verichains

SECURITY AUDIT OF

UNIZEN DEX AGGREGATOR

Public Report

Apr 11, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Ethereum	An opensource platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.
Stargate	Stargate is a community-driven organization building the first fully composable native asset bridge, and the first dApp built on LayerZero. Stargate's vision is to make cross-chain liquidity transfer a seamless, single transaction process.
Layerzero	LayerZero is an interoperability protocol that connects blockchains, allowing developers to build seamless omnichain applications, tokens, and experiences. The protocol relies on immutable on-chain endpoints, a configurable Security Stack, and a permissionless set of Executors to transfer censorship-resistant messages between chains.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Apr 11, 2024. We would like to thank the Unizen for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Unizen Dex Aggregator. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team identified some vulnerable issues in the contract code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Unizen Dex Aggregator

The Unizen Ecosystem is housing and aggregating trades across trusted first and third-party exchange modules. Unizen is currently powered by the below liquidity providers but is continuously adding on more sources of liquidity.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the Unizen Dex Aggregator. It was conducted on commit `b7e50eb2aa37d1d9f8127a29dbc18a41ccdda1ce` from git repository *https://github.com/unizen-io/unizen-trade-aggregator*

The latest version of the following files were made available in the course of the review:

SHA-1 Sum	File
230f5a44f75132f9b537db68c40257a5000beb4c1caa67f66d8e07713d7658c3	contract/UnizenDexAggr.sol
21d509663917c82225b39bf5f70b635e004106f5a28fb3c11024bbb2df04a2c4	contract/UnizenDexAggrETH.sol

The Unizen team has been updated with the findings and recommendations. The team has acknowledged the issues and provided updates on the issues. The team has also responded to the findings and recommendations. The latest version of the code was reviewed on commit `aa8539dfa6ecd70fe3fba958824fa7eb11495507`.

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert

- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Unizen acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. Unizen understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, Unizen agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the Unizen will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the Unizen, the final report will be considered fully accepted by the Unizen without the signature.

2. AUDIT RESULT

2.1. Overview

The Unizen Dex Aggregator is developed in the `Solidity` language, adhering to version `^0.8.0` requirements, and built upon the OpenZeppelin library.

The contracts `UnizenDexAggr` and `UnizenDexAggrEth` share identical primary functionalities. These include aggregating swaps, facilitating cross-chain swaps, and enabling integrators to earn fees.

2.1.1. Aggregator swaps

Both contracts serve as aggregators for batch swap calls from various decentralized exchanges (DEXs). They support multiple tokens, including native tokens, and operate within a whitelist of approved exchanges.

2.1.2. Cross-Chain Aggregator Swaps via Stargate

Similar to aggregator swaps, cross-chain aggregator swaps facilitate batch swaps across multiple DEXs on different chains. These aggregators operate on both local and remote chains, leveraging a whitelist of supported exchanges. Notably, cross-chain swaps are executed via the Stargate, enabling seamless liquidity across multiple chains and benefiting from Instant Guaranteed Finality (IGF).

2.1.3. Integrator Fee Earnings

Integrators partnering with the aggregator earn fees based on the volume of swaps conducted by users, whether on a single chain or across multiple chains. This fee calculation is directly tied to the transaction volume, incentivizing integrators to contribute to the ecosystem's liquidity and functionality.

2.2. Findings

#	Issue	Severity	Status
1	Does not verify calls.data is match with the srcToken	CRITICAL	Fixed
2	Users take advantage of the balance of the contract to swap	CRITICAL	Fixed
3	Reentrancy in `integratorsWithdrawPS`	CRITICAL	Fixed
4	Missed check sum of amounts of calls matches swapInfo.amount	HIGH	Fixed
5	Missed Check If `_srcAddress` and `_chainId` Are The Trusted Source	HIGH	Fixed
6	Lack of Verification for msg.value to Cover Fee	HIGH	Fixed
7	Function availability during contract pause	LOW	Fixed
8	Missing zero check in setters	INFORMATIVE	Acknowledged
9	Does not emit event in setters	INFORMATIVE	Acknowledged
10	Clarification of _nonce parameter usage	INFORMATIVE	Acknowledged
11	Redundant codes	INFORMATIVE	Acknowledged

2.2.1. Does not verify `calls.data` is match with the `srcToken` - CRITICAL

Affected files:

- contracts/UnizenDexAggr.sol
- contracts/UnizenDexAggrETH.sol

The `swapSTG` function allows users deposit their `srcToken` into the contract and arbitrary data in `dstCalls` to swapping in the `targetExchange`

A vulnerability arises from the fact that the contract does not verify whether the data in `dstCalls` matches the `srcToken`.

This issue can lead to unexpected behavior, such as users deposit meme token and swap another token in the `targetExchange`. This can result in a loss of funds or other unintended consequences.

```
if (!swapInfo.isFromNative) {
    srcToken.safeTransferFrom(msg.sender, address(this), swapInfo.amount);
}
// ...
for (uint8 i = 0; i < calls.length;) {
    require(isWhiteListedDex(calls[i].targetExchange), "Not-verified-dex");
    {
        bool success;
        if (calls[i].sellToken == address(0)) {
            // trade ETH
            success = _executeTrade(calls[i].targetExchange, calls[i].amount,
calls[i].data);
        } else {
            success = _executeTrade(calls[i].targetExchange, 0, calls[i].data);
        }
        require(success, "Call-Failed");
    }
}
```

UPDATES

- *9 Apr 2024*, The team has implemented a check to verify that the `_srcToken` is the same as the `calls[0].sellToken`, ensuring the transferred token into the contract matches the first of the call batch.

2.2.2. Users take advantage of the balance of the contract to swap - CRITICAL

Affected files

- contracts/UnizenDexAggr.sol:230
- contracts/UnizenDexAggrETH.sol:228

The contract's swapping functionality on the destination chain presents a critical vulnerability. When executing swaps according to the `dstCalls` parameter, if any call within them fails, users receive their `amountLD` token back. However, successful swaps prior to the failure are not reverted. This vulnerability allows users to exploit the contract's balance for swapping without losing any tokens.

```
for (uint8 i = 0; i < dstCalls.length; ) {
    if (!_executeTrade(dstCalls[i].targetExchange, 0, dstCalls[i].data)) {
        IERC20(_token).safeTransfer(user, amountLD);
        emit CrossChainSwapped(_chainId, user, amountLD, 0);
        return;
    }
}
```

UPDATES

- *9 Apr 2024*: The team updated code that use try/catch block to handle any reverts and refunds all a swap amount to users.

2.2.3. Reentrancy in `integratorsWithdrawPS` - CRITICAL

- Affected file: contracts/UnizenDexAggr.sol

The `integratorsWithdrawPS` function poses a vulnerability due to potential reentrancy attacks. This function allows integrators to receive fees and can be called by anyone. However, the integrators are not inherently trusted, even if they are added by the contract owner.

The vulnerability arises when integrators call the `withdrawFee` function and exploit reentrancy by repeatedly invoking the sendFee function using low-level call methods. This enables the integrators to drain all native tokens from the contract, leading to a loss of funds.

```
function integratorsWithdrawPS(address[] calldata tokens) external {
    address integratorAddr = msg.sender;
    // ...
    if (integratorPSEarned[integratorAddr][address(0)] > 0) {
        integratorAddr.call{value: integratorPSEarned[integratorAddr][address(0)]}(""); //
ability callback here
        integratorClaimed[integratorAddr][address(0)] +=
integratorPSEarned[integratorAddr][address(0)];
        integratorPSEarned[integratorAddr][address(0)] = 0;
    }
}
```

RECOMMENDATION

To mitigate this vulnerability, it is imperative to update the `integratorPSEarned` state before transferring fees to the integrators. This ensures that the state is updated prior to any external calls, preventing reentrancy attacks. At the same time also using `nonReentrant` modifier, and it always trusted integrator addresses.

UPDATE

- *5 Apr 2024*: This vulnerability is critical due to the potential compromise of integrator addresses. Attackers could exploit this weakness to deploy new code to these addresses or carry out unauthorized actions without the admin's knowledge or consent. And by the way, the integrator addresses are not inherently trusted, even if they are added by the contract owner.
- *8 Apr 2024*: The team has implemented the `nonReentrant` modifier to prevent reentrancy attacks in the `integratorsWithdrawPS` function. This update enhances the contract's security by ensuring that integrators cannot exploit the vulnerability to drain native tokens from the contract.

2.2.4. Missed check sum of amounts of `calls` matches `swapInfo.amount` - HIGH

Affected files:

- contracts/UnizenDexAggr.sol:132
- contracts/UnizenDexAggrETH.sol:131

Users need to pass the amount of tokens to swap via the `swapInfo.amount` parameter. After that, the contract will swap these tokens for other tokens in the `targetExchange` via the `calls` parameter. The contract does not verify whether the sum of the amounts of `calls` matches the `swapInfo.amount`. This can lead to unexpected behavior, such as when the fee is taken and the sum of the `calls`'s amount is not enough to trade, the contract will revert the transaction. This is a critical issue because the trade will use the fees of integrators to pay, and the integrators will lose their funds.

```
function swapSTG(
        CrossChainSwapSg memory swapInfo,
        SwapCall[] memory calls,
        SwapCall[] memory dstCalls
    ) external payable nonReentrant {
    if (!swapInfo.isFromNative) {
        srcToken.safeTransferFrom(msg.sender, address(this), swapInfo.amount);
    } else {
        require(msg.value >= swapInfo.amount + swapInfo.nativeFee, "Invalid-amount");
    }
    if (bytes(swapInfo.uuid).length != 0 && integratorFees[swapInfo.uuid] != 0) {
```

```
        _takeIntegratorFee(swapInfo.uuid, swapInfo.isFromNative, srcToken,
swapInfo.amount);
    }
    for (uint8 i = 0; i < calls.length;) {
        // ...
        {
            bool success;
            if (calls[i].sellToken == address(0)) {
                // vulnerable code
                success = _executeTrade(calls[i].targetExchange, calls[i].amount,
calls[i].data);
            } else {
                success = _executeTrade(calls[i].targetExchange, 0, calls[i].data);
            }
            require(success, "Call-Failed");
        }
    }
    // ...

}
```

UPDATES

9 Apr 2024: The team has implemented a check to verify that the sum of amounts of `calls` matches the `swapInfo.amount`.

2.2.5. Missed check if `_srcAddress` and `_chainId` are the trusted source - HIGH

Affected file:

- contracts/UnizenDexAggr.sol
- contracts/UnizenDexAggrETH.sol

The `sgReceive` function is accessible to the Stargate router, Stargate address and anyone if they access to the Stargate router from a source chain.

Typically, this function should only be callable by a designated source endpoint, with the `_srcAddress` parameter representing the sender on the source chain.

However, in the current implementation, the `_srcAddress` parameter is not being verified to ensure it originates from a trusted caller. This oversight poses a critical security risk, as it allows arbitrary callers to invoke the `sgReceive` function via the Stargate Router and potentially inject malicious data into parameters, leading to potential system compromises.

Like the `_srcAddress`, the `_chainId` should be validated in a trusted source chains cause the chain id can be faked.

RECOMMENDATION

Implement a check to ensure that `_srcAddress` originates from a whitelist of trusted addresses. This verification step is crucial for securing the `sgReceive` function and preventing unauthorized access.

Below is an example snippet demonstrating how this check can be implemented: OmmichainFungibleToken example by Stargate.

UPDATES

- *5 Apr 2024*: The `_srcAddress` and `_chainId` parameters should be validated to ensure they originate from a trusted source. This validation step is crucial for securing the `sgReceive` function and preventing unauthorized access. We agree that the issue is complexity to exploit but it is a good practice to validate the source of the data to prevent potential attacks.
- *9 Apr 2024*: The team has implemented a check to verify that the `_srcAddress` and `_chainId` originate from a trusted source. This update enhances the contract's security by ensuring that only trusted sources can invoke the `sgReceive` function.

2.2.6. Lack of verification for `msg.value` to cover fee - HIGH

- Affected file:
  - contracts/UnizenDexAggr.sol
  - contracts/UnizenDexAggrETH.sol

The contract facilitates asset swapping across multiple chains, with the Stargate router requiring a specific fee to cover gas costs at the destination chain. Users bear this fee when conducting transactions from the source chain.

However, a critical vulnerability exists where the `msg.value` is not verified to ensure it exceeds the `nativeFee` when users swap tokens (non-native tokens). Leading to two potential scenarios:

- Transactions failing at the destination contract due to insufficient `msg.value` to cover gas costs.
- Transactions consuming available funds from the contract

```
function swapSTG(
        CrossChainSwapSg memory swapInfo,
        SwapCall[] memory calls,
        SwapCall[] memory dstCalls
) external payable nonReentrant {
    // ...
    if (!swapInfo.isFromNative) {
```

```
        srcToken.safeTransferFrom(msg.sender, address(this), swapInfo.amount);
    } else {
        require(msg.value >= swapInfo.amount + swapInfo.nativeFee, "Invalid-amount");
    }
// ...
_sendCrossChain(
        swapInfo.dstChain,
        swapInfo.srcPool,
        swapInfo.dstPool,
        msg.sender,
        swapInfo.nativeFee, // specific fee
        swapInfo.amount,
        dstCalls.length == 0 ? swapInfo.receiver : destAddr[swapInfo.dstChain],
        swapInfo.gasDstChain,
        payload
    );
}
```

RECOMMENDATION

To mitigate this vulnerability, it should verify the `msg.value` exceeds the `nativeFee` when users swap tokens.

UPDATES

- *9 Apr 2024*, In special cases, when a user swaps the `srcToken` (not a native token), the user should transfer the amount of token and not transfer any ETH into the contract. The cross-chain swap needs some ETH to pay for the fee; the contract currently holds ETH (maybe it is a fee of integrators). This swap will use this ETH to pay the fee. Finally, the contract will lose funds, and users will take advantage of this.

To mitigation the bug, please verify `msg.value` is greater or equal to the fee of cross-swap. An example patching code:

```
function swapSTG(
        CrossChainSwapSg memory swapInfo,
        SwapCall[] memory calls,
        SwapCall[] memory dstCalls
) external payable nonReentrant {
    // ...
    if (!swapInfo.isFromNative) {
        srcToken.safeTransferFrom(msg.sender, address(this), swapInfo.amount);
        require(swapInfo.nativeFee <= msg.value, "Insufficient-fee"); // patch
    } else {
        require(msg.value >= swapInfo.amount + swapInfo.nativeFee, "Invalid-amount");
    }
// ...
_sendCrossChain(
        swapInfo.dstChain,
```

```
        swapInfo.srcPool,
        swapInfo.dstPool,
        msg.sender,
        swapInfo.nativeFee, // specific fee
        swapInfo.amount,
        dstCalls.length == 0 ? swapInfo.receiver : destAddr[swapInfo.dstChain],
        swapInfo.gasDstChain,
        payload
    );
}
```

- *10 Apr 2024*: The team has implemented a check to verify that the `msg.value` exceeds the `nativeFee` when users swap tokens. This update enhances the contract's security by ensuring that users provide sufficient ETH to cover the gas costs of the transaction.

2.2.7. Function availability during contract pause - LOW

- Affected file:
  - contracts/UnizenDexAggr.sol:104
  - contracts/UnizenDexAggrETH.sol:105

The `swapSTG` function remains accessible for swapping assets even when the contract is paused due to incidents or emergencies. This poses a security vulnerability as it allows users to perform transactions that may have unintended consequences while the contract is in a paused state.

```
function swapSTG(
        CrossChainSwapSg memory swapInfo,
        SwapCall[] memory calls,
        SwapCall[] memory dstCalls
    ) external payable nonReentrant {
    //...
}
```

RECOMMENDATION

Add the `whenNotPaused` modifier to the `swapSTG` function to restrict its availability when the contract is paused.

UPDATES

- *10 Apr 2024*: The team has implemented the `whenNotPaused` modifier to restrict the availability of the `swapSTG` function when the contract is paused. This update enhances the contract's security by preventing users from performing transactions during a paused state.

2.2.8. Missing zero check in setters - INFORMATIVE

- Affected file:
  - contracts/UnizenDexAggr.sol
  - contracts/UnizenDexAggrETH.sol

The contract does not validate zero addresses, potentially allowing for unintended operations or vulnerabilities.

We suggest adding a require statement to check that the address is not zero.

RECOMMENDATION

Check that the address is not zero.

UPDATES

- *11 Apr 2024*: The team has acknowledged the issue.

2.2.9. Does not emit event in setters - INFORMATIVE

- Affected file:
  - contracts/UnizenDexAggr.sol
  - contracts/UnizenDexAggrETH.sol

Functions for mutating storage should emit events to facilitate off-chain monitoring. Events provide transparent, real-time insights into contract state changes, enabling stakeholders to track actions and detect anomalies. Without event emission, contract behavior becomes opaque, hindering monitoring and increasing the risk of errors or exploitation.

We suggest the team should emit events for transparent monitoring.

UPDATES

- *11 Apr 2024*: The team has acknowledged the issue.

2.2.10. Clarification of `_nonce` parameter usage - INFORMATIVE

Affected files:

- contracts/UnizenDexAggr.sol
- contracts/UnizenDexAggrETH.sol

The `_nonce` parameter, obtained from the Stargate router within the `sgReceive` function, serves multiple purposes including transaction order tracking, error handling, and replay protection.

However, in its current implementation, the `_nonce` parameter is utilized in an unclear and potentially incorrect manner. It is used for calculating the remaining balance of tokens and facilitating refunds to users.

```
_nonce = IERC20(_token).balanceOf(address(this)) + amountLD -
contractStatus.balanceSrcBefore;
    if (_nonce > 0) {
        IERC20(_token).safeTransfer(user, _nonce);
    }

    if (dstToken == address(0)) {
        // trade to ETH
        contractStatus.balanceDstAfter = address(this).balance; // eth balance of contract
        _nonce = contractStatus.balanceDstAfter - contractStatus.balanceDstBefore;
        if (_nonce > 0) {
            payable(user).sendValue(_nonce);
        }
    } else {
        contractStatus.balanceDstAfter = IERC20(dstToken).balanceOf(address(this));
        _nonce = contractStatus.balanceDstAfter - contractStatus.balanceDstBefore;
        if (_nonce > 0) {
            IERC20(dstToken).safeTransfer(user, _nonce);
        }
    }
```

RECOMMENDATION

To mitigate confusion and ensure proper usage, it is recommended to rename the `_nonce` parameter and clarify its intended purpose. This will help maintain code clarity and prevent unintended behaviors associated with its usage.

UPDATES

- *11 Apr 2024*: The team has acknowledged the issue.

2.2.11. Redundant codes - INFORMATIVE

Affected files:

- contracts/UnizenDexAggr.sol
- contracts/UnizenDexAggrETH.sol

States are not used in the contract, so they are redundant and should be removed to reduce gas costs and simplify the codebase.

```
import {ILayerZeroReceiver} from "./interfaces/ILayerZeroReceiver.sol";
import {ILayerZeroEndpoint} from "./interfaces/ILayerZeroEndpoint.sol";
import {ILayerZeroUserApplicationConfig} from
"./interfaces/ILayerZeroUserApplicationConfig.sol";
import {IFeeClaimer} from "./interfaces/IFeeClaimers.sol"

address public stargateRouter;
address public layerZeroEndpoint;
address public stable;
uint16 public stableDecimal;
mapping(uint16 => uint16) public chainStableDecimal;
mapping(uint16 => bytes) public trustedRemoteLookup;
uint256 public dstGas;
address public vipOracle;
uint256 public tradingFee;
uint256 public vipFee;
address public treasury;
address public feeClaimer;

function setFeeClaimer(address feeClaimerAddr) external onlyOwner {
    feeClaimer = feeClaimerAddr;
}
```

The approval to zero is not necessary, just approve the amount to the targetExchange

```
function sgReceive(
        uint16 _chainId,
        bytes memory _srcAddress,
        uint256 _nonce,
        address _token,
        uint256 amountLD,
        bytes memory payload
    ) external override {
    sellToken.safeApprove(dstCalls[i].targetExchange, 0);
    sellToken.safeApprove(dstCalls[i].targetExchange, dstCalls[i].amount);
}
```

UPDATES

- *8 Apr 2024*: The team replied that the contract is upgradable smart contract, so team needs to keep that variables slot as its already in storage, else it will cause override slot with wrong data, because some token, they don't allow to approve again, have to set to 0 then approve another amount. It more safe to keep that way some new tokens standard, they are using `increaseAllowance` or `decreaseAllowance` but it same logic in side.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Apr 11, 2024*	Public Report	Verichains Lab

Table 2. Report versions history