Security Audit Report

ZIGChain Update

v1.1

December 12, 2025

Table of Contents

License

Disclaimer

THE CONTENT OF THIS AUDIT REPORT IS PROVIDED “AS IS”, WITHOUT REPRESENTATIONS AND WARRANTIES OF ANY KIND.

THE AUTHOR AND HIS EMPLOYER DISCLAIM ANY LIABILITY FOR DAMAGE ARISING OUT OF, OR IN CONNECTION WITH, THIS AUDIT REPORT.

THIS AUDIT REPORT WAS PREPARED EXCLUSIVELY FOR AND IN THE INTEREST OF THE CLIENT AND SHALL NOT CONSTRUE ANY LEGAL RELATIONSHIP TOWARDS THIRD PARTIES. IN PARTICULAR, THE AUTHOR AND HIS EMPLOYER UNDERTAKE NO LIABILITY OR RESPONSIBILITY TOWARDS THIRD PARTIES AND PROVIDE NO WARRANTIES REGARDING THE FACTUAL ACCURACY OR COMPLETENESS OF THE AUDIT REPORT.

FOR THE AVOIDANCE OF DOUBT, NOTHING CONTAINED IN THIS AUDIT REPORT SHALL BE CONSTRUED TO IMPOSE ADDITIONAL OBLIGATIONS ON COMPANY, INCLUDING WITHOUT LIMITATION WARRANTIES OR LIABILITIES.

This audit has been performed by

Oak Security GmbH

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security GmbH has been engaged by Highend Technologies LLC to perform a security audit of the ZIGChain chain implementation updates.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/ZIGChain/zigchain-private
Commit	`ee96e9b83d488b6540a9c80cd3614b83a1876666`
Scope	Updates since the last audit performed by Oak Security, which was conducted at commit `a0584a1329b55109ff1d612636534f7bada3d01f`.
Fixes verified at commit	`fe817f66275e6f13a0609755fcb2df6b41598ada` Note that only fixes to the issues described in this report have been reviewed at this commit. Any further changes such as additional features have not been reviewed.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The scope of the audit for ZIGChain features updates since the previous audit to the following custom modules:

- The `x/dex` module enables users to create pools, provide or withdraw liquidity, and swap tokens.
- The `x/factory` module allows users to create custom native tokens with granular access controls via the bank and metadata admin, while enforcing the max supply.
- The `x/tokenwrapper` module facilitates the bridging between ZIG tokens on Axelar (originating from Ethereum) and native ZIG tokens in ZIGChain. Tokens are wrapped or unwrapped accordingly when transferring or receiving from ZIGChain.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending**, **Acknowledged**, **Partially Resolved**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	**Medium-Low**	-
Code readability and clarity	**Medium-High**	-
Level of documentation	**High**	Detailed documentation is available at https://docs.zigchain.com and in the respective `README` files.
Test coverage	**Medium-High**	`go cover` reports a coverage of `70.2%`, which excludes proto-generated files. For more information on how we compute test coverage, please see this section in the appendix.

Summary of Findings

No	Description	Severity	Status
1	Sent tokens are not refunded when the IBC counterpart fails	Critical	Resolved
2	Token precision loss during conversion locks more tokens than intended	Minor	Acknowledged
3	Operator address change during recovery inflates `TotalTransferredIn`	Informational	Acknowledged
4	Unresolved TODO and FIXME comments	Informational	Resolved
5	Running the test suite does not succeed	Informational	Resolved
6	Redundant code	Informational	Resolved
7	Simulation of MsgRecoverZig does cover successful scenarios	Informational	Resolved
8	Redundant config creation in a simulation	Informational	Resolved
9	MsgStoreCode filter does not handle wrapped authz messages	Informational	Resolved

Detailed Findings

1. Sent tokens are not refunded when the IBC counterpart fails

Severity: Critical

In `x/tokenwrapper/module/on_acknowledgment_packet.go`, the function `OnAcknowledgementPacket` is defined in lines `14-137`. This function handles the acknowledgement of the packet reception by the IBC counterpart.

In line `19`, the function declares the parameter `acknowledgement` of type `[]byte`. On line `108`, the `acknowledgement` is converted into the `Acknowledgement` type using the function `NewResultAcknowledgement`, and the result is assigned to the variable `ack`. Further, handling of failed acknowledgements is implemented on lines `111-122`.

However, the function `NewResultAcknowledgement` is not the correct way to deserialize bytes into the `Acknowledgement` structure. It results in `ack.Response` always being of type `Acknowledgement_Result`, and never being of type `Acknowledgement_Error`.

The `if` condition on line `113` guards the failure handler using the `!ack.Success()` condition, where `Success` is defined as `ack.Response` having exactly the type `Acknowledgement_Result`. This means that `!ack.Success` is always `false`, because `NewResultAcknowledgement` never produces the type `Acknowledgement_Error`.

As a consequence:

- Refunding, implemented on the line `113` using the call to `im.handleRefund`, never occurs. The user effectively loses the sent tokens.
- The success path is executed in case of a failure, too. Specifically, lines `129-139`, which increment the total number of successfully sent tokens, using the call to `AddToTotalTransferredOut`.

Any failure on the other side of the IBC connection results in irrecoverably lost tokens and incorrect internal accounting.

Recommendation

We recommend deserializing the acknowledgement correctly, using the `json.Unmarshal` function.

Status: Resolved

2. Token precision loss during conversion causes tokens loss in dust amount

Severity: Minor

In `x/tokenwrapper/keeper/keeper.go:548`, the `RecoverZig` function converts IBC tokens to native tokens via the `ScaleDownTokenPrecision` function, which performs integer division using `amount.Quo(conversionFactor)` to convert from 18 decimals to 6 decimals. This operation rounds down and discards the remainder (modulus).

The `RecoverZig` function retrieves the entire IBC voucher balance from the user's address via `k.bankKeeper.GetBalance(ctx, address, recvDenom)`. This full balance is locked via `LockIBCTokens`, but only the scaled-down `convertedAmount` is returned to users as native tokens. This means that users permanently lose the fractional amount that gets discarded during conversion.

For example, if a user has 1,000,000,100,000,000,000 IBC vouchers, i.e. $10^{18} + 10^{11}$ tokens, the conversion yields 1,000,000 native tokens, or 10^6. The opposite conversion from 10^6 would only produce 10^{18}. But the entire balance of $10^{18} + 10^{11}$ is locked. The user permanently loses the fractional amount equal to 10^{11}, as it remains locked in the module account.

Since `RecoverZig` is permissionless, allowing anyone to process other users' entire balances, the impact is immediately visible to users who lose a portion of their funds with each recovery operation.

The same issue also occurs in the `OnRecvPacket` function at `x/tokenwrapper/module/on_recv_packet.go:152`, where IBC token amounts are locked, but only the scaled-down converted amount is unlocked as native tokens.

Recommendation

We recommend modifying the token locking logic only to lock the amount that will actually be converted and returned to the user. Calculate the exact lockable amount by multiplying the converted amount back by the conversion factor before locking.

For example, if the `convertedAmount` is 10^6, lock only 10^{18} instead of the original amount.

Status: Acknowledged

3. Operator address change during recovery inflates `TotalTransferredIn`

Severity: Informational

In `x/tokenwrapper/keeper/keeper.go:523`, the `RecoverZig` function implements a validation that prevents recovery on the operator address by comparing the `address.String() == k.GetOperatorAddress(ctx)` condition.

The issue is that the operator address can be changed via `MsgProposeOperatorAddress` and `MsgClaimOperatorAddress`. If the operator address is changed, the previous operator address would no longer be recognized as the current operator.

This could allow recovery operations to proceed at the previously operator address, which may contain IBC denoms, and potentially inflate the `TotalTransferredIn` through the `k.AddToTotalTransferredIn(ctx, convertedAmount)` call at the end of the function.

Recommendation

We recommend implementing a temporary pause mechanism for recovery operations during an operator address change.

Status: Acknowledged

4. Unresolved TODO and FIXME comments

Severity: Informational

The codebase contains multiple FIXME and TODO comments indicating incomplete work that has not been addressed:

- `x/tokenwrapper/module/validators.go:25`: FIXME comment above the `validateChannel` function for `ICS4Wrapper`.
- `x/tokenwrapper/module/validators.go:70`: FIXME comment above the `checkCounterypartyChannelMatchesIBCSettings` function for `ICS4Wrapper`.
- `x/tokenwrapper/module/validators.go:142`: FIXME comment above the `validateConnectionClientId` function for `ICS4Wrapper`.
- `x/tokenwrapper/simulation/recover_zig.go:26`: TODO comment indicating the missing simulation functionality.
- `x/tokenwrapper/module/simulation.go:28-39`: TODO comments indicating that simulation weights are not determined yet.

These functions are duplicated between `IBCModule` and `ICS4Wrapper` types, and the FIXME comments indicate the intent to refactor them into a shared location to reduce code duplication and improve maintainability.

Additionally, in `x/tokenwrapper/simulation/recover_zig.go:26`, the `SimulateMsgRecoverZig` function contains a TODO comment stating "*Handling the RecoverZig simulation*", but the simulation is not implemented.

While these issues do not affect functionality or security in production, unresolved FIXME and TODO comments could accumulate technical debt and make the codebase harder to maintain over time.

Recommendation

We recommend addressing the outstanding comments.

Status: Resolved

5. Running the test suite does not succeed

Severity: Informational

Currently, running the command `make test`, defined by the `Makefile`, results in multiple errors like `api/zigchain/factory/tx.pulsar.go:11446:2: unreachable code`.

On other hand, when using a simpler approach with `go test ./...`, the test cases pass until one of them crashes with message `panic: failed to create VM for 08 light client`.

Every time, the failing test case is different and running this test case individually with separate `WASMVM_CACHE_DIR` succeeds, so there is no undetected risk.

However, this indicates that the test suite is not being run systematically and the prepared command `make test` is not up-to-date.

Recommendation

We recommend ensuring that all test cases execute successfully when a developer runs `make test`, documenting the testing procedure in the `README` file and establishing Continuous Integration by running the test suite on every commit to the repository.

Status: Resolved

6. Redundant code

Severity: Informational

The are several unreachable parts of the codebase:

- Several files, that seem to be automatically generated in the past, cannot be re-generated anymore. There are no corresponding `.proto` files in the current state of the codebase. These files are stale artifacts which are not used currently:
  - `api/zigchain/factory/auth.pulsar.go`
  - `api/zigchain/tokenwrapper/packet.pulsar.go`

    - `x/tokenwrapper/types/packet.pb.go`
- Folder `app/upgrades/v1`, containing two files identical to files with same names in the folder `app/upgrades/v2`. The `v1` upgrade is not included into the Upgrades array, defined in `app/setup_handlers.go:14`, meaning that it is not used and can be removed:
    - `app/upgrades/v1/upgrades.go`
    - `app/upgrades/v1/constants.go`
- File `app/keepers/keepers.go` contains commented code on lines `228-242`.
- The function `MakeTestTxConfig` in `x/tokenwrapper/simulation/helpers.go:22-25` is not used in the codebase, and the same function is already defined in `app/sim_test.go:55-63`.
- The simulation function `SimulateMsgAddLiquidity` in `x/dex/simulation/add_liquidity.go:18-19` does not utilize its parameters, and the body of the function is `TODO`.
    - Similar issue is present in some simulation functions in other files:
        - `SimulateMsgRemoveLiquidity`
        - `SimulateMsgSwap`
        - `SimulateMsgWithdrawModuleFees`
        - `SimulateMsgCreatePool`

Redundancies in the codebase hinder reviews and maintainability. Commented-out code usually does not serve as documentation or a recipe for future improvements, instead it tends to become obsolete and reduces readability.

Recommendation

We recommend removing the redundant code.

Status: Resolved

7. Simulation of `MsgRecoverZig` does not cover successful scenarios

Severity: Informational

In `x/tokenwrapper/simulation/recover_zig.go:37`, the function `SimulateMsgRecoverZig` generates a random address `recoverAddress` which will be used to simulate tokens recovery.

However, since the address is random it does not own any IBC vouchers for successful recovery. Hence, the simulation exercises only failure paths. The successful scenarios are very unlikely.

Recommendation

We recommend simulating the main scenario by adding IBC vouchers to `recoverAddress`.

Status: Resolved

8. Redundant config creation in a simulation

Severity: Informational

In `x/tokenwrapper/simulation/recover_zig.go:49`, the function `SimulateMsgRecoverZig` creates new `simulation.OperationInput` with `moduletestutil.MakeTestEncodingConfig().TxConfig` as value for the field `TxGen`.

However, the function accepts parameter `txCfg` of type `client.TxConfig` on line `20`. This parameter represents the config passed to all other simulations and should be used instead of new configuration.

Recommendation

We recommend replacing `moduletestutil.MakeTestEncodingConfig().TxConfig` with the parameter `txCfg` on line `49`.

Status: Resolved

9. `MsgStoreCode` filter does not handle wrapped `authz` messages

Severity: Informational

The `wasmModuleSimulationWrapper` in `app/app.go:141-160` filters out `MsgStoreCode` operations during simulation to avoid address codec errors.

However, the current implementation only checks for direct `MsgStoreCode` messages by examining the message name and route. This check does not account for `MsgStoreCode` messages wrapped in a `MsgExec` when called through the `authz` module.

When a user grants another account authorization to execute `MsgStoreCode` on their behalf, the actual `MsgStoreCode` is encapsulated within a `cosmos.authz.v1beta1.MsgExec` message.

In this scenario, the outer message would have a name or route of `MsgExec`, allowing the wrapped `MsgStoreCode` to bypass the filter during simulation.

Recommendation

We recommend documenting this limitation or extending the filter logic to inspect the inner messages of `MsgExec` operations.

Status: Resolved

Appendix

1. Command to compute test coverage

To compute the test coverage accurately without including code generated by protoc-gen-gogo and protoc-gen-grpc-gateway, file extensions that end with `.pb.go` or `.pb.gw.go` need to be excluded.

The following command is executed to compute the test coverage:

```
go test -coverprofile=coverage.out ./... && grep -vE
"(simulation|mocks?|testutil|mock_|migrations|events|api|docs|upgrades|sample|nu
llify|network|debug|tests|constants|\.pb\.go|\.pb\.gw|\.pulsar|testnet.go|testne
t_multi_node.go)" coverage.out | go tool cover -func=/dev/stdin | grep total |
awk '{print $3}'
```

2. Detailed test coverage

A bit more details about coverage of individual files can be retrieved using other commands:

```
go list ./... | grep -vE
"(simulation|mocks?|testutil|mock_|migrations|events|api|docs|upgrades|sample|null
ify|network|debug|tests|constants|\.pb\.go|\.pb\.gw|\.pulsar|testnet.go|testnet_mu
lti_node.go)" \
  | xargs go test -coverprofile=coverage.out
```

Which displays the following information:

```
ok      zigchain/app    0.074s    coverage: 3.5% of statements
    zigchain/app/keepers        coverage: 0.0% of statements
    zigchain/cmd/zigchaind        coverage: 0.0% of statements
    zigchain/cmd/zigchaind/cmd        coverage: 0.0% of statements
ok      zigchain/wasmbinding    0.097s    coverage: 6.6% of statements [no tests
to run]
?       zigchain/wasmbinding/bindings    [no test files]
    zigchain/x/dex/client/cli        coverage: 0.0% of statements
ok      zigchain/x/dex/keeper    0.132s    coverage: 96.7% of statements
ok      zigchain/x/dex/module    0.071s    coverage: 15.1% of statements
ok      zigchain/x/dex/types    0.079s    coverage: 2.9% of statements
ok      zigchain/x/factory/keeper    0.122s    coverage: 93.0% of statements
ok      zigchain/x/factory/module    0.071s    coverage: 7.7% of statements
ok      zigchain/x/factory/types    0.073s    coverage: 3.5% of statements
    zigchain/x/tokenwrapper/client/cli        coverage: 0.0% of statements
ok      zigchain/x/tokenwrapper/keeper    4.404s    coverage: 91.2% of statements
ok      zigchain/x/tokenwrapper/module    0.102s    coverage: 85.5% of statements
ok      zigchain/x/tokenwrapper/types    0.086s    coverage: 2.5% of statements
ok      zigchain/zutils/validators    0.079s    coverage: 96.2% of statements
```