OpenZeppelin | security

Tesseract Protocol Audit

November 12, 2024

Table of Contents

Summary

Type	DeFi
Timeline	From 2024-09-09 To 2024-09-16
Languages	Solidity

Total Issues	19 (15 resolved, 1 partially resolved)
Critical Severity Issues	0 (0 resolved)
High Severity Issues	2 (2 resolved)
Medium Severity Issues	4 (3 resolved)
Low Severity Issues	4 (2 resolved, 1 partially resolved)
Notes & Additional Information	9 (8 resolved)

Scope

We audited the tesseract-protocol/smart-contracts repository at commit 940fc66.

In scope were the following files:

```
src
├── interfaces
│   ├── ICell.sol
│   └── IYakRouter.sol
└── Cell.sol
├── HopOnlyCell.sol
└── YakSwapCell.sol
```

Update: *Last commit after conclusion of the fix review is 6d6484b, includes all the fixes for the findings in this audit. However, further changes were made that were not in-scope for this audit.*

System Overview

The Tesseract Protocol is a liquidity aggregator whose goal is to provide a comprehensive solution for transferring tokens across the Avalanche L1 ecosystem. It gives users the freedom to specify the paths that they want to go through when moving tokens, while also allowing for optional swaps on intermediate Avalanche L1s. The base contract is the `Cell` contract which provides two main entry points, the `crossChainSwap` function, later renamed to `initiate` at commit 6d6484b, and the `receiveTokens` function. The former is meant to be triggered directly by users who will specify the actions, paths, amounts, and tokens that are meant to be used in the cross-chain swap/transfer. The `receiveTokens` function is meant to be called by bridges whenever an action created from a source chain is being finalized on the destination chain.

Supported actions include `Hop` (a simple token transfer to another chain), `HopAndCall` (a transfer with a contract call on the destination chain), `SwapAndHop` (a swap on the current chain and a transfer to another), and `SwapAndTransfer` (a swap on the current chain followed by a transfer on the same chain to the final recipient).

The `Cell` contract is abstract because it does not implement the internal `_swap` function and the external `route` one. The `HopOnlyCell` and `YakSwapCell` contracts extend from `Cell`, implementing both the necessary functions.

The `HopOnlyCell` is an instance of `Cell` that allows only for hops (cross-chain transfers). For this reason, the implementation of `route` returns an empty path (no route needed when there are no swaps) while the implementation of `_swap` returns the same input values as output.

The `YakSwapCell` contract is also an instance of `Cell` but it is meant to allow for swaps. It will interact with the `YakRouter` contract, both for `route` and for `_swap`.

Security Model and Trust Assumptions

Apart from the assumption that all out-of-scope components behave as expected, the following assumptions regarding the security model of the codebase were made during the audit.

Trades are not meant to happen instantly but are submitted and executed in two separate actions. During this time frame, liquidity across Avalanche L1s might have changed significantly, affecting the trades that have been submitted but not executed yet. Since retrial and recoverability of such trades are neither included in the protocol's design nor within the audit scope, it is up to the protocol team and users to handle these matters.

Finally, the team let us know that in order to provide the correct data input, users will interact with a front-end application that will facilitate proper input data sourcing logic. In the event that the front end is compromised or the user is misguided into a phishing attack, there is a risk of user funds being stolen.

Privileged Roles

There are no privileged roles within the audited codebase, with all functions being callable by any actor with any input. For this reason, the team made us aware of the fact that instances of the `Cell` contract are not meant to hold assets at any moment given the lack of restrictions on its functionalities.

Compatibility

The codebase is fairly simple and general-purpose. However, some special cases and tokens might not be compatible with the current version. Specifically the following:

- Tokens with fees on transfer are not supported.
- Tokens with rebasing mechanisms are not supported. Especially if the rebasing mechanism can be triggered manually in which case opportunities to steal part of user tokens might arise.

- The `TokenHome` or `TokenRemote` contracts with custom logic on approvals and/or transfers might be incompatible.
- Swaps from native to ERC-20 or the other way around are not supported.
  - ***Update:*** *Supported at commit 6d6484b.*
- Tokens that are both native to the Avalanche L1 and ERC-20-compliant at the same time are not supported.
  - ***Update:*** *Supported at commit 6d6484b.*
- Tokens with hooks on transfers and/or approvals are not supported since they can trigger reentrancy scenarios or execute unsupported custom logic.
- The `TokenHome` and `TokenRemote` contracts must be verified to ensure they are not malicious and do not implement undesirable logic, such as using unknown teleporters as data sources or implementing access control functionality that could lead to the theft of users' tokens on the source Avalanche L1. Because any address can be registered as a remote token on the home Avalanche L1, every contract included in the list of possible paths must be verified.

The audit team was also asked to provide the Avalanche L1 properties for the protocol deployment as the client is considering deploying on multiple Avalanche L1s:

- For Avalanche L1s using `NativeTokenRemote` - meaning that the swapped assets are native to the Avalanche L1 - the `NATIVE_MINTER` precompile should be enabled and access for minting should be granted.
- The set of Avalanche L1 validators should be diverse, with most of the validator stake being either distributed among different validators or held by trusted entities. Otherwise, depending on the Avalanche Warp Messaging signature acceptance threshold, the Avalanche L1 validators could transmit malformed messages, potentially disrupting the protocol.
- The code of the Teleporter contract on the Avalanche L1 must be checked to ensure that it is not malicious and that it prevents replay attacks.
- The Avalanche L1 VM should implement all opcodes used by the protocol in the manner expected by the protocol, i.e., without any unintended behavior.
- After deploying the protocol, constant observation of Avalanche L1 upgrades is required; every upgrade should be carefully considered. For example, an Avalanche L1 upgrade may disable a previously enabled precompile, which can lead to the protocol's inability to function or result in funds being locked on source or destination Avalanche L1s.

High Severity

H-01 Native Token Bridge Functionality is Unsupported

The protocol currently supports bridging ERC-20 tokens. In this process, the `BridgeToken` contract approves the `amount` to the `Cell` contract, which then transfers the tokens for further operations.

However, if the token is an ERC-20 token on the origin Avalanche L1 but a native token on the destination or intermediate Avalanche L1, the `TokenHome` contract will attempt to call `Cell.receiveTokens` for native tokens. This call differs in several arguments and uses a different function selector, leading to a revert, as the target contract does not implement a function for handling native token bridge calls or any `fallback` mechanism.

If the `fallbackRecipient` is an account that cannot receive tokens on the Avalanche L1 (for example, on an intermediate Avalanche L1), the tokens may become locked. This happens because the `TokenHome` contract reverts in such cases, as it uses the `sendValue` function from the OpenZeppelin library, which reverts on send failure, as shown here.

Consider implementing functionality to handle native tokens in the bridging process to avoid these scenarios.

Update: *Resolved in pull request #2. The team stated:*

> *This PR addresses the following:*
> 1. *Fixes issue H-01. 2. Incorporates suggestions from N-06 and N-07. Remarks:*
>
> *1. New calldata property: `sourceBridgeIsNative`. - Added due to current limitations deducting this information onchain in a reasonable way. - Temporary solution pending Avalabs update. - Avalabs has committed to addressing this in a future release. - We plan to remove this property once Avalabs implements their fix. 2. Token handling approach: - Received native tokens are immediately wrapped. - This may result in some transactions where tokens are wrapped and unwrapped in the same transaction, potentially increasing gas consumption. - Decision made to prioritize code readability and maintainability over gas optimization in these cases.*

H-02 Cross-Function Reentrancy in `YakSwapCell`

The protocol is vulnerable to being drained (either accumulated dust, tokens sent to the contract or tokens that might be stuck for any other reason) through cross-function reentrancy specifically on the functions `crossChainSwap` and `receiveTokens` that might be called within the same execution, by anyone, and in any order.

Let's start with the `crossChainSwap` function being called first with a `0` `tokenIn` amount. Initially, the contract tracks a balance of `X` `tokenOut`. Then, `YakRouter.swapNoSplit` is invoked, where a malicious adapter can be specified. Let's imagine now that a malicious adapter reenters the contract by invoking the `receiveTokens` function with `amount == X`, `hop == 0`, `tokenOut` address as `address(0)` and a `SwapAndTransfer` action. As a result

- The contract balance becomes `2X`.
- The `swap` operation intentionally fails on the `tokenOut` check, returning a `false` status flag.
- Due to the `receiveTokens` function call with `hop == 0`, and since the contract increments the hop before calling the `_route` function, the code enters the branch `if (payload.hop == 1)` and approves the `amount` for transfer to an arbitrary contract. However, the tokens must **not** be transferred at this step, as they will be needed later in the call sequence.

Once the reentrant call ends, upon returning to the main call, the contract recalculates its balance, resulting in a total of `2X`. It then transfers `X` tokens to the receiver, while leaving the approval for `X` tokens in place for the attacker's contract, allowing the attacker to steal the remaining `X` amount.

The secret gist with the PoC was created to demonstrate the actual scenario.

Consider using the `nonReentrant` modifier from the OpenZeppelin library with the `crossChainSwap` and `receiveTokens` functions.

Update: *Resolved in pull request #3. The team added the `nonReentrant` modifier.*

Medium Severity

M-01 Zero Fee in `YakSwapCell._swap` Causes Revert When Fee Is Enabled

In the `_swap` function of the `YakSwapCell` contract, the `fee` argument passed to `YakRouter.swapNoSplit` is set to 0. However, if the fee is enabled in the router (i.e., `MIN_FEE > 0`), the transaction will always revert in the `_applyFee` function.

Consider adding the fee as a function argument to allow the protocol to function when the fee is activated in the router.

Update: *Resolved in pull request #4. The team added an extra fee parameter both in the encoding and decoding steps*

M-02 Incompatibility with Tokens That Implement Approval Race Protection

Some tokens, such as USDT, include an approval race protection mechanism, requiring the allowance to be set to 0 before calling the `approve` function again.

In the protocol, the `approve` function is used to increase allowances for token bridges. If any dust allowance remains after these calls, it may prevent future token bridging.

Consider using the `forceApprove` function from the OpenZeppelin library to handle this scenario.

Update: *Resolved in pull request #5.*

M-03 ABI Decode Might Produce Unexpected Reverts

Whenever the codebase executes `abi.decode` in the `Cell` or `YakSwapCell` contract it is assuming that the decoding data is actually encoded correctly. This might be obvious in the `receiveFunction` since it is supposed to be called by a contract which is assumed to behaves correctly, but it might not be true in the `route` and `_swap` functions.

In particular, the `_swap` function might revert if decoding bad data. If that happens, tokens might be lost since the entire `_trySwap` function would fail and the assumption that `_swap` does not fail will be broken.

Consider managing eventual throws of the `abi.decode` call in the `_swap` function and thinking about the remaining examples on whether they need special attention and/or explicit documentation.

Update: *Acknowledged, not resolved. The team stated:*

> *In the current version of Solidity, there is no way to implement a try/catch mechanism for `abi.decode`. While this is unfortunate for users who might accidentally pass incorrect data while attempting a swap, any resulting revert will be handled by the sending bridge. In such cases, the tokens will be sent to the fallback receiver. The route function is not called onchain by the protocol itself. The risk that users could potentially end up with tokens on an intermediate chain will be clearly stated in the frontend interface and documentation.*

M-04 Fixed Gas Limit in Single-Hop Transfers

In the `_send` and `_trySwap` functions of the `Cell` contract, a fixed amount of gas is used for single-hop send operations. A relayer on the destination Avalanche L1 must provide at least `GAS_LIMIT_BRIDGE_HOP` amount of gas for the teleporter call. This gas is used for the execution of the `TokenHome` or `TokenRemote` contracts and the underlying asset transfer when the destination chain is the home for the token. If the call fails with this gas limit, the receiver does not receive the tokens, yet the call is considered successful, and the relayer receives the reward.

In particular, there are two cases where the current fixed amount of gas could be insufficient:

- The destination chain is the home for the token, and its transfer uses gas-intensive logic (e.g., hooks, rebasing functionality), requiring more gas than the remaining amount.
- The asset is a native token on the destination chain, and the receiver is a contract that executes additional logic upon receiving transfers, consuming more gas than provided.

In these cases, manual recovery of funds is possible by calling the `TeleporterMessenger.retryMessageExecution` function. An account must call this function and provide enough gas for execution, spending their own funds in addition to the fee already paid to the relayer.

Consider providing users with the ability to specify the gas limit for single-hop send operations.

Update: *Resolved in pull request #6.*

Low Severity

L-01 Incorrect Handling of `secondaryFee` for Single Hops

In the `_send` and `_sendAndCall` functions of the `Cell` contract, when creating the `input` struct, the `secondaryFee` field can accept arbitrary values. For multihop scenarios, it may be non-zero, but for single hop cases, it must always be 0.

Consider enforcing `secondaryFee` to be 0 for single hops.

Update: *Resolved in pull request #7. The team added a conditional expression to set the correct value in case of single hop.*

L-02 Unrestricted Access to the `receiveTokens`

The `receiveTokens` function in the `Cell` contract is intended to be called by a token bridge contract. However, it also allows direct calls to the function, which can result in the `CellReceivedTokens` event being triggered repeatedly, potentially misleading off-chain observers.

Consider reviewing the `receiveTokens` function, such as implementing access restrictions (e.g., whitelisting token bridges) or removing the event emission.

Update: *Acknowledged, not resolved. The team stated:*

> *We are considering adding access restrictions once the previously mentioned `BridgePathRegistry` is ready for deployment. We recognize the importance of protecting this to prevent potential griefing attacks, particularly on Avalanche L1s where transactions might be free or very cheap. If we find that we have sufficient logs for analytics and other purposes without the event, we will remove it.*

L-03 Lack of Input Validation

All major functions within the codebase lack of input validation and are generally callable by any user. Even if this does not immediately reflects specific issues, there are many different ways in which unexpected errors or result might arise. In particular:

- Not all values are checked to be non zero or within bounds. Moreover, unchecked amounts of tokens or for other variables like `gasLimit` might produce unexpected underflows or overflows which are not correctly handled.
- The `crossChainSwap` function suggests that the action being passed implies a swap, but this is not checked to do so.
- If `token` and `tokenOut` in `YakSwapCell._swap` are the same, then user will lose tokens, since `IERC20(tokenOut).balanceOf(address(this)) - balanceBefore` will be the same and `amountOut = 0`. Additionally, if fees are taken out of the swap, `balanceAfter` can be even lower than `balanceBefore` and make the transaction to revert.

Following assumption that explicit code is better code and the principle of failing early and loud, consider adding relative sanity checks in place to avoid wrong inputs to create unexpected outcomes.

Update: *Partially resolved in pull request #8. The team changed the name of the `crossChainSwap`, improved the control flow in case `tokenOut == token`, and added a zero amount check in the `amount` input parameter of the `crossChainSwap` function.*

L-04 `SwapFailed` Event Redundancy

The `SwapFailed` event is emitted only when a swap fails and `payload.hop == 1`. However, this event adds no value, as the same information can be obtained from other events. For instance, the `Rollback` event, which is emitted under the same conditions, includes the `token` address, and the `amount` is emitted by the bridge contract.

Consider removing the `SwapFailed` event if it is not needed.

Update: *Resolved in pull request #9.*

Notes & Additional Information

N-01 Use of Custom Errors

Since Solidity version 0.8.4, custom errors provide a cleaner and more cost-efficient way to explain to users why an operation failed. Multiple instances of revert and/or require messages were found within the `Cell` contract.

- The require statement with the message "Invalid fee".
- The revert statement with the message "Swap failed".

For conciseness and gas savings, consider replacing require and revert messages with custom errors.

Update: *Resolved in pull request #10.*

N-02 Non Explicit Import Are Used

In the codebase, non explicit imports are used in all main contracts and files.

In order to improve readability and explicitness, consider importing using explicit imports in the form `import {...} from "..."`.

Update: *Resolved in pull request #11.*

N-03 Missing Validation for Hop Index

In the `_route` function of the `Cell` contract, the payload for the current hop is retrieved based on the `payload.hop` value. However, this index is not validated, which could cause a revert with panic error "Array Out of Bounds".

Consider validating that `payload.hop` is within the range of `payload.instructions.hops` array.

Update: *Resolved at commit 6d6484b, with the removal of Hop Index.*

N-04 Unused Import

In the `Cell.sol` file, the `TeleporterRegistry` is imported but never used.

Consider removing this unused import.

Update: *Resolved in pull request #13.*

N-05 Redundant Indexing of `amount` in `Rollback` Event

Indexing the `uint256 amount` parameter in the `Rollback` event does not provide any meaningful benefit.

Consider removing the `indexed` keyword from the `amount` argument to reduce gas consumption, unless it is required for an off-chain component.

Update: *Resolved in pull request #14.*

N-06 Excessive Data in Multihop Payload Transmission

The protocol uses a struct containing the `Hop[] hops` field for data transmission, where each element in the array serves as the data for the next hop.

In multihop scenarios, the target contract on the next chain is also a variant of `Cell`, and the current payload is passed along to the next call. This results in excessive data being transmitted between chains. For example, in a scenario with five hops, by the fourth hop, the data from the second hop becomes irrelevant but still incurs additional gas costs as it is passed to subsequent chains.

Consider reducing the payload size at each hop, as data from previous hops is no longer necessary. However, it is necessary to retain the information from the starting chain to ensure rollback functionality in the `_trySwap` function. It is also possible to remove the `CellPayload` struct and retain only the `Instructions` struct, since the data for the next hop is always in the first element of the array. This will reduce costs for users by transmitting fewer bytes between chains. Additionally, consider removing the `isMultiHop` flag, as this functionality can be implemented in the contract by querying the `bridgeSourceChain` and analyzing the retrieved data.

Update: *Resolved in pull request #2. The team stated:*

> *Fixed with H-01*

N-07 Excessive Complexity of the `Cell` Contract

The `Cell` contract currently has unnecessarily complex control flow. Two main areas could be revisited to improve the overall clarity of the codebase.

Firstly, the chain of `if`/`else` statements in the `_route` function contains nested `if`/`else if` statements within the first `else` block. Consider flattening these nested statements into root-level `else if` statements to enhance readability. Additionally, since the function ultimately calls either `_send` or `_sendAndCall`, it might be more efficient to assign the appropriate function to a local variable and call it once at the end, reducing repetition.

Secondly, the `_trySwap` function could be simplified or possibly removed. In case of a swap failure, it performs rollback functionality, which could be moved to a separate function to streamline the logic.

Consider integrating these changes to enhance the overall clarity and readability of the code.

Update: *Resolved in pull request #2. The team stated:*

> *Fixed with H-01*

N-08 Redundant Value Returned by `route` Function in `YakSwapCell` Contract

The `route` function in the `YakSwapCell` contract returns `trade` data, which already includes the `offer.gasEstimate` value, yet it also separately returns `offer.gasEstimate`, resulting in redundancy.

Consider returning only the `trade` variable to avoid duplicating information.

Update: *Acknowledged, not resolved. The team stated:*

> *While this implementation handles the trade parameter in a specific way, other `Cell` implementations may use it differently. The trade return parameter can be freely customized by concrete `Cell` developers and will be passed through unchanged.*

> *However, the `gasEstimate` return parameter is specifically required by users (Front-End) to properly set gas limits for ICTT operations.*

N-09 Potential Panic Error in `YakSwapCell` Contract's `route` Function

The `route` function in the `YakSwapCell` contract does not verify that `extras.slippageBips` is less than or equal to the constant `BIPS_DIVISOR` (set to `10,000`). This may lead to an underflow, causing a potential panic error.

Consider validating the value of `extras.slippageBips`.

Update: *Resolved at commit 0126cde. The team added the proper check, adding also a limitation to the slippage tolerance that needs now to be strictly less than 100%.*

Conclusion

The Tesseract Protocol provides users with aggregate liquidity from every Avalanche L1 in the Avalanche ecosystem, enabling cross-chain swaps using Avalanche ICTT. Two high-severity issues were reported, along with four medium ones. The general design is simple and straightforward, allowing the protocol to scale by adding new strategies called `Cells`. Additionally, we have suggested several design improvements to enhance codebase clarity and maintainability. The Tesseract team was very responsive throughout the audit.

Appendix

Out of Scope Issues

Relayers Can Manipulate Transactions for Profit

When a user calls the `Cell` contract, it produces a call to the token bridge contract (e.g., `TokenRemote`), which then calls the `Teleporter` contract (here and here). The `Teleporter` uses Warp Messaging precompile to convey the message to the destination chain. A relayer then calls the `Teleporter` contract on the destination chain, providing the message data to be executed, which validates and executes the message. One of the parameters that can be passed to the `Teleporter` is an allowlist for relayers. The `TokenHome` and `TokenRemote` contracts are currently specifying the allowlist as empty, allowing anyone to be a relayer.

A dishonest user can track transactions and check them for potential profit. For example, consider two cases:

- The relayer sees the `SwapAndTransfer` hop and checks whether pool manipulation is profitable for them. If so, they sandwich the swap and gain profit from the user's swap. This scenario is more controllable since the user can manage the `amountOut` argument. It is also worth mentioning front-running and back-running opportunities not only from the relayer but also from other users, since even the relayer's transaction could be subject to MEV.
- The relayer sees the `SwapAndHop` hop with `payload.hop == 0` (which will be increased to 1 during hop execution) and determines which scenario is more profitable: using the method mentioned above or forcing the user's transaction to fail due to the `amountOut` after the swap (using pool manipulation). This would trigger the rollback mechanism, sending tokens back to the source subnet. Then, the dishonest relayer picks up the rollback hop and receives fees gained maliciously, even if normally, the user's swap would have succeeded.

Consider reviewing the protocol design, especially the rollback mechanism and consider whether mitigations can be applied.