contact@veridise.com

► **Version History:**

July 15, 2022	V1

Contents

1 Executive Summary

From July 5 to July 13, ZetaChain engaged Veridise to review the security of their blockchain implementation which supports generic, decentralized, omnichain smart contracts. The audit covered only the smart contract portion of the protocol, which included implementations of the Zeta token (used by the protocol), "connector" contracts which enable communication across blockchains, and a few useful tools for DeFi app developers, such as the ZetaInteractor which implements some useful safety checks and two ZetaTokenConsumers which use UniSwap to facilitate exchange of Zeta for other tokens. Veridise conducted this assessment over 2 person-weeks, with 2 engineers working on code from commit `10a8344` to `30cd3d2` of the `zeta-chain/zetachain` repository. The auditing strategy involved tool-assisted analysis of the source code performed by Veridise engineers. The tools used in the audit included a combination of static analysis and formal verification.

Summary of issues detected. The audit uncovered 9 issues. The most severe issues included two reentrancy attacks which enable a malicious user to reorder events that are potentially used by external applications. In addition to these issues, auditors also identified other potential issues regarding the interaction of the contracts with the protocol – in these cases, improper handling of this interaction could lead to loss of funds and other potential protocol disruptions. Additionally, our auditors also discovered multiple gas optimizations and code suggestions which can help improve the efficiency, usability, and maintainability of the code.

Code assessment. The smart contracts being assessed for this audit are relatively small, but are essential to the proper functioning of the blockchain protocol. Furthermore, these contracts are not a fork of an existing protocol but were created by the developers from scratch. The contracts make good use of the well-known and audited contracts from OpenZeppelin. The code also includes testing of the contracts using the Hardhat framework which provide good coverage of the contract behaviors. However, as far as our auditors can tell, no tests included in the repo test the interaction of the contracts with the rest of the protocol.

Recommendations In accordance with the contract between ZetaChain and Veridise, our audit focused only on the smart contracts described and not the rest of the blockchain protocol. However, the correctness of the contracts considered (as well as the protocol overall) requires that (1) the rest of the protocol is implemented correctly and (2) information is appropriately communicated from the contracts to the protocol and vice versa. To ensure the safety of the full protocol, we suggest ZetaChain seek an audit for the full protocol including both the smart contracts and core protocol implementation.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Project Dashboard 2

Table 2.1: Application Summary.

Name	Version	Type	Platform
Smart Contracts	`10a8344` - `30cd3d2`	Solidity	Ethereum

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
July 5 - July 13, 2022	Manual & Tools	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Resolved
Critical-Severity Issues	0	0
High-Severity Issues	0	0
Medium-Severity Issues	2	2
Low-Severity Issues	0	0
Warning-Severity Issues	3	2
Informational-Severity Issues	4	1
TOTAL	9	5

Table 2.4: Category Breakdown.

Name	Number
Reentrancy	2
Maintainability	2
Gas Optimizations	2
Protocol	3

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of the ZetaChain blockchain implementation, focusing specifically on the smart contracts used to implement the Zeta token and "Connector" contracts used for communicating information across chains. In our audit, we sought to answer the following questions regarding these contracts:

- If a user sends ZETA from blockchain A to blockchain B and the send fails on blockchain B, will the user eventually be refunded their ZETA?
- If a user successfully sends ZETA from blockchain A to blockchain B, is their total balance (across both chains) appropriately adjusted?
- Can a malicious user prevent a user from sending/receiving ZETA cross-chain?
- Do the contracts have appropriate access control? In particular, is all functionality that should be exclusive to the validators (collectively) appropriately limited to that group?
- Are inputs cross-chain properly verified/sanitized? That is, can a malicious user cause unwanted behavior by sending carefully crafted requests?
- Are events appropriately emitted to ensure that the protocol can track relevant actions?
- Can a malicious user alter the order/timing of events to inappropriately manipulate the behavior of the protocol?
- Do the Token Consumer contracts appropriately leverage Uniswap to facilitate trading between Zeta and other tokens?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis and testing tools. In particular, we conducted our audit with the aid of the following techniques:

- Static Analysis. We leveraged the open-source tool Slither to find common vulnerabilities in smart contracts, such as reentrancy and uninitialized variables. In addition to the default behavior of Slither, we also extended Slither with custom detectors for more precise and expansive bug detection.
- Formal Verification. We used Veridise's tool Eurus which enables symbolic reasoning about smart contracts. Our engineers wrote a set of pre/post conditions (i.e. specifications) for each function of interest in the smart contracts and used Eurus to formally verify that the functions satisfy the specifications.

Scope. To understand the scope of the audit, we first reviewed the documentation shared by the ZetaChain developers, including the online documentation, whitepaper, and video descriptions of the code. We found the example DeFi apps provided in the documentation particularly useful in understanding the desired behavior of the code. After assessing documentation,we ran the provided Hardhat test suite, ran Slither, and performed a manual inspection/audit of the code. Finally, using the understanding gained from manual audit, we wrote preconditions and postconditions for each function of interest in the audit and used our verifier Eurus to formally verify each property.

Concretely, the contracts considered in this audit were the following:

- Zeta.eth.sol
- Zeta.non-eth.sol
- ZetaConnector.base.sol
- ZetaConnector.eth.sol
- ZetaConnector.non-eth.sol
- ZetaInteractor.sol
- ZetaTokenConsumerUniV2.strategy.sol
- ZetaTokenConsumerUniV3.strategy.sol

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows:

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows:

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowleged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-ZET-VUL-001	Event Reordering via Reentrancy	Medium	Fixed
V-ZET-VUL-002	Event Reordering via Reentrancy	Medium	Fixed
V-ZET-VUL-003	Immutable Deadline for Token Exchange	Warning	Addressed
V-ZET-VUL-004	Use Separate ZetaToken Interface	Warning	Fixed
V-ZET-VUL-005	Declare Constant Value as Constant	Warning	Fixed
V-ZET-VUL-006	Sanitize Input to onReceive and onRevert	Info	Acknowledged
V-ZET-VUL-007	Malicious User Can Prevent ZetaRevert Event	Info	Expected Behavior
V-ZET-VUL-008	Protocol Concerns When onRevert Reverts	Info	Acknowledged
V-ZET-VUL-009	Declare Immutable Addresses as Immutable	Info	Fixed

Note, for the statuses listed above, we define them as follows:

Fixed	Developers acknowledged the issue and added the fix suggested by Veridise auditors.
Addressed	Developers acknowledged the issue and added a different fix than the fix suggested by Veridise auditors. Veridise auditors verified that the fix performed as the customer expected.
Acknowledged	Developers acknowledged the issue but indicated that it should be addressed (or already is) by code outside of the scope of the current audit. If necessary, developers have forwarded the issue to the team/s in charge of the relevant code.
Open	Developers acknowledged the issue but have not yet addressed it.
Expected Behavior	Developers say the issue identified by Veridise auditors is actually the intended behavior of the code.

4.1 Detailed Description of Bugs

In this section, we describe each uncovered vulnerability in more detail.

4.1.1 V-ZET-VUL-001: Possible Event Reordering via Reentrancy in getEthFromZeta

Severity	Medium	**Commit**	10a8344
Type	Reentrancy	**Status**	Fixed
Files	ZetaTokenConsumerUniV3.strategy.sol		
Functions	gerEthFromZeta		

Description 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;
}

```

Snippet 4.1: Function getEthFromZeta

Attack Scenerio 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();
  }
}

```

Snippet 4.2: Contract Reentrant

```
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);
    }
  }
}
```

Snippet 4.3: Contract AttackHelper

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

Developer Response The developers acknowledged the issue and took our recommendation in commit `30cd3d2`.

4.1.2 V-ZET-VUL-002: Possible Event Reordering via Reentrancy in getZetaFromToken

Severity	Medium	**Commit**	10a8344
Type	Reentrancy	**Status**	Fixed
Files	ZetaTokenConsumerUniV2/3.strategy.sol		
Functions	getZetaFromToken		

Description 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, allowing 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);
}
```

Snippet 4.4: Function getZetaFromToken

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

Developer Response The developers acknowledged the issue and added a reentrancy guard in commit `30cd3d2`.

4.1.3 V-ZET-VUL-003 Allow Updating of MAX_DEADLINE for TokenConsumers

Severity	Warning	**Commit**	10a8344
Type	Maintainability	**Status**	Addressed
Files	ZetaTokenConsumerUniVx.strategy.sol (versions 2 and 3)		
Functions	-		

Description 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.

Developer Response The developers acknowledged this could be an issue, but given that these are helper functions for users only, they opted instead to increase the deadline to the constant of 200 and presume this will be sufficient for the forseeable future.

4.1.4 V-ZET-VUL-004: Use Separate ZetaToken Interface

Severity	Warning	**Commit**	10a8344
Type	Maintainability	**Status**	Fixed
Files	ZetaConnector.non-eth.sol		
Functions	burnFrom, mint		

Description 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;
}
```

Snippet 4.5: Interface ZetaToken

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.

Developer Response The developers acknowledged the issue and implemented our suggested fix in commit `30cd3d2`.

4.1.5 V-ZET-VUL-005 Save Gas by Replacing keccak(constant) with a Constant

Severity	Warning	**Commit**	10a8344
Type	Gas Optimization	**Status**	Fixed
Files	ZetaInteractor.sol		
Functions	_isValidChainId		

Description 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)));
}
```

Snippet 4.6: Function _isValidChainId

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

Developer Response The developers acknowledged the issue and adopted our suggested fix in commit `30cd3d2`.

4.1.6 V-ZET-VUL-006: Protocol Must Sanitize Inputs to onReceive and onRevert

Severity	Informational	**Commit**	10a8344
Type	Protocol	**Status**	Acknowledged
Files	ZetaTokenConnector.eth.sol, ZetaTokenConnector.non-eth.sol		
Functions	onReceive, onRevert		

Description 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 {}
```

Snippet 4.7: Functions onReceive and onRevert

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.

Developer Response The developers acknowledged the potential concern here and agreed to forward these recommendations to the team in charge of protocol development.

4.1.7 V-ZET-VUL-007: A malicious User Can Prevent ZetaRevert Event

Severity	Informational	**Commit**	10a8344
Type	Protocol	**Status**	Expected Behavior
Files	ZetaTokenConsumerUniV3.strategy.sol		
Functions	onRevert		

Description 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(...);
}
```

Snippet 4.8: ZetaConnector.eth.sol

```
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(...);
}
```

Snippet 4.9: ZetaConnector.non-eth.sol

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.

Developer Response The developers described that this is desired behavior and that they have discussed related issues already with the protocol developers.

4.1.8 V-ZET-VUL-008: Potential Protocol Concerns When onRevert Reverts

Severity	Informational	**Commit**	10a8344
Type	Protocol	**Status**	Acknowledged
Files	ZetaConnector.eth.sol, ZetaConnector.non-eth.sol		
Functions	onRevert		

Description 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.

Developer Response Developers describe that they have opted for option (3) and will only execute the `onReceive` and `onRevert` calls up to a fixed amount of gas provided by the user.

4.1.9 V-ZET-VUL-009: Save Gas by Declaring Addresses as Immutable and External Contracts as Local Variables

Severity	Informational	**Commit**	10a8344
Type	Gas Optimization	**Status**	Fixed
Files	ZetaConnector.base.sol, ZetaInteractor.sol		
Functions	-		

Description 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;
  ...
}
```

Snippet 4.10: Contract ZetaConnector.base.sol

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;
  ...
}
```

Snippet 4.11: Contract ZetaInteractor.sol

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(...);
    ...
  }
}
```

Snippet 4.12: Contract ZetaTokenConsumerUniV2.strategy.sol

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)`.

Author Response The developers acknowledged and implemented all of the suggested optimizations.

5 Formal Verification Results

5.1 Description

In this section, we detail all of the properties we formally checked. For each property, we give its formal representation as well as an English description describing what it is testing. A $\surd$ indicates the property was verified, while a $\times$ indicates the property was falsified. For this project, we checked 17 properties designed by our engineers after they carefully read documentation and code to determine the desired behavior of each function. All 17 properties were verified by Eurus.

5.2 Specifications

Specifications are given as preconditions and postconditions for each function. We express a specification using the following syntax

$$\{P\}\ f(\bar{A})\ \{Q\}$$

where P is a logical formula representing the precondition, $f(\bar{A})$ is the function being verified (where $\bar{A}$ are the names of the arguments of f), and Q is the postcondition. Informally, one can think of this spec as saying "if P holds before f executes, then after f executes, Q must hold".

In our context, $f(\bar{A})$ is a function in a smart contract (let's call the contract C) and P and Q are first-order formulas over the state variables of C, arguments $\bar{A}$ of f, and pure function calls from C. As an example, consider the following specification for a mint function in a simple token:

$$\{msg.sender = owner\}$$

$$\text{mint(address } a, \text{uint256 } amt)$$

$$\{\text{balanceOf}(a) = \text{old}(\text{balanceOf}(a)) + amt\}$$

This specification states that a call to mint with target address a and amount amt will add the desired amount amt to the balance of address a, presuming the function is called by the owner of the token, denoted $owner$. Note that the expression old(balanceOf(a)) uses the $old(e)$ expression, which indicates the value of e before the execution of the transaction.

5.3 Results

In what follows, we give the formal properties checked by Eurus for each file/function considered, as well as an English description of the property and the result of that check.

5.3.1 Zeta.non-eth.sol

√ Property 0

`renounceTssAddressUpdater()` appropriately sets the $tssAddressUpdater$ to be the $tssAddress$.

$$\{msg.sender = tssAddressUpdater \land tssAddress \neq 0\}$$

`renounceTssAdressUpdater()`

$$\{tssAddressUpdater = tssAddress\}$$

√ Property 1

A valid call to mint will increase the total supply of tokens by the minted amount.

$$\{msg.sender = connectorAddress \land a \neq 0\}$$

$$\text{mint(address } a, \text{ uint } amt)$$

$$\{\text{totalSupply()} = \text{old(totalSupply())} + amt\}$$

√ Property 2

A valid call to mint will increase the balance of the mintee by the minted amount.

$$\{msg.sender = connectorAddress \land a \neq 0\}$$

$$\text{mint(address } a, \text{ uint } amt)$$

$$\{\text{balanceOf}(a) = \text{old(balanceOf}(a)) + amt\}$$

√ Property 3

`updateTssAndConnectorAddresses` will update the desired addresses if called by a permitted user with non-zero addresses.

$$\{(msg.sender = tssAddressUpdater \lor msg.sender = tssAddress) \land t \neq 0 \land c \neq 0\}$$

$$\text{updateTssAndConnectorAddresses(address } t, \text{ address } c)$$

$$\{tssAddress = t \land connectorAddress = c\}$$

√ Property 4

A valid call to `burn` will decrease the total supply of tokens by the burned amount.

$$\{msg.sender = connectorAddress \wedge a \neq 0 \wedge allowance[a][msg.sender] \geq amt\}$$

$$\text{burnFrom(address } a, \text{ uint } amt)$$

$$\{\text{totalSupply() = old(totalSupply()) - } amt\}$$

√ Property 5

A valid call to `burn` will decrease the balance of the burnee by the burned amount.

$$\{msg.sender = connectorAddress \wedge a \neq 0 \wedge allowance[a][msg.sender] \geq amt\}$$

$$\text{burnFrom(adress } a, \text{ uint } amt)$$

$$\{\text{balanceOf(}a\text{) = old(balanceOf(}a\text{)) - } amt\}$$

5.3.2 ZetaConnector.base.sol

√ Property 6

`updatePauserAddress` updates the pauser address to the desired address presuming it is called by the pauser address and the updated pauser address is not 0.

$$\{msg.sender = pauserAddress \wedge p \neq 0\}$$

$$\text{updatePauserAddress(address } p)$$

$$\{pauserAddress = p\}$$

√ Property 7

`updateTssAddress` updates the $tssAddress$ to the desired address presuming it is called by the current $tssAddress$ or the $tssAddressUpdater$ and the new $tssAddress$ is not 0.

$$\{(msg.sender = tssAddress \vee msg.sender = tssAddressUpdater) \wedge t \neq 0\}$$

$$\text{updateTssAddress(address } t)$$

$$\{tssAddress = t\}$$

√ Property 8

`renounceTssAddressUpdater` updates the $tssAddressUpdater$ to $tssAddress$ presuming it is called by the current $tssAddressUpdater$ and $tssAddress$ is not 0.

$$\{msg.sender = tssAddressUpdater \land tssAddress \neq 0\}$$

$$\text{renounceTssAddressUpdater()}$$

$$\{tssAddressUpdater = tssAddress\}$$

√ Property 9

When `pause` is called by the pauser, the $\_pause$ flag is set to `True`.

$$\{msg.sender = pauserAddress\}$$

$$\text{pause()}$$

$$\{\_pause\}$$

√ Property 10

When `unpause` is called by the pauser the $\_pause$ flag is set to `False`.

$$\{msg.sender = pauserAddress\}$$

$$\text{unpause()}$$

$$\{\neg\_pause\}$$

5.3.3 ZetaConnector.non-eth.sol

√ Property 11

A `send` decreases the total supply by the value indicated by the sender, presuming they have enough funds to start with.

$$\{\neg\_pause \land \text{ ZetaToken}(zetaToken)\text{.balanceOf}(msg.sender) \geq input.zetaValueAndGas\}$$

$$\text{send(SendInput } input)$$

$$\{\text{totalSupply() = old(totalSupply()) - } input.zetaValueAndGas\}$$

√ Property 12

A `send` decreases the sender's balance by the value indicated, presuming they have enough funds to start with.

$$\{\neg\_pause \wedge \text{ZetaToken}(zetaToken).\text{balanceOf}(msg.sender) \geq input.zetaValueAndGas\}$$

$$\text{send(SendInput } input)$$

$$\{\text{balanceOf}(msg.sender) = \text{old}(\text{balanceOf}(msg.sender)) \text{ - } input.zetaValueAndGas\}$$

√ Property 13

`onReceive` increases the total supply by the value indicated by the sender, presuming the amount does not exceed the current limit.

$$\{\neg\_paused \wedge msg.sender = tssAddress \wedge zetaValueAndGas + \text{ZetaToken}(zetaToken).\text{totalSupply}() \leq maxSupply \wedge destinationAddress \neq 0\}$$

$$\text{onReceive(address } zetaTxSenderAddress, \text{ uint256 } sourceChainId, \text{ address } destinationAddress, \text{ uint256 } zetaValueAndGas, \text{ bytes } message, \text{ bytes32 } internalSendHash)$$

$$\{\text{totalSupply}() = \text{old}(\text{totalSupply}()) + zetaValueAndGas\}$$

√ Property 14

`onReceive` increases the balance of the *destinationAddress* by the value indicated by the sender, presuming the amount does not exceed the current limit.

$$\{\neg\_paused \wedge msg.sender = tssAddress \wedge zetaValueAndGas + \text{ZetaToken}(zetaToken).\text{totalSupply}() \leq maxSupply \wedge destinationAddress \neq 0\}$$

$$\text{onReceive(address } zetaTxSenderAddress, \text{ uint256 } sourceChainId, \text{ address } destinationAddress, \text{ uint256 } zetaValueAndGas, \text{ bytes } message, \text{ bytes32 } internalSendHash)$$

$$\{\text{balanceOf}(destinationAddress) = \text{old}(\text{balanceOf}(destinationAddress)) + zetaValueAndGas\}$$

√ Property 15

`onRevert` increases the total supply by the value indicated by the sender, presuming the amount does not exceed the current limit.

$$\{\neg\_paused \wedge msg.sender = tssAddress \wedge zetaValueAndGas + \text{ZetaToken}(zetaToken).\text{totalSupply}() \leq maxSupply \wedge zetaTxSenderAddress \neq 0\}$$

$$\text{onRevert(address } zetaTxSenderAddress, \text{ uint256 } sourceChainId, \text{ address } destinationAddress, \text{ uint256 } zetaValueAndGas, \text{ bytes } message, \text{ bytes32 } internalSendHash)$$

$$\{\text{totalSupply}() = \text{old}(\text{totalSupply}()) + zetaValueAndGas\}$$

√ Property 16

`onReceive` increases the balance of the $zetaTxSenderAddress$ by the value indicated by the sender, presuming the amount does not exceed the current limit.

$$\{\neg\_paused \wedge msg.sender = tssAddress \wedge zetaValueAndGas + \text{ZetaToken}(zetaToken).\text{totalSupply}() \leq maxSupply \wedge zetaTxSenderAddress \neq 0\}$$

$$\text{onRevert}(\text{address } zetaTxSenderAddress, \text{uint256 } sourceChainId, \text{address } destinationAddress, \text{uint256 } zetaValueAndGas, \text{bytes } message, \text{bytes32 } internalSendHash)$$

$$\{\text{balanceOf}(zetaTxSenderAddress) = \text{old}(\text{balanceOf}(zetaTxSenderAddress)) + zetaValueAndGas\}$$