GLUEX SECURITY REVIEW

GlueX Router V1

JULY 2025

Prepared by

Pelz

Introduction

A time-boxed security review of the GluexRouter protocol was done by Pelz, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About GlueXRouter

The GluexRouter is a decentralised token swap router designed to facilitate seamless and efficient trading across various liquidity pools. It supports both native tokens and ERC20 tokens, enabling users to interact with liquidity pools through dynamic routing. The system includes features for fee management, slippage tolerance, and flexible route execution, making it a versatile solution for decentralised trading.

Severity Classification

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

Impact

- High - Leads to a significant loss of assets in the protocol or significantly harm a group of users
- Medium - Only a small amount of funds is lost or core contract functionality is broken or affected
- Low - Can lead to any kind of unexpected behaviour with no major impact

Likelihood

- High - Attack path is possible with reasonable assumptions that mimic on chain conditions and the cost of the attack is relatively low compared to the value lost or stolen
- Medium - Only a conditionally incentivised attack vector but still likely
- Low - Has too many or too unlikely assumptions

Actions Required For Severity Levels

- High - Must fix (before deployment, if not already deployed)
- Medium - Should fix
- Low - Could fix

Security Assessment Summary

review commit hash:

- 9c754a3985fa32b72d847c88c83575f29a86bc01

The following number of issues were found, categorised by their severity:

Critical & High: 3 issues

Low : 2 issues

Findings Summary

ID	Title	Severity	Status
[H-01]	**Routing Fee from Swap Is Always Zero in `else if` Branch Due to Incorrect Assignment Order**	**High**	**Resolved**
[H-02]	**Missing Validation on Partner Slippage Share Allows Excessive Extraction**	**High**	**Resolved**
[H-03]	**Underflow Occurs When `slippage > 0` but `surplus == 0`**	**High**	**Resolved**

[L-01]	**The `indexed` Keyword in Events Causes Data Loss for Variables of type `bytes`**	**Low**	**Resolved**
[L-02]	**Missing Sanity Check on `desc.partnerAddress` Can Lead to Token Burn**	**Low**	**Resolved**

[H-01] Routing Fee from Swap Is Always Zero in `else if` Branch Due to Incorrect Assignment Order

Severity

Impact: High

Likelihood: High

Description

In the `swap` function, when the condition falls into the `else if` branch:

```
} else if (finalOutputAmount > desc.effectiveOutputAmount) {
    finalOutputAmount = desc.effectiveOutputAmount;
    routingFee = finalOutputAmount - desc.effectiveOutputAmount; // @audit-is
}
```

The assignment of `finalOutputAmount` to `desc.effectiveOutputAmount` **before** calculating the `routingFee` causes the routing fee to always be `0`. This is because the subtraction becomes `desc.effectiveOutputAmount - desc.effectiveOutputAmount`, which equals zero.

This results in the routing fee being completely bypassed in scenarios where the `finalOutputAmount` exceeds the `effectiveOutputAmount`. As a consequence, the protocol may miss out on fees that should have been collected, and any excess output may not be properly accounted for.

Recommendations

Reorder the assignments to compute the routing fee **before** overwriting `finalOutputAmount`, like so:

```
} else if (finalOutputAmount > desc.effectiveOutputAmount) {
    routingFee = finalOutputAmount - desc.effectiveOutputAmount;
    finalOutputAmount = desc.effectiveOutputAmount;
}
```

This ensures that the correct routing fee is calculated and charged when there is excess output beyond the effective output amount.

[H-02] Missing Validation on Partner Slippage Share Allows Excessive Extraction

Severity

Impact: High

Likelihood: Medium

Description

Within the `swap` function, there is a logic block that distributes `surplus` and `slippage` to the partner:

```
if (surplus > 0 || slippage > 0) {
    // Calculate and transfer partner surplus
    uint256 partnerSurplus = (surplus * desc.partnerSurplusShare) / 10000;
    uint256 partnerSlippage = (slippage * desc.partnerSlippageShare) / 10000;
```

While a validation exists to ensure the partner does not receive an excessive portion of the `surplus` via `desc.partnerSurplusShare`, there is **no equivalent validation** for the `desc.partnerSlippageShare`. As a result, a malicious or misconfigured partner could set

`partnerSlippageShare` to 10000 (i.e., 100%) and extract the **entire slippage amount**, bypassing the intended limit.

This contradicts the declared slippage share limit in the contract:

```
uint256 public _PARTNER_SLIPPAGE_SHARE_LIMIT = 3300; // 33% (3300 bps
```

However, this value is **never enforced** in the `validateSwap` function:

```
function validateSwap(RouteDescription calldata desc) internal view {
    ...
    // Validate partner surplus share
    if (desc.partnerSurplusShare > _PARTNER_SURPLUS_SHARE_LIMIT) revert
    ...
}
```

The absence of a corresponding check for `partnerSlippageShare` creates a critical gap in validation and breaks the protocol's fee fairness assumptions.

Recommendations

Update the `validateSwap` function to include a validation check for the partner's slippage share and also the protocolSlippageShare, similar to the surplus share validation:

```
if (desc.partnerSlippageShare > _PARTNER_SLIPPAGE_SHARE_LIMIT) {
    revert PartnerSlippageShareTooHigh();
}

if (desc.protocolSlippageShare > _PARTNER_SLIPPAGE_SHARE_LIMIT) {
    revert ProtocolSlippageShareTooHigh();
}
```

[H-03] Underflow Occurs When `slippage > 0` but `surplus == 0`

Severity

Impact: High

Likelihood: Medium

Description

In the `swap` function, the following logic is used to handle surplus and slippage distribution:

```
if (surplus > 0 || slippage > 0) {
    uint256 partnerSurplus = (surplus * desc.partnerSurplusShare) / 10000;
    uint256 partnerSlippage = (slippage * desc.partnerSlippageShare) / 10000;
    uint256 partnerShare = partnerSurplus + partnerSlippage;

    uint256 protocolSurplus = surplus - partnerShare; // @audit-issue this can u
}
```

The issue arises when `slippage > 0` but `surplus == 0`. In such cases:

- `partnerSurplus` is `0` (since `surplus == 0`)
- `partnerSlippage` is non-zero (derived from `slippage`)
- `partnerShare = partnerSlippage`
- `protocolSurplus = surplus - partnerShare` becomes `0 - partnerSlippage`, causing an underflow and reverting the transaction

This makes it impossible to complete swaps where slippage is present but surplus is not, even though such a case should be valid.

Recommendations

Instead of subtracting the entire `partnerShare` (which includes both surplus and slippage portions) from `surplus`, only subtract the `partnerSurplus`. This ensures the calculation remains valid even when there is no surplus:

```
uint256 partnerSurplus = (surplus * desc.partnerSurplusShare) / 10000;
uint256 partnerSlippage = (slippage * desc.partnerSlippageShare) / 10000;
uint256 partnerShare = partnerSurplus + partnerSlippage;

uint256 protocolSurplus = surplus - partnerSurplus;
```

This adjustment avoids underflows and correctly separates the logic for handling surplus and slippage values.

[L-01] The `indexed` Keyword in Events Causes Data Loss for Variables of type `bytes`

Severity

Impact: Low

Likelihood: Medium

Description

In the `GlueXRouter.sol` contract, the `Routed` event is defined as follows:

```
event Routed(
    bytes indexed uniquePID,
    address indexed userAddress,
    address outputReceiver,
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 outputAmount,
    uint256 partnerFee,
    uint256 routingFee,
    uint256 finalOutputAmount,
    uint256 surplus
);
```

Here, the `uniquePID` parameter of type `bytes` is marked as `indexed`. This is problematic because when the `indexed` keyword is used on reference types such as `bytes`, `string`, or dynamic arrays, Solidity does not store the actual value in the event log. Instead, it stores the `Keccak-256` hash of the encoded value.

As a result, any listener such as a frontend UI, indexer, or backend service will receive an opaque 32-byte hash in place of the actual `uniquePID` data. This defeats the purpose of event logging as a reliable off-chain communication mechanism, and can lead to mismatches, confusion, or even complete loss of important context if the original data is not recoverable elsewhere.

This design flaw can break event decoding and filtering logic. For more details on how `indexed` works with dynamic types, refer to the Solidity documentation:

Solidity ABI Specification - Events

Recommendations

- Remove the `indexed` keyword from the `bytes` parameter:

```
event Routed(
    bytes uniquePID,
    address indexed userAddress,
    address outputReceiver,
    IERC20 inputToken,
    uint256 inputAmount,
    IERC20 outputToken,
    uint256 outputAmount,
    uint256 partnerFee,
    uint256 routingFee,
    uint256 finalOutputAmount,
    uint256 surplus
);
```

- If filtering on this value is critical, consider using a `bytes32` type instead and emitting both the hashed and raw versions separately (e.g., `bytes32 indexed hashedPID, bytes uniquePID`).

[L-02] Missing Sanity Check on `desc.partnerAddress` Can Lead to Token Burn

Severity

Impact: Low

Likelihood: Low

Description

Within the `swap` function, when calculating and transferring the partner's share of surplus/slippage, the following block executes if the computed share is greater than zero:

```
if (partnerShare > 0) {
    uniTransfer(desc.outputToken, desc.partnerAddress, partnerShare); // @au
}
```

However, there is no validation to ensure that `desc.partnerAddress` is not the zero address. If a zero address is passed in as the `partnerAddress`, and the `outputToken` is the native token (e.g. ETH), the call to `uniTransfer` will silently burn the tokens, causing an irreversible loss of funds.

Even for ERC20 tokens, transferring to the zero address may result in different behavior depending on the token implementation, some may revert, while others may burn the tokens. Either way, the lack of validation introduces a serious risk.

The contract already includes a utility function `checkZeroAddress()` which is used elsewhere to guard against this exact scenario. However, it is not used here.

Recommendations

Add a zero address check for `desc.partnerAddress` before performing the transfer. This can be done inline within the `if` block:

```
if (partnerShare > 0) {
    checkZeroAddress(desc.partnerAddress); // Sanity check to prevent token
    uniTransfer(desc.outputToken, desc.partnerAddress, partnerShare);
}
```

This ensures tokens are never transferred to an invalid address and prevents accidental burns due to misconfiguration or malicious input.