shieldify

TOKI

Bridge

SECURITY REVIEW

Date: 12 December 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and have secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Vyper, Rust, Cairo, Move and Go.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About TOKI - Bridge

TOKI Bridge is an IBC-based system for native cross-chain transfers. It relies on a multi-prover security model built on light-client verification supported by trusted execution environments and zero-knowledge proofs. This architecture reduces external trust assumptions and improves auditability.

Transfer

With Transfer, users can effortlessly swap native tokens between any blockchain with Guaranteed Finality. In addition, considering TOKI's high security, transactions can be processed at the fastest and cheapest level, with any transaction of size or chain.

Rebalance Mechanism

To facilitate seamless and successful token swaps between liquidity pools, TOKI ensures adequate balances in each pool. Swaps can shift balances, depleting destination chain pools while increasing source chain pools. To address this, TOKI employs the rebalance mechanism that incentivizes swaps that restore balance and discourages actions that would deplete pool reserves.

Gas Refuel

To enable additional transactions on the destination chain after a swap via TOKI, TOKI offers a gas delivery service. This allows users to transfer native gas tokens from the source chain to the destination chain, with multiple options.

Learn more about Toki's concept and the technicalities behind it: here

4. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1 Impact

- **High** – results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** – results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** – losses will be limited but bearable – and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** – almost certain to happen and highly lucrative for execution by malicious actors
- **Medium** – still relatively likely, although only conditionally possible
- **Low** – requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 7 days with a total of 112 hours dedicated to the audit by two researchers from the Shieldify team.

Overall, the code is well-written. The audit report contributed by identifying one High, five Medium and six Low severity issues. They're mainly related to gaps in validation, insufficient safeguards around payload handling, and flaws in logic that allow malformed inputs or state drift to undermine protocol guarantees.

The Toki team has done a great job with their test suite and provided support and responses to all of the questions that the Shieldify researchers had.

5.1 Protocol Summary

Project Name	TOKI – Bridge
Repository	toki-bridge-evm-contracts-poc
Type of Project	DeFi, Bridge
Security Review Timeline	7 days
Review Commit Hash	8a6dc6ce8b8efaaca793b4e87a7371459c4d5395
Fixes Review Commit Hash	6095a7d4920d1c1ed8a7b6a85905fd0b8b998920

5.2 Scope

The following smart contracts were in the scope of the security review (for bridge v1.1, only the differences from the original v1 were in scope, not the whole contracts):

File	nSLOC
src/dex/DexBridgeRouteUniswap.sol	383
src/dex/UniswapLibraries.sol	294
src/dex/DexReceiverUniswap.sol	420
src/bridge/v1.1/BridgeManagerV1_1.sol	117
src/bridge/v1.1/BridgeQuerierV1_1.sol	67
src/bridge/v1.1/BridgeBaseV1_1.sol	306
src/bridge/v1.1/BridgeV1_1.sol	645
src/bridge/v1.1/BridgeFallbackV1_1.sol	335
src/replaceable/RelayerFeeCalculator.sol	142
src/replaceable/TransferPoolFeeCalculator.sol	266
src/replaceable/TransferPoolFeeCalculatorV2.sol	119
Total	**3094**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **High** issues: **1**
- **Medium** issues: **5**
- **Low** issues: **6**
- **Info** issues: **2**

ID	Title	Severity	Status
[H-01]	Recipient Bytes Silent Burn When Non-20-Byte Payloads Pass Validation	High	Fixed
[M-01]	Destination Slippage Protections Broken by Amount Drift	Medium	Fixed
[M-02]	Denial of Service via Large Payload Storage Exhaustion	Medium	Fixed
[M-03]	Fee Calculation Regression Enables Arbitrage on Price Drifts	Medium	Acknowledged
[M-04]	Retry Payload Channel Collision	Medium	Acknowledged
[M-05]	Retry Decoder Ignores Payload App Version Check	Medium	Fixed

[L-01]	Incorrect ERC-7201 Storage Slot in `BridgeStore` Contract	Low	Fixed
[L-02]	Early Liquidity Check Issue in `TransferPoolFeeCalculatorV2`	Low	Fixed
[L-03]	Missing Pre-validation Causes Panic on Invalid Swap Parameters	Low	Fixed
[L-04]	Missing Destination Swap Token Validation	Low	Fixed
[L-05]	Lack of Fallback Oracle	Low	Acknowledged
[L-06]	Owner Can Effectively Set Fees to 100% for a Pool	Low	Acknowledged
[I-01]	Redundant Deadline Check in `_swapAndTransfer()`	Info	Fixed
[I-02]	Native Token Trapping in `executeFullSwapNative()`	Info	Fixed

7. Findings

[H-01] Recipient Bytes Silent Burn When Non-20-Byte Payloads Pass Validation

Severity

High Risk

Description

All user-facing bridge entrypoints (`transferPool()` , `transferToken()` , `withdrawRemote()` , etc.) invoke `_validateToLength(to)` before constructing the packet, but this helper merely enforces that the payload is non-empty and at most 1 KB. It never ensures the bytes represent a 20-byte EVM address, nor does it attempt to decode them on the source chain. Once the packet reaches the destination, `onRecvPacket()` immediately calls `IBCUtils.decodeAddress(p.to)` . The assembly decoder succeeds only if the calldata word equals 20, otherwise, it returns `(address(0), false)` . Whenever decoding fails, the bridge emits `Unrecoverable` and executes `_handlePoolRecvFailure()` (which just unlocks accounting state) without persisting any `revertReceive()` payload. Because the source pool already burned or escrowed the user's tokens, the transfer becomes irrecoverable due solely to a malformed `to` length.

Location of Affected Code

File: src/bridge/v1.1/BridgeV1_1.sol#L200-L204

```
function transferPool(..., bytes calldata to, ...) external payable
    nonReentrant {
     // code
     _validateToLength(to);
     _validatePayloadLength(externalInfo.payload);
     _validateDstOuterGas(externalInfo.dstOuterGas);
     uint256 dstChainId = getChainId(srcChannel, true);
     // code
}
```

File: src/bridge/v1.1/BridgeV1_1.sol#L891-L898

```
function _validateToLength(bytes calldata to) internal pure {
    if (to.length == 0) {
        revert TokiInvalidRecipientBytes();
    }
    if (to.length > MAX_TO_LENGTH) {
        revert TokiExceed("to", to.length, MAX_TO_LENGTH);
    }
}
```

File: src/bridge/v1.1/BridgeV1_1.sol#L415-L464

```
function onRecvPacket( Packet calldata packet, address /* relayer */ )
   external override onlyRole(IBC_HANDLER_ROLE) nonReentrant returns (
   bytes memory acknowledgement) {
    uint8 messageType = IBCUtils.parseType(packet.data);
    if (messageType == MessageType._TYPE_TRANSFER_POOL) {
        IBCUtils.TransferPoolPayload memory p = IBCUtils.
           decodeTransferPool(packet.data);
        _updateCredit(...);
        (address recipient, bool success) = IBCUtils.decodeAddress(p.to);
        if (success) {
            _receivePool(... recipient ...);
        } else {
            _handlePoolRecvFailure(packet.destinationChannel, p.srcPoolId
               , p.dstPoolId, p.feeInfo);
            emit Unrecoverable(getChainId(packet.destinationChannel,
               false), packet.sequence);
        }
    }
    // code
}
```

File: src/library/IBCUtils.sol#L731-L739

```
function decodeAddress(bytes memory data) internal pure returns (address
   addr, bool success) {
    assembly {
        success := eq(mload(data), 20)
        addr := mload(add(data, 20))
    }
}
```

Impact

Any `transferPool()`, `transferToken()`, or `withdrawRemote()` call whose `to` parameter has a length other than 20 passes `_validateToLength()`, burns or escrows the tokens on the source chain, and sends an IBC packet. The destination bridge then fails to decode the bytes and chooses the `Unrecoverable` branch, which (a) never writes a retry payload, (b) leaves the funds debited,

and (c) provides the user no actionable remediation path besides pleading with pool administrators. Simple user mistakes, therefore, escalate to permanent loss of the bridged amount.

Proof of Concept

1. Call `transferPool(...)` with `to = hex"yash@123456789"` (4 bytes). `_validateToLength()` allows it because the payload is non-empty and below 1 KB, so the bridge burns LP tokens and emits the packet.
2. On the destination chain, `IBCUtils.decodeAddress(p.to)` finds `mload(data) = 4`, so `success = 0`.
3. The branch in `onRecvPacket()` falls through to `_handlePoolRecvFailure()` (which just restores internal accounting) and emits `Unrecoverable(...)` without persisting any data to `revertReceive()`.
4. The user attempts `retryOnReceive(...)`, but there is no payload to consume; the transfer remains stuck indefinitely while the source liquidity has already been reduced.

Recommendation

Enforce `to.length == 20` (or decode to `address` in the source contract) for all entrypoints whose destination logic expects an EVM address. Alternatively, provide a dedicated path for arbitrary recipient payloads so that non-address transfers can be retried safely instead of being dropped.

Team Response

Fixed.

[M-01] Destination Slippage Protections Broken by Amount Drift

Severity

Medium Risk

Description

The `DexReceiverUniswap._recalculateAmountIn()` tries to reconcile the user-provided swap parameters with the actual token amount delivered by the bridge. When `actualAmount > totalAmountIn`, it **only** mutates the last hop by adding the entire surplus to that hop's `amountIn` while leaving every other field (including `amountOutMin`, path, and fee tier) untouched. `_swapAndTransfer()` then trusts the mutated parameters, approves `token` for the inflated `amountIn`, and forwards the full balance to the Universal Router, allowing counterparties to satisfy a stale minimum price while capturing all surplus value. Multi-hop routes magnify the issue because the surplus is forced into the final hop even when the sender purposely splits liquidity to reduce price impact. This path is mainly reached when the v1 version of `TransferPoolFeeCalculator.sol` is used for fee calculation over the pool.

The opposite branch is also unsafe. When `actualAmount < totalAmountIn` (due to bridge fees, rounding, or upstream slippage), `_recalculateAmountIn()` truncates later hops but **copies the**

original `amountOutMin` **values** into the reduced hops. The destination swap must now achieve the same minimum output with fewer input tokens, so honest liquidity reverts, leaving funds stuck in `DexReceiverUniswap`.

Location of Affected Code

File: src/dex/DexReceiverUniswap.sol#L519-L531

```
function _recalculateAmountIn(uint256 actualAmount, IDexUniswap.SwapParam
    [] memory swapParams) private pure returns (IDexUniswap.SwapParam[]
    memory) {
     uint256 totalAmountIn = swapParams.getAmountInSum();
     if (actualAmount > totalAmountIn) {
          uint256 excessAmount = actualAmount - totalAmountIn;
          swapParams[swapParams.length - 1].amountIn += excessAmount;
          return swapParams;
     }
     // code
}
```

File: src/dex/DexReceiverUniswap.sol#L262-L330

```
function _swapAndTransfer(uint256 amountIn, address token, IDexUniswap.
    DstSwapInfo memory dstSwapInfo) private returns (bool) {
     dstSwapInfo.swapParams = _recalculateAmountIn(amountIn, dstSwapInfo.
        swapParams);
     // code
     IUniversalRouter(UNIVERSAL_ROUTER).execute(commands, inputs,
        dstSwapInfo.deadline);
}
```

File: src/dex/DexReceiverUniswap.sol#L551-L557

```
function _recalculateAmountIn( uint256 actualAmount, IDexUniswap.
    SwapParam[] memory swapParams ) private pure returns (IDexUniswap.
    SwapParam[] memory) {
     // code
     validSwaps[i] = IDexUniswap.SwapParam({
          command: swapParams[i].command,
          action: swapParams[i].action,
          amountIn: remainingAmount,
          amountOutMin: swapParams[i].amountOutMin,
          path: swapParams[i].path
     });
     // code
}
```

Surplus creation and delivery shortfalls both originate from the bridge pools:

File: src/Pool.sol#L468

```
amountLD = _GDToLD(feeInfo.amountGD + feeInfo.eqReward);
```

Impact

- **Stale minimums on surplus**: Whenever the bridge delivers more tokens than expected (rewards, dust handling, deliberate overfunding), the final hop enforces the pre-surplus `amountOutMin` . LPs, MEV searchers, or malicious routers can clear the trade at that stale bound and keep the entire surplus value, defeating users' destination slippage protections.
- **Impossible minimums on deficits**: Whenever the bridge delivers less than `totalAmountIn` , the copied `amountOutMin` becomes unachievable for the reduced input. Swaps revert or remain stuck in retry queues.

Recommendation

Please rethink the approach to handling the surplus and deficit amounts.

Team Response

Fixed.

[M-02] Denial of Service via Large Payload Storage Exhaustion

Severity

Medium Risk

Description

The bridge allows users to send cross-chain transfers with payloads up to 10KB in size. When a receiver contract fails to process this payload, the bridge tries to store the entire 10KB payload on-chain for future retry attempts. However, storing 10KB requires approximately 6.4 million gas, which can exceed the remaining gas in the transaction and cause it to revert.

Since the bridge uses an ordered IBC channel, all packets must be processed in sequence. If one packet fails and reverts the entire transaction, the channel becomes blocked and all subsequent transfers are stuck until the issue is resolved.

An attacker can exploit this by crafting a malicious transfer that:

1. Contains the maximum 10KB payload
2. Requests the maximum 5M gas for the receiver call
3. Uses a receiver contract that fails after consuming most of the 5M gas

When the relayer processes this packet with a typical gas limit of 6-7M, the receiver consumes 5M gas, then the storage operation needs 6.4M more gas, but does not have enough remaining. The transaction reverts, blocking the entire channel.

The team previously stated that this vector was addressed by tweaking the `RelayerFeeCalculator` , but the current v1.1 deployment still writes the entire payload into storage and nothing enforces a higher relayer gas limit. The calculator only scales the fee the user pays, it does not restrict `externalInfo.payload` , force a minimum relay gas limit, or reserve gas so the retry record can be persisted.

```
function calcFee( uint256 peerChainId, uint256 peerPoolId, address from,
    uint256 amountLD ) external view returns (ITransferPoolFeeCalculator.
    FeeInfo memory feeInfo) {
     // code
     uint256 gas = gasUsed;
     if (
         messageType == MessageType._TYPE_TRANSFER_POOL ||
         messageType == MessageType._TYPE_TRANSFER_TOKEN
     ) {
         gas +=
             gasPerPayloadLength *
             externalInfo.payload.length +
             externalInfo.dstOuterGas;
     }
     relayerFee.fee =
         (gas *
             relayerFee.dstGasPrice *
             premiumBPS *
             relayerFee.dstTokenPrice *
             decimalRatioNumerator) /
         (10000 * relayerFee.srcTokenPrice * decimalRatioDenominator);
     // code
}
```

As soon as the external callback burns most of its allotted 5M gas, the surrounding transaction runs out of gas before the catch block can enqueue a retry, meaning the ordered channel still deadlocks exactly as in the original finding.

Location of Affected Code

File: src/bridge/v1.1/BridgeBaseV1_1.sol#L317-L328

```
function _outServiceCallCore(
    string memory dstChannel,
    uint256 srcChainId,
    uint64 sequence,
    address token,
    uint256 amount,
    address to,
    uint256 refuelAmount,
    IBCUtils.ExternalInfo memory externalInfo,
    bool doRefuel,
    uint256 lastValidHeightOrZero
) internal {
    // code

    // External call with user-specified gas limit
    if (externalInfo.payload.length > 0 && to.code.length > 0) {
        try
            ITokiOuterServiceReceiverV1_1(to).onReceivePool{
                gas: externalInfo.dstOuterGas  // Up to 5M gas
            }(dstChannel, sequence, token, amount, externalInfo.payload)
        returns (bool successOuter) {
            if (!successOuter) {
                errorFlag += 2;
            }
        } catch {
            errorFlag += 2;
        }
    }

    // If external call failed, store entire payload for retry
    if (errorFlag == 2) {
        // @audit Stores full 10KB payload requiring ~6.4M gas
        _asReceiveRetryableSelf().addRevertExternal(
            srcChainId,
            sequence,
            token,
            amount,
            to,
            externalInfo,  // Contains up to 10KB payload
            lastValidHeightOrZero
        );
    }
}
```

File: src/bridge/v1.1/BridgeFallbackV1_1.sol#L254-L286

```
function addRevertExternal(
    uint256 srcChainId,
    uint64 sequence,
    address token,
    uint256 amount,
    address to,
    IBCUtils.ExternalInfo memory externalInfo,
    uint256 lastValidHeight
) external onlySelf {
    BridgeStorage storage $ = getBridgeStorage();

    // @audit Storing 10KB costs ~6.4M gas (320 slots × 20k gas/slot)
    $.revertReceive[srcChainId][sequence] = IBCUtils
        .encodeRetryExternalCall(
            APP_VERSION,
            lastValidHeight,
            token,
            amount,
            to,
            externalInfo  // Full 10KB payload stored in contract storage
        );

    emit RevertExternal(srcChainId, sequence);
}
```

File: src/bridge/v1/BridgeStore.sol#L59-L62

```
uint256 internal constant MAX_PAYLOAD_LENGTH = 10 * 1024; // 10KB
uint256 internal constant MAX_OUTER_GAS = 5_000_000; // 5M
```

File: src/bridge/v1.1/BridgeV1_1.sol#L566-L587

```
function onChanOpenInit(IIBCModuleInitializer.MsgOnChanOpenInit calldata
   msg_) external view override onlyRole(IBC_HANDLER_ROLE) returns (
   address, string memory version) {
    if (msg_.order != Channel.Order.ORDER_ORDERED) {
        revert TokiRequireOrderedChannel();
    }
    // @audit Ordered channel means packets must be processed
       sequentially
}
```

Impact

An attacker can completely block all cross-chain activity on a specific bridge channel by sending a single malicious transaction. This prevents:

- All legitimate user transfers
- Token withdrawals
- DEX swap operations
- Credit updates

Attack Execution Steps:

1. **Attacker prepares malicious transfer** on Source Chain:
   - Creates a 10KB payload filled with arbitrary data
   - Sets `dstOuterGas` to 5,000,000 (maximum allowed)
   - Targets any receiver contract
2. **Attacker calls** `transferPool(...)` paying minimal gas fees (cheap on L2s)
3. **Relayer picks up packet** and submits it to Destination Chain with ~6-7M gas limit
4. **Bridge processes packet** in `onRecvPacket(...)`:
   - Calls `_outServiceCallCore(...)`
   - Executes receiver with 5M gas limit
   - Receiver fails after consuming ~4-5M gas
5. **Bridge attempts to store failure**:
   - Calls `addRevertExternal(...)`
   - Needs to write 10KB to storage
   - Requires ~6.4M gas
   - Only ~1-2M gas remains
6. **Transaction reverts with Out of Gas**
7. **IBC ordered channel is blocked**:
   - Packet never acknowledged
   - All subsequent packets are stuck in the queue
   - Users cannot bridge tokens
   - Channel unusable until manual intervention

Recommendation

- Reduce `MAX_PAYLOAD_LENGTH` (or persist only a hash/URI) so the bridge can always store failures even when `dstOuterGas` is maxed.
- Reject packets whose requested gas + payload size would exceed the relayer's guaranteed gas limit instead of merely charging a higher fee; do not rely on `RelayerFeeCalculator` pricing to enforce safety.
- Validate/allowlist destination callbacks or introduce per-chain limits so that arbitrary `to` contracts cannot intentionally waste gas before the retry state is written.

Team Response

Fixed.

[M-03] Fee Calculation Regression Enables Arbitrage on Price Drifts

Severity

Medium Risk

Description

`TransferPoolFeeCalculatorV2` now charges only static protocol/LP fees plus a binary `Depeg` check. When both pools are marked `PriceDeviationStatus.Drift`, the calculator ignores any real price delta between them, so an attacker can bridge out of the cheaper pool into the more expensive one, pay only the tiny LP fee, and pocket the spread while draining the destination pool's reserves.

Location of Affected Code

File: src/replaceable/TransferPoolFeeCalculatorV2.sol#L137-L174

```
function calcFee( SrcPoolInfo calldata srcPoolInfo, IPool.PeerPoolInfo
    calldata dstPoolInfo, address from, uint256 amountGD ) external view
    notDepeg(srcPoolInfo.id, dstPoolInfo.id) returns (FeeInfo memory
    feeInfo) {
     // code
     feeInfo.protocolFee = getProtocolFee(whitelisted, amountGD);
     feeInfo.lpFee = getLpFee(whitelisted, amountGD);
     uint256 minTransactionFee = getMinimumTransactionFee(
         whitelisted,
         srcPoolInfo.id
     );
     uint256 transactionFee = feeInfo.protocolFee + feeInfo.lpFee;
     if (transactionFee < minTransactionFee) {
         uint256 adjustment = minTransactionFee - transactionFee;
         feeInfo.protocolFee += adjustment;
     }
     if (amountGD <= feeInfo.protocolFee + feeInfo.lpFee) {
         feeInfo.amountGD = 0;
     } else {
         feeInfo.amountGD = amountGD - feeInfo.protocolFee - feeInfo.lpFee
             ;
     }
}
```

Impact

Attacker repeatedly bridges from a discounted pool to a premium pool, withdrawing full-value tokens while paying almost no compensation. Destination pool liquidity collapses, causing honest withdrawals to revert with `TokiInsufficientPoolLiquidity`, and the attacker realizes the inter-chain price spread as profit.

Proof of Concept

1. Pool A token trades at 0.97 USD; Pool B trades at 1.00 USD, the oracle still reports `PriceDeviationStatus.Normal`.
2. Attacker deposits 10 M tokens into Pool A. With 7 bps LP fee, `feeInfo.amountGD  9.993 M`.
3. Bridge mints ~9.993 M tokens on Pool B, which the attacker sells for ~$9.993 M, after paying only ~$9.7 M on Pool A.
4. Pool B loses ~9.993 M liquidity per iteration, so repeating the trade drains the pool until withdrawals fail.

Recommendation

Reintroduce a drift-aware fee equal to the expected arbitrage value, e.g., `(dstPrice - srcPrice) / dstPrice * amountGD` whenever `srcPrice < dstPrice`, or revert transfers once the relative deviation exceeds a configurable bps threshold. Add regression tests ensuring bridging across unequal prices is never profitable.

Team Response

Acknowledged.

[M-04] Retry Payload Channel Collision

Severity

Medium Risk

Description

Packet sequences are scoped per IBC channel, but the retry ledger collapses all channels that share a `counterpartyChainId` into the same storage slot. `BridgeStore` exposes a two-level map keyed exclusively by the destination chain and sequence (`revertReceive[chainId][sequence]`). Every failure handler in `BridgeFallbackV1_1` stores its payload into that map using only the chain ID resolved from the channel (`getChainId(dstChannel, true)`), so two different channels (or a re-opened channel whose sequence restarted) inevitably reuse the same bucket once they emit identical sequence numbers. When `_retryOnReceiveInternal()` later looks up the payload, it uses the same `[chainId][sequence]` key and finds only the most recent failure, making the older user permanently unable to replay their packet even though the source pool already debited their funds.

Location of Affected Code

File: src/bridge/v1/BridgeStore.sol#L34

```
mapping(uint256 => mapping(uint64 => bytes)) revertReceive; // [chainId][
    sequence] = payload
```

File: src/bridge/v1.1/BridgeBaseV1_1.sol#L44-L60

```
function getChainId(string memory localChannel, bool checksAppVersion)
    public view returns (uint256 counterpartyChainId) {
     ChannelInfo memory channelInfo = $.channelInfos[localChannel];
     counterpartyChainId = channelInfo.counterpartyChainId;
     if (checksAppVersion && channelInfo.appVersion != APP_VERSION) {
         revert TokiInvalidAppVersion(channelInfo.appVersion, APP_VERSION)
             ;
     }
}
```

File: src/bridge/v1.1/BridgeFallbackV1_1.sol#L114-L149

```
function addRevertReceivePool(
    uint256 srcChainId,
    uint64 sequence,
    // code
) external onlySelf {
    // code
    $.revertReceive[srcChainId][sequence] = IBCUtils.
       encodeRetryReceivePool(
        APP_VERSION,
        lastValidHeight,
        srcPoolId,
        dstPoolId,
        to,
        fee,
        refuelAmount,
        externalInfo
    );
    emit RevertReceive(IBCUtils._TYPE_RETRY_RECEIVE_POOL, srcChainId,
       sequence, lastValidHeight);
}
```

File: src/bridge/v1.1/BridgeFallbackV1_1.sol#L331-L477

```
function _retryOnReceiveInternal(string calldata dstChannel, uint64
   sequence, uint256 newDstOuterGas) internal {
    uint256 srcChainId = getChainId(dstChannel, true);
    bytes memory payload = $.revertReceive[srcChainId][sequence];
    if (payload.length <= 0) revert TokiNoRevertReceive();
    $.revertReceive[srcChainId][sequence] = "";
    if (IBCUtils.parseType(payload) == IBCUtils._TYPE_RETRY_RECEIVE_POOL)
        {
        IBCUtils.RetryReceivePoolPayload memory p = IBCUtils.
           decodeRetryReceivePool(payload);
        _receivePool(dstChannel, sequence, p.srcPoolId, p.dstPoolId, p.to
           , p.feeInfo, ...);
    }
    // code
}
```

Impact

Whenever any channel toward the same chain produces a failing packet with an already-used sequence, its `addRevert*` call overwrites the previous bytes in `revertReceive[chainId][sequence]`. The original packet's data disappears before `_retryOnReceiveInternal()` can consume it, so the bridge can never reconstruct or replay that transfer despite having removed the user's liquidity on the source chain. Because ordered channels must reuse increasing sequences starting from 1 whenever they are created or reset, the overwrite is guaranteed to happen under concurrent channels or after upgrades.

Proof of Concept

1. Configure `channel-0` and `channel-1` so both resolve to the same `counterpartyChainId` through `getChainId()`.
2. Cause packet `sequence = 5` on `channel-0` to fail, making `addRevertReceivePool()` store data at `revertReceive[chainId][5]`.
3. Drive `channel-1` until its ordered sequence also reaches 5 and force another failure. Because it reuses the same `chainId` (derived from the channel mapping) and identical `sequence`, the new payload overwrites the previous bytes.
4. When the `channel-0` user calls `retryOnReceive(channel-0, 5)`, `_retryOnReceiveInternal()` reads the overwritten payload, which either replays the wrong transfer or reverts due to a mismatched pool, leaving the legitimate retry impossible (the slot was already cleared).

Recommendation

Key retries by a unique channel identifier (e.g., `mapping(bytes32 => mapping(uint64 => bytes))` keyed by `keccak256(localChannel)` or port+channel) so different channels cannot collide. Existing entries should be migrated or replayed before deploying the change to avoid further overwrites.

Team Response

Acknowledged.

[M-05] Retry Decoder Ignores Payload App Version Check

Severity

Medium Risk

Description

Retry payloads capture the exact `appVersion` that produced them so operators can safely upgrade channel/bridge logic while packets are in flight. `IBCUtils.encodeRetry*` always places the version right after the type discriminator, giving the destination enough context to choose the right decoder. However, `_retryOnReceiveInternal()` ignores that field entirely: it parses the payload, immediately ABI-decodes it into the current struct definition, and forwards the values to `_receivePool()`, `_receiveToken()`, `_receiveWithdrawConfirm()`, etc. If the struct layout or semantics changed between the version that created the payload and the version that is currently running (for example, adding a new field or reordering existing ones), the decode step treats legacy bytes as the new layout. The resulting garbage values corrupt pool identifiers, recipients, amounts, or gas settings, causing retries to revert forever or to execute against the wrong pools. Because the payload is deleted before processing, the user loses the only copy of their retry data.

Location of Affected Code

File: src/library/IBCUtils.sol#L394-L416

```
function encodeRetryReceivePool(
    uint256 appVersion,
    uint256 lastValidHeight,
    uint256 srcPoolId,
    uint256 dstPoolId,
    address to,
    ITransferPoolFeeCalculator.FeeInfo memory feeInfo,
    uint256 refuelAmount,
    IBCUtils.ExternalInfo memory externalInfo
) internal pure returns (bytes memory) {
    return abi.encode(
        _TYPE_RETRY_RECEIVE_POOL,
        appVersion,
        lastValidHeight,
        srcPoolId,
        dstPoolId,
        to,
        feeInfo,
        refuelAmount,
        externalInfo
    );
}
```

File: src/bridge/v1.1/BridgeFallbackV1_1.sol#L331-L477

```
function _retryOnReceiveInternal(string calldata dstChannel, uint64
   sequence, uint256 newDstOuterGas) internal {
    uint256 srcChainId = getChainId(dstChannel, true);
    bytes memory payload = $.revertReceive[srcChainId][sequence];
    if (payload.length <= 0) revert TokiNoRevertReceive();
    $.revertReceive[srcChainId][sequence] = "";
    uint8 rtype = IBCUtils.parseType(payload);
    if (rtype == IBCUtils._TYPE_RETRY_RECEIVE_POOL) {
        IBCUtils.RetryReceivePoolPayload memory p = IBCUtils.
           decodeRetryReceivePool(payload);
        _validateRetryExpiration(p.lastValidHeight);
        ReceiveOptionV1_1 memory receiveOption = ReceiveOptionV1_1(false,
            p.lastValidHeight);
        _receivePool(dstChannel, sequence, p.srcPoolId, p.dstPoolId, p.to
           , p.feeInfo, p.refuelAmount, p.externalInfo, receiveOption);
    } else if (rtype == IBCUtils._TYPE_RETRY_RECEIVE_TOKEN) {
        IBCUtils.RetryReceiveTokenPayload memory p = IBCUtils.
           decodeRetryReceiveToken(payload);
        // code
    }
    // code
}
```

File: src/bridge/v1.1/BridgeBaseV1_1.sol#L40-L60

```
function appVersion() external view returns (uint256) {
    return APP_VERSION;
}

function getChainId(string memory localChannel, bool checksAppVersion)
   public view returns (uint256 counterpartyChainId) {
    ChannelInfo memory channelInfo = $.channelInfos[localChannel];
    counterpartyChainId = channelInfo.counterpartyChainId;
    if (checksAppVersion && channelInfo.appVersion != APP_VERSION) {
        revert TokiInvalidAppVersion(channelInfo.appVersion, APP_VERSION)
            ;
    }
}
```

Impact

Whenever the bridge upgrades to a new implementation (or to a new channel version), any outstanding `revertReceive()` payloads continue to hold the bytes produced by the old version. Because `_retryOnReceiveInternal()` never inspects `p.appVersion`, it blindly decodes the bytes as if they were produced by the new ABI. Even a harmless-looking change—such as inserting a single field before `dstPoolId` or adjusting the layout of nested structs like `FeeInfo`—causes all subsequent fields to shift. The corrupted values then make `_receivePool()` look up the wrong pools, underflow fee math, or throw when dereferencing nonexistent IDs. Since the payload is deleted before the handler executes, a single revert bricks the retry permanently and leaves the user's funds stranded.

Proof of Concept

1. v1.0 struct layout: `type, appVersion, lastValidHeight, srcPoolId dstPoolId, address to, feeInfo, refuelAmount, externalInfo`.
2. v1.1 adds a `uint256 extraNonce` before `dstPoolId`. All encoded retries that were already stored on-chain keep the old 8-field layout.
3. After upgrading bridge/fallback contracts, `_retryOnReceiveInternal()` decodes an old payload using the new layout. `p.dstPoolId` now equals the former `to` bytes, so `_receivePool()` looks up a pool at an arbitrary 20-byte value and reverts.
4. The payload was erased at the beginning of `_retryOnReceiveInternal()`, so the retry cannot be re-queued or reinterpreted by legacy logic, the user's transfer remains permanently stuck.

Recommendation

Compare `p.appVersion` to `APP_VERSION` before processing any retry. Reject unsupported versions with an explicit error, or dispatch to a version-specific handler (e.g., delegatecall to a legacy fallback) that decodes and executes the old layout correctly. This ensures upgrades cannot silently corrupt pending retries.

Team Response

Fixed.

[L-01] Incorrect ERC-7201 Storage Slot in `BridgeStore` Contract

Severity

Low Risk

Description

The `BridgeStore` contract uses an incorrect ERC-7201 storage slot location that does not match the declared namespace. The contract declares the storage location as `erc7201:toki.bridge`, but the hardcoded storage slot value `0x183a6125c38840424c4a85fa12bab2ab606c4b6d0e7cc73c0c06ba5300eab500` does not correspond to this namespace.

According to ERC-7201, the storage slot should be calculated as:

```
keccak256(abi.encode(uint256(keccak256("toki.bridge")) - 1)) & ~bytes32(
    uint256(0xff))
```

When calculated correctly, this yields:

```
0xb04bb14e246c8ed12b0fb1503efcdd29f5bf10851fa1eb22ec0591e573888e00
```

However, the contract uses `0x183a6125c38840424c4a85fa12bab2ab606c4b6d0e7cc73c0c06ba5300eab500`, which appears to be the storage slot for a different namespace (possibly "example.main" based on the Chisel output provided).

This discrepancy violates the ERC-7201 standard and creates a mismatch between the declared namespace and the actual storage location used by the contract.

Location of Affected Code

File: src/bridge/v1/BridgeStore.sol#L52-L54

```
// keccak256(abi.encode(uint256(keccak256("toki.bridge")) - 1)) & ~
    bytes32(uint256(0xff));
bytes32 internal constant BRIDGE_STORAGE_SLOT =
    0x183a6125c38840424c4a85fa12bab2ab606c4b6d0e7cc73c0c06ba5300eab500;
```

Impact

The incorrect storage slot location poses several risks:

1. **Standards Non-Compliance**: The contract does not comply with ERC-7201, which may cause issues with tooling, analyzers, and other contracts that expect standard-compliant storage layouts.

Proof of Concept

The Chisel output demonstrates the correct calculation:

```
> chisel
 keccak256(abi.encode(uint256(keccak256("toki.bridge")) - 1)) & ~bytes32(
    uint256(0xff))
Type: uint256
 Hex: 0xb04bb14e246c8ed12b0fb1503efcdd29f5bf10851fa1eb22ec0591e573888e00
 Hex (full word): 0
    xb04bb14e246c8ed12b0fb1503efcdd29f5bf10851fa1eb22ec0591e573888e00
 Dec: 797407985963283857672522181860693066350597835222048689 51...
 keccak256(abi.encode(uint256(keccak256("example.main")) - 1)) & ~bytes32
    (uint256(0xff))
Type: uint256
 Hex: 0x183a6125c38840424c4a85fa12bab2ab606c4b6d0e7cc73c0c06ba5300eab500
 Hex (full word): 0
    x183a6125c38840424c4a85fa12bab2ab606c4b6d0e7cc73c0c06ba5300eab500
 Dec: 109586559832611522718484366922911372754430242756535229 9...
```

The correct storage slot should be `0xb04bb14e246c8ed12b0fb1503efcdd29f5bf10851fa1eb22ec0591e573888e00`, but the contract uses `0x183a6125c38840424c4a85fa12bab2ab606c4b6d0e7cc73c0c06ba5300eab500`.

Recommendation

Update the `BRIDGE_STORAGE_SLOT` constant to use the correct ERC-7201 calculated value:

```
// keccak256(abi.encode(uint256(keccak256("toki.bridge")) - 1)) & ~
   bytes32(uint256(0xff))
bytes32 internal constant BRIDGE_STORAGE_SLOT =
    0xb04bb14e246c8ed12b0fb1503efcdd29f5bf10851fa1eb22ec0591e573888e00;
```

Team Response

Fixed.

[L-02] Early Liquidity Check Issue in `TransferPoolFeeCalculatorV2`

Severity

Low Risk

Description

The `calcFee(...)` function in `TransferPoolFeeCalculatorV2.sol` is responsible for calculating fees for cross-chain transfers and validating that the destination pool has sufficient liquidity to cover the transfer. The function performs a liquidity check at the beginning using the pre-fee transfer amount (`amountGD`).

However, the actual amount deducted from the destination pool's balance is `amountGD - feeInfo.lpFee` (since `eqReward` is zero in V2). The early liquidity check compares `dstPoolInfo.balance < amountGD` before fees are calculated, but the actual balance decrease that occurs in `Pool.transfer()` is

`feeInfo.balanceDecrease = amountGD - feeInfo.lpFee + feeInfo.eqReward` . Since fees reduce the amount that needs to be covered, the early check is overly conservative and can prevent valid transfers from executing.

This means that transfers may be incorrectly rejected even when the pool has sufficient liquidity to cover the actual balance decrease after fees are deducted.

Location of Affected Code

File: src/replaceable/TransferPoolFeeCalculatorV2.sol#L137-L174

```
function calcFee(...) external view returns (FeeInfo memory feeInfo) {
    // @audit Early liquidity check using pre-fee amount
    if (dstPoolInfo.balance < amountGD) {
        revert TokiInsufficientPoolLiquidity(dstPoolInfo.balance,
           amountGD);
    }

    // ... fee calculation logic ...
    feeInfo.lpFee = getLpFee(whitelisted, amountGD);
}
```

File: src/Pool.sol#L404-L412

```
// Actual balance decrease calculation and check
feeInfo.balanceDecrease = amountGD - feeInfo.lpFee + feeInfo.eqReward;
if (peerPoolInfo.balance < feeInfo.balanceDecrease) {
    revert TokiInsufficientPoolLiquidity(
        peerPoolInfo.balance,
        feeInfo.balanceDecrease
    );
}
```

Impact

1. **Reduced Liquidity Utilization**: The pool may have sufficient liquidity to cover the actual transfer amount after fees, but transfers are rejected due to the early check using the pre-fee amount.
2. **User Experience Degradation**: Legitimate transfers that should succeed are incorrectly rejected, causing frustration and potential loss of trust in the protocol.

Proof of Concept

Consider the following scenario:

1. User attempts to transfer 1000 GD tokens
2. Destination pool has a balance of 998 GD
3. LP fee is calculated as 3 GD (0.3% of 1000 GD)
4. The early check at compares: `998 < 1000` **reverts**
5. However, the actual balance decrease would be: `1000 - 3 = 997 GD`
6. The pool has sufficient liquidity (998 >= 997), but the transfer is incorrectly rejected

The discrepancy occurs because:

- Early check uses: `amountGD` (1000 GD)
- Actual deduction uses: `amountGD - lpFee` (997 GD)
- Pool balance (998 GD) is sufficient for the actual deduction but insufficient for the pre-fee amount

Recommendation

Move the liquidity check to after the fee calculation and compare against the actual balance decrease amount. The check should be performed using `feeInfo.balanceDecrease` or `amountGD - feeInfo.lpFee` instead of the raw `amountGD`.

```
function calcFee(...) external view returns (FeeInfo memory feeInfo) {
    // ... fee calculation logic ...
    feeInfo.lpFee = getLpFee(whitelisted, amountGD);
    // code

    // Move liquidity check to after fee calculation
    // Check against actual balance decrease: amountGD - lpFee (eqReward
       is 0 in V2)
    uint256 balanceDecrease = amountGD - feeInfo.lpFee;
    if (dstPoolInfo.balance < balanceDecrease) {
        revert TokiInsufficientPoolLiquidity(
            dstPoolInfo.balance,
            balanceDecrease
        );
    }
}
```

Team Response

Fixed.

[L-03] Missing Pre-validation Causes Panic on Invalid Swap Parameters

Severity

Low Risk

Description

The `DexBridgeRouteUniswap._validSrcSwapInfoAndTransferInfo()` derives the `bridgeableToken` by calling `srcSwapInfo.getBridgeableToken()` before verifying that the swap info itself is valid. `getBridgeableToken()` assumes `swapParams.length > 0` and each path has at least one hop, so when a caller provides an empty `swapParams` array (or an empty path), the validation function reverts with a low-level panic instead of returning the intended `InvalidSrcSwapInfo` error.

This ordering bug prevents the router from surfacing the canonical swap validation errors and instead bricks execution with an opaque revert, making it hard for front ends or relayers to distinguish bad swap inputs from other failures.

Location of Affected Code

File: src/dex/DexBridgeRouteUniswap.sol#L429-L447

```
function _validSrcSwapInfoAndTransferInfo(IDexUniswap.SrcSwapInfo memory
    srcSwapInfo, TransferInfo memory transferInfo) private view returns (
    bool, ValidateRouterResult) {
     address bridgeableToken = srcSwapInfo.getBridgeableToken(); // <
        validation
     if (!_isValidatePoolToken(transferInfo.srcPoolId, bridgeableToken)) {
         return (false, ValidateRouterResult.InvalidPoolToken);
     }

     if (!srcSwapInfo.validate()) {
         return (false, ValidateRouterResult.InvalidSrcSwapInfo);
     }
     // code
}
```

File: src/dex/UniswapLibraries.sol#L61-L68

```
function getBridgeableToken(
    IDexBridgeRouteUniswap.SrcSwapInfo memory srcSwapInfo
) internal pure returns (address) {
    uint256 lastPathIndex = srcSwapInfo.swapParams[0].path.length - 1;
    return srcSwapInfo.swapParams[0].path[lastPathIndex].tokenOut;
}
```

Impact

Any call that accidentally (or maliciously) provides an empty `swapParams` array causes the router to revert with an unhandled panic instead of the expected `TokiInvalidSwapParams`, reducing debuggability for integrators.

Recommendation

Call `srcSwapInfo.validate()` before invoking `getBridgeableToken()`, so malformed payloads return `InvalidSrcSwapInfo` instead of panicking:

```
function _validSrcSwapInfoAndTransferInfo(
    IDexUniswap.SrcSwapInfo memory srcSwapInfo,
    TransferInfo memory transferInfo
) private view returns (bool, ValidateRouterResult) {
    // Validate swap info first to ensure swapParams and paths are valid
    if (!srcSwapInfo.validate()) {
        return (false, ValidateRouterResult.InvalidSrcSwapInfo);
    }

    // Now safe to call getBridgeableToken since validation passed
    address bridgeableToken = srcSwapInfo.getBridgeableToken();
    if (!_isValidatePoolToken(transferInfo.srcPoolId, bridgeableToken)) {
        return (false, ValidateRouterResult.InvalidPoolToken);
    }

    if (!_isValidateParamsCount(srcSwapInfo.swapParams)) {
        return (false, ValidateRouterResult.InvalidCountParams);
    }
    return (true, ValidateRouterResult.Valid);
}
```

Team Response

Fixed.

[L-04] Missing Destination Swap Token Validation

Severity

Low Risk

Description

The `DexReceiverUniswap` contract lacks validation to ensure that the destination swap parameters start with the actual bridgeable token delivered by the bridge transfer. While this doesn't lead to fund loss, it does cause transaction failures and poor user experience when users provide mismatched swap paths.

The receiver validates swap parameter structure and deadlines, but never compares the first hop (`swapParams[0].path[0].tokenIn`) with the token passed into `onReceivePool()` . Because of this, mismatched swap configurations are only caught once the Universal Router execution reverts.

Location of Affected Code

File: src/dex/DexReceiverUniswap.sol#L241-L258

```
function isSwapExecutable(
    IDexUniswap.DstSwapInfo memory dstSwapInfo
) public view returns (bool, ValidateReceiverResult) {
    if (!dstSwapInfo.validate()) {
        return (false, ValidateReceiverResult.InvalidDstSwapInfo);
    }
    if (!_validateParamsCount(dstSwapInfo.swapParams)) {
        return (false, ValidateReceiverResult.InvalidCountParams);
    }
    if (dstSwapInfo.deadline <= block.timestamp) {
        return (false, ValidateReceiverResult.InvalidDeadline);
    }
    return (true, ValidateReceiverResult.Valid);
}

// Called right after isSwapExecutable passes
bool success = _swapAndTransfer(amount, token, dstSwapInfo);
if (!success) {
    emit SwapSkipped(
        token,
        dstToken,
        dstSwapInfo.toAddress,
        amount,
        sequence
    );
    _transferBridgeableToken(
        dstChannel,
        sequence,
        token,
        dstSwapInfo.toAddress,
        amount
    );
    return true;
}
```

Impact

This validation gap results in:

When destination swap paths don't start with the correct bridged token, the Universal Router execution fails, leading to the bridgeable token being transferred directly to users instead of being swapped.

Proof of Concept

Consider a cross-chain swap scenario where the validation gap causes transaction failure:

1. **Source Chain Setup**: User initiates a swap with `SrcSwapInfo` like `USDT -> DAI -> USDC`, where USDC is the bridgeable token that gets sent to the destination chain.
2. **Destination Chain Setup**: User provides a mismatched `DstSwapInfo` like `WETH -> DAI -> TARGET_TOKEN`, where the first token (WETH) doesn't match the actual bridged token (USDC).

3. **Validation Bypass**: `isSwapExecutable()` validates structure, deadlines, and path counts but never checks the `token` argument received from the bridge, so the mismatched path is flagged as valid.
4. **Execution Failure**: `_swapAndTransfer()` forwards `token` and `dstSwapInfo()` to the Universal Router. The router reverts because it tries to pull WETH (which the contract never received) instead of USDC.
5. **Fallback Behavior**: The `catch` block emits `SwapSkipped` and `_transferBridgeableToken()` returns the untouched USDC to the user, surprising them with an unexpected token instead of the desired output.

Recommendation

Add token validation in the `isSwapExecutable()` function to verify destination swap consistency:

```
function isSwapExecutable(
    IDexUniswap.DstSwapInfo memory dstSwapInfo,
    address bridgedToken // Add parameter for the actual bridged token
) public view returns (bool, ValidateReceiverResult) {
    if (!dstSwapInfo.validate()) {
        return (false, ValidateReceiverResult.InvalidDstSwapInfo);
    }
    if (!_validateParamsCount(dstSwapInfo.swapParams)) {
        return (false, ValidateReceiverResult.InvalidCountParams);
    }
    if (dstSwapInfo.deadline <= block.timestamp) {
        return (false, ValidateReceiverResult.InvalidDeadline);
    }

    // Add token consistency validation
    address expectedFirstToken = dstSwapInfo.swapParams[0].path[0].
       tokenIn;
    if (bridgedToken != expectedFirstToken) {
        return (false, ValidateReceiverResult.InvalidBridgedToken);
    }

    return (true, ValidateReceiverResult.Valid);
}
```

This ensures mismatched token configurations are caught early in the validation phase rather than during expensive Universal Router execution attempts.

Team Response

Fixed.

[L-05] Lack of Fallback Oracle

Severity

Low Risk

Description

In `TransferPoolFeeCalculatorV2.sol`, the contract relies on the predefined oracle to detect a depeg event. However, on oracle failure, the contract may wrongly assume it happened. There is no secondary source of truth to verify the result. This may cause unavailability of the contract if Oracle is unavailable.

Location of Affected Code

File: src/replaceable/TransferPoolFeeCalculatorV2.sol#L48-L57

```
modifier notDepeg(uint256 srcPoolId, uint256 dstPoolId) {
    if (srcPoolId != dstPoolId) {
        IStableTokenPriceOracle.PriceDeviationStatus stat =
           stableTokenPriceOracle
                .getCurrentPriceDeviationStatus(srcPoolId);
        if (stat == IStableTokenPriceOracle.PriceDeviationStatus.Depeg) {
            revert TokiDepeg(srcPoolId);
        }
    }
    _;
}
```

Recommendation

Implement a fallback oracle that can be queried when the primary one is stale/unavailable.

Team Response

Acknowledged.

[L-06] Owner Can Effectively Set Fees to 100% for a Pool

Severity

Low Risk

Description

In `TransferPoolFeeCalculatorV2.sol`, the contract lets the owner configure a perpool minimum fee in "global decimals" via `setMinimumFee(poolId, minimumFeeGD)`. In `calcFee()`, after computing the protocol + LP fee, it enforces:

```
uint256 minTransactionFee = getMinimumTransactionFee(whitelisted,
   srcPoolInfo.id);
uint256 transactionFee = feeInfo.protocolFee + feeInfo.lpFee;
if (transactionFee < minTransactionFee) {
    uint256 adjustment = minTransactionFee - transactionFee;
    feeInfo.protocolFee += adjustment;
}

if (amountGD <= feeInfo.protocolFee + feeInfo.lpFee) {
    feeInfo.amountGD = 0;
} else {
    feeInfo.amountGD = amountGD - feeInfo.protocolFee - feeInfo.lpFee;
}
```

So if the owner sets `minimumFeeGD amountGD`, the "amount after fees" is simply 0: effectively, the entire transfer amount becomes fees. However, this is rather lack of a safeguard or centralization risk as it can only be set by the owner.

Recommendation

Add a sanity threshold so the fees cannot exceed certain level e.g. 20%.

Team Response

Acknowledged.

[I-01] Redundant Deadline Check in `_swapAndTransfer()`

Severity

Informational Risk

Description

The `_swapAndTransfer()` function performs a validation check `dstSwapInfo.deadline < block.timestamp`. However, this function is `private` and only invoked by `onReceivePool()`. The `onReceivePool()` function already validates the deadline via the `isSwapExecutable` helper before `_swapAndTransfer()` can be reached.

If `dstSwapInfo.deadline <= block.timestamp`, `isSwapExecutable()` returns `false`, causing `onReceivePool()` to exit early. Therefore, the secondary check inside `_swapAndTransfer()` is dead code that is guaranteed to be false.

Location of Affected Code

File: src/dex/DexReceiverUniswap.sol#L120-L199

```
function onReceivePool( string memory dstChannel, uint64 sequence,
    address token, uint256 amount, bytes memory payload ) external
    onlyTokiBridge nonReentrant returns (bool) {
     // code
     (bool isExecutable, ValidateReceiverResult reason) = isSwapExecutable
        (dstSwapInfo); // returns bool, ValidateReceiverResult

     // code
     bool success = _swapAndTransfer(amount, token, dstSwapInfo);

     // code
}
```

File: src/dex/DexReceiverUniswap.sol#L241-L258

```
function isSwapExecutable( IDexUniswap.DstSwapInfo memory dstSwapInfo )
    public view returns (bool, ValidateReceiverResult) {
     // code
     if (dstSwapInfo.deadline <= block.timestamp) {
         return (false, ValidateReceiverResult.InvalidDeadline);
     }
}
```

File: src/dex/DexReceiverUniswap.sol#L262-L330

```
function _swapAndTransfer(uint256 amountIn, address token, IDexUniswap.
    DstSwapInfo memory dstSwapInfo) private returns (bool) {
     // code
     if (dstSwapInfo.deadline < block.timestamp) {
         return false;
     }
}
```

Impact

The redundancy adds unnecessary lines of code and marginally increases deployment gas costs. It does not introduce a security vulnerability, but removing it improves code clarity and efficiency.

Recommendation

Remove the redundant deadline check from `_swapAndTransfer()`.

```
function _swapAndTransfer(uint256 amountIn, address token, IDexUniswap.
    DstSwapInfo memory dstSwapInfo) private returns (bool) {
-    // slither-disable-next-line timestamp
-    if (dstSwapInfo.deadline < block.timestamp) {
-        return false;
-    }
}
```

Team Response

Fixed.

[I-02] Native Token Trapping in `executeFullSwapNative()`

Severity

Informational Risk

Description

In `src/dex/DexBridgeRouteUniswap.sol`, the `executeFullSwapNative()` entrypoint handles flows where the user pays with native ETH and the contract prepends a WRAP_ETH command before routing the swap through Uniswap's Universal Router, so the swap path is implicitly expected to start from the canonical wrapped native token for the chain.

However, while the inline comment acknowledges this assumption, the implementation does not enforce that `srcSwapInfo.swapParams[0].path[0].tokenIn` is actually the canonical WETH address, meaning a misconfigured path that starts from another token will only be detected when `UNIVERSAL_ROUTER.execute()` reverts. This causes wasted gas and potential user confusion.

Recommendation

Add an explicit validation in the native entrypoints (e.g. `executeFullSwapNative()` and `executeSrcSwapOnlyNative()`) that checks `srcSwapInfo.swapParams[0].path[0].tokenIn` against a stored `wrappedNativeToken` address for the chain.

Team Response

Fixed.

shieldify

Thank you!