GlueX Router Security Review

Prepared By Pelz

December 2024

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:

- 1df4eaef9c3063a3171961e1f8bba3eb83c6b7e1
- 84709cb973df0426fd59062ec872138bf6a7f53b

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

Critical & High: 2 issues

Medium: 2 issues

Low & Info: 1 issues

Findings Summary

ID	Title	Severity	Status
[H-01]	**Bypassing Protocol Fees via Manipulated routingFee**	High	Resolved
[H-02]	**Incorrect Slippage Check Allows Unnecessary Swap Failures**	High	Resolved
[M-01]	**Missing Check for `msg.value` with Non-Native Tokens in `swap` Function**	Medium	Resolved
[M-02]	**Potential Loss Of Funds Due to Missing Validation of `outputReceiver`**	Medium	Resolved
[I-01]	**[I-01] Use of Floating Pragma**	Informational	Acknowledged

[H-01] Bypassing Protocol Fees via Manipulated routingFee

Severity

Impact: High
Likelihood: High

Description

The `routingFee` field in the `RouteDescription` struct is intended to charge users a fee for executing swaps via the `GlueXRouter`. However, the only validation currently applied ensures that the `routingFee` is greater than zero:

```
require(desc.routingFee > 0, "Negative routing fee");
```

This minimal check allows users to manipulate the calldata of the `RouteDescription` struct and set an arbitrarily low fee (e.g., 1 wei), effectively bypassing the intended protocol fees. This vulnerability puts the protocol at risk of significant revenue loss.

Relevant Code

```
/// @dev generic route description
struct RouteDescription {
    IERC20 inputToken;
    IERC20 outputToken;
    address payable inputReceiver;
    address payable outputReceiver;
    uint256 inputAmount;
    uint256 outputAmount;
    uint256 routingFee;
    uint256 effectiveOutputAmount;
    uint256 minOutputAmount;
    bool isPermit2;
}

// from the swap function
require(desc.routingFee > 0, "Negative routing fee");
```

Impact

- **Revenue Loss**: Users can manipulate the \`routingFee\` to avoid paying fair fees to the protocol, leading to a substantial loss of revenue.

Recommendations

1. **Enforce Minimum and Maximum Routing Fee:**

Introduce strict validation for the `routingFee` to ensure it falls within a reasonable range:

```
require(desc.routingFee >= MIN_ROUTING_FEE, "Routing fee too
require(desc.routingFee <= MAX_ROUTING_FEE, "Routing fee too
```

Define `MIN_ROUTING_FEE` and `MAX_ROUTING_FEE` as constants in the contract to establish acceptable boundaries.

2. **Calculate Fees Dynamically:**

Rather than relying solely on user input, calculate the `routingFee` dynamically based on the swap value or other metrics. For example:

```
    uint256 expectedFee = (desc.inputAmount * FEE_PERCENTAGE)
    require(desc.routingFee >= expectedFee, "Routing fee too l
```

[H-02] Incorrect Slippage Check Allows Unnecessary Swap Failures

Severity

Impact: Medium
Likelihood: High

Description

The `swap` function in the `GlueXRouter` includes a check to ensure that the `finalOutputAmount` from a swap is greater than the `minOutputAmount` specified by the user. However, the current implementation requires the `finalOutputAmount` to strictly exceed the `minOutputAmount`:

```
require(finalOutputAmount > desc.minOutputAmount, "Negative s
```

This is problematic because users typically set `minOutputAmount` to account for slippage. If the `finalOutputAmount` equals `minOutputAmount`, the transaction unnecessarily reverts, even though this should be acceptable behaviour given the user's slippage tolerance.

This incorrect validation leads to failed swaps, potentially causing users to lose gas fees and miss trade opportunities.

Impact

Swaps that result in exactly the `minOutputAmount` will revert unnecessarily, wasting user gas fees and potentially causing users to miss profitable trades.

Recommendation

Update the slippage validation to allow `finalOutputAmount` to meet or exceed the `minOutputAmount`:

```
require(finalOutputAmount >= desc.minOutputAmount, "Negative
```

This change will align the slippage check with user expectations and reduce unnecessary reverts, enhancing the protocol's usability.

[M-01] Missing Check for `msg.value` with Non-Native Tokens in `swap` Function

Severity

Impact: Medium
Likelihood: Medium

Description

The `swap` function in `GlueXRouter.sol` is designed to handle both native token and ERC20 token inputs. While it verifies the `msg.value` when the input token is a native token, it fails to check whether `msg.value` is zero when the input token is an ERC20 token.

This oversight can lead to a loss of funds, as users may unintentionally send ETH (`msg.value`) alongside an ERC20 token transaction.

Relevant Code

```
bool isNativeTokenInput = (address(desc.inputToken) == _nativ
if (isNativeTokenInput) {
    require(msg.value == desc.inputAmount, "Invalid native to
}

IERC20 inputToken = desc.inputToken;
if (!isNativeTokenInput) {
    inputToken.safeTransferFromUniversal(msg.sender, desc.inp
}
```

Impact

- If `isNativeTokenInput` is `false`, there is no check to ensure that `msg.value` is zero.
- Users sending `msg.value` along with ERC20 token inputs will inadvertently lose those funds.

Recommendation

Add a validation check to ensure that `msg.value` is zero when the input token is not a native token:

```
if (!isNativeTokenInput) {
    require(msg.value == 0, "ETH value must be zero for ERC20
    inputToken.safeTransferFromUniversal(msg.sender, desc.inp
}
```

[M-02] Potential Loss Of Funds Due to Missing Validation of `outputReceiver`

Severity

Impact: Medium
Likelihood: Medium

Description

The `swap` function in `GlueXRouter.sol` attempts to transfer the swap output to the `outputReceiver` address via the `uniTransfer` function. However, it does not validate whether the `outputReceiver` is a valid address.

If the `outputReceiver` is set to `address(0)`, the output funds from a successful swap will either be irretrievably lost (for native tokens) or cause a failed transaction (for ERC20 tokens). This creates a significant risk of user funds being lost due to incorrect or malicious configurations.

Relevant Code

```
uniTransfer(outputToken, desc.outputReceiver, finalOutputAmou
```

The `uniTransfer` function implementation:

```
function uniTransfer(
    IERC20 token,
    address payable to,
    uint256 amount
) internal {
    if (amount > 0) {
        if (address(token) == _nativeToken) {
            if (address(this).balance < amount) revert Insuff
            (bool success, ) = to.call{value: amount, gas: _R
            if (!success) revert NativeTransferFailed();
        } else {
            token.safeTransfer(to, amount);
        }
    }
}
```

Impact

The `uniTransfer` function will attempt to transfer funds to an invalid address without validation, leading to unintended outcomes:

- **Native Tokens:** Funds sent to an invalid address are effectively burned.

- **ERC20 Tokens:** The transaction fails due to the `safeTransfer` function rejecting transfers to `address(0)` .

Recommendation

Modify the `swap` function to check if the `outputReceiver` is set to `address(0)` and, if so, default it to `msg.sender`.

```
// Ensure the output receiver address is valid
address payable receiver = desc.outputReceiver == address(0)

// Proceed with transfer
uniTransfer(outputToken, receiver, finalOutputAmount);
```

This change ensures that funds are safely redirected to the transaction initiator (`msg.sender`) in cases where the `outputReceiver` is not specified, preventing loss of funds and maintaining functionality.

[I-01] Use of Floating Pragma

Impact: Informational
Likelihood: Low

Description

The contracts use a floating Solidity pragma (`^0.8.0`), which allows the compiler to use any version from `0.8.0` to below `0.9.0.` While this provides flexibility, it can expose the contracts to unexpected behaviour or incompatibilities introduced in newer versions of Solidity.

Recommendation

Specify a fixed compiler version (e.g., `pragma solidity 0.8.28;`) to ensure consistent compilation behaviour and avoid unexpected issues with future Solidity updates.