GLUEX SECURITY REVIEW

GlueX Router V1_2

AUGUST 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 Gluex Protocol V1

Gluex Protocol is a modular settlement and routing system designed to automate complex token swaps and asset movements in DeFi. It allows protocols to create custom settlement flows using flashloans from Aave, handle margin deposits, and execute flexible callbacks before and after swaps. The system is built to support advanced trading, lending, and cross-chain settlement use cases, with an extensible design for building custom modules.

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:
e63cf6631cdeb1eff2a38189e11c20eaf7a88edb

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

Critical & High: 2 issues

Medium: 2 issues

Low & Informational : 8 issues

Findings Summary

ID	Title	Severity	Status
[H-01]	Anyone Can Drain Protocol-Owned ERC20 Tokens via Public `settle()` and Incorrect Balance Handling in `handleTokenTransfers()`	High	Resolved
[H-02]	Settlement Flow Permanently Broken: Incorrect Borrow Logic Uses `msg.sender` Instead of User Address in `executePostRouteCallback()`	High	Resolved
[M-01]	Native Token Settlements Can Be DoS'ed Due to Incorrect Balance Check in `handleTokenTransfers()` Used by `settle()`	Medium	Resolved
[M-02]	Improper Approval Logic Allows Temporary Denial of Service in Structured Settlements For Some Tokens	Medium	Resolved
[L-01]	Missing Zero Address Check for `nativeToken` in Constructor	Low	Resolved
[L-02]	Unnecessary `payable` Modifiers Cause ETH Sent to Settlement Functions to Be Permanently Stuck in Contract	Low	Resolved
[L-03]	Incorrect Free Memory Pointer Update in `skipSelector()` May Cause Unexpected Execution Failures or Malformed Settlements	Low	Resolved
[L-04]	Unrestricted Settlers and Missing Native Token Value Checks Enable Direct Draining of Protocol Native Tokens	Low	Resolved
[I-01]	Ensure Event Emissions Are Implemented and TODO Comments Are Removed	Informational	Resolved

ID	Title	Severity	Status
[I-02]	**Ensure `executePreRouteCallback()` Reverts to Prevent Unintended Use**	Informational	Resolved
[I-03]	**Unnecessary `{value: 0}` in GlueX Router Call**	Informational	Resolved
[I-04]	**Redundant `using SafeERC20` Declaration in `GluexAaveV3FlashLoanSimple`**	Informational	Resolved

[H-01] Anyone Can Drain Protocol-Owned ERC20 Tokens via Public `settle()` and Incorrect Balance Handling in `handleTokenTransfers()`

Severity

High

Impact: High
Likelihood: High

Description

The `settle()` function in the `GluexProtocolSettlement` contract is publicly accessible, allowing anyone to initiate settlements. This function relies on an internal helper, `handleTokenTransfers()`, to process token transfers.

In `handleTokenTransfers()`, when handling ERC20 tokens (non-native tokens), the contract checks whether it already holds enough balance of the `inputToken`. If so, it assumes those tokens are meant for the current settlement and transfers them directly to the specified receiver:

```
} else {
    if (address(desc.inputToken) != _nativeToken) {
        desc.inputToken.safeTransfer(desc.inputReceiver,
desc.inputAmount);
    }
}
```

The issue arises because this check does **not verify whether those tokens were supplied by the caller or intended for the current settlement.** Instead, it relies only on whether the contract has sufficient balance of the `inputToken` at the time of the function call.

This creates a dangerous situation:

- If the contract holds any ERC20 token balance (for example, from protocol fees, accidental transfers, or rounding dust), **any external user can exploit this.**
- An attacker can monitor token balances held by the contract, then submit a settlement request using that token and specify themselves (or any address) as the receiver.
- The protocol will treat the available tokens as settlement funds and transfer them directly to the attacker.

Effectively, **any ERC20 tokens held in the contract can be stolen by anyone** simply by calling the public `settle()` function with a carefully crafted request.

This vulnerability puts all protocol-owned ERC20 tokens at direct risk.

Recommendations

Remove reliance on pre-existing token balances.

For ERC20 tokens, the settlement process should always require tokens to be transferred directly from the caller for each settlement.

Suggested approach:

- Remove the current balance check-based logic.
- Always enforce:

  ```
  desc.inputToken.safeTransferFrom(msg.sender, desc.inputReceiver,
  desc.inputAmount);
  ```

This ensures that only the tokens actively supplied by the caller are processed even if they are structured settlements, preventing misuse of any tokens already held in the contract.

[H-02] Settlement Flow Permanently Broken: Incorrect Borrow Logic Uses `msg.sender` Instead of User Address in `executePostRouteCallback()`

Severity

High

Impact: High
Likelihood: High

Description

In the `GluexAaveV3FlashLoanSimple.sol` contract, the `executePostRouteCallback()` function is designed to borrow funds from Aave on behalf of the user to finalize flashloan repayments. However, the implementation incorrectly uses `msg.sender` (the GlueX Router) as the `onBehalfOf` parameter in the borrow call:

```
IPool(settlementTrigger).borrow(asset, amount, interestRateMode,
referralCode, msg.sender);
```

Since this function is restricted to `onlyGluexRouter`, `msg.sender` will **always be the GlueX Router**. For the borrow to succeed:

- The GlueX Router itself would need to supply collateral to Aave (which it does not).
- The Router would also need to delegate borrowing rights to itself, which is against standard protocol design.

As a result:

- Every attempt to execute `executePostRouteCallback()` will consistently **fail and revert**, breaking settlement logic.
- Flashloan settlement flows cannot proceed, effectively halting critical functionality of the protocol.
- No user debt positions are opened, but settlement calls relying on the borrow will fail entirely.

While funds aren't directly stolen, the inability to complete settlements leads to broken protocol functionality.

Recommendations

Use `tx.origin` as the Borrower Address

Replace:

```
IPool(settlementTrigger).borrow(asset, amount, interestRateMode,
referralCode, msg.sender);
```

With:

```
IPool(settlementTrigger).borrow(asset, amount, interestRateMode,
referralCode, tx.origin);
```

This allows the protocol to correctly borrow on behalf of the **original transaction sender** (the user), assuming they have pre-approved delegation to the settlementTrigger contract and Since the user is always the true initiator of the transaction, using `tx.origin` aligns with the intended debt holder, also the Aave protocol itself ensures that delegation must exist for borrowing to succeed, preventing unauthorized borrowing.

Important Risks of Using `tx.origin`

Using `tx.origin` introduces a potential **man-in-the-middle (MITM) attack vector**:

- If external systems or malicious contracts can route calls through your GlueX Router, they can trick the Router into opening debt positions on behalf of the `tx.origin` unexpectedly.
- For example, if a user unknowingly interacts with a contract that triggers GlueX settlements using leftover delegation approvals, that contract could borrow against the user's credit line without their awareness.
- This is especially risky if users grant broad approvals to the settlementTrigger.

While Aave's delegation permissions limit abuse, the risk still exists

To mitigate:

- Clearly document the risk for integrators and users.
- Encourage users to limit delegation approvals.

[M-01] Native Token Settlements Can Be DoS'ed Due to Incorrect Balance Check in `handleTokenTransfers()` Used by `settle()`

Severity

Medium

Impact: Medium
Likelihood: Medium

Description

Within the `settle()` function of `GluexProtocolSettlement.sol`, token transfers are handled via the internal `handleTokenTransfers()` function. This internal function performs a strict balance equality check to determine whether settlement input tokens are available:

```
uint256 inputBalance = uniBalanceOf(desc.inputToken, address(this));

if (inputBalance != desc.inputAmount) {
    if (address(desc.inputToken) == _nativeToken) {
        revert InvalidNativeTokenInputAmount();
    }
    ...
}
```

In cases where the input token is the native token (`ETH`/`WETH`), the contract requires its native token balance to be **exactly equal** to the required input amount before allowing the settlement to proceed.

```
if (inputBalance != desc.inputAmount)
```

If the contract holds **any additional native token balance** even 1 wei, due to protocol fees, rounding dust, or intentional frontrunning deposits this strict equality check will fail, causing the contract to revert with:

```
revert InvalidNativeTokenInputAmount();
```

This causes the contract to reject any settlement where the protocol holds even minimal additional native token balances.

Recommendations

- Modify the validation logic in `handleTokenTransfers()` to allow for pre-existing balances:

  ```
  if (inputBalance < desc.inputAmount) {
      revert InvalidNativeTokenInputAmount();
  }
  ```

[M-02] Improper Approval Logic Allows Temporary Denial of Service in Structured Settlements For Some Tokens

---

Severity

Medium

Impact: Medium
Likelihood: Medium

Description

In the `executePostRouteCallback()` function of the `GluexAaveV3FlashLoanSimple.sol` contract, the following approval logic is used after borrowing assets from Aave:

```
IERC20(asset).approve(settlementTrigger, amount);
```

This design assumes that setting a fresh approval for `settlementTrigger` will always succeed. However, many widely-used ERC20 tokens like **USDT**, and **BUSD** implement **non-standard approval behavior**. These tokens require that existing approvals be set to zero before a new non-zero approval can be made.

An Example Attack:
A malicious user could:

- Use the post-route callback mechanism to borrow a large but permissible amount(as his collateral permits).
- The flashloan pool then Partially consume the approval(taking only what it needs to repay the flashloan).
- This causes the contract to enter a stuck state for that token, where future settlement attempts revert due to approval failure.

And as a result, creates a **temporary Denial of Service (DoS)**, as no future settlements involving that token can proceed until the existing allowance is manually reset

---

Recommendations

Replace the vulnerable approval pattern:

```
IERC20(asset).approve(settlementTrigger, amount);
```

With OpenZeppelin's **forceApprove()** method:

```
asset.forceApprove(settlementTrigger, amount);
```

[L-01] Missing Zero Address Check for `nativeToken` in Constructor

Severity

Low

Impact: Low
Likelihood: Low

Description

The constructor of the `GluexProtocolSettlement` contract validates that the `gluexTreasury` address is not the zero address by calling the `checkZeroAddress()` function. However, it fails to perform a similar check for the `nativeToken` parameter.

This omission allows deployment of the contract with `_nativeToken` set to the zero address (`address(0)`), which may lead to unintended behavior when interacting with `_nativeToken` in downstream token transfer or settlement logic.

Performing a zero address check on both constructor parameters would ensure consistent input validation and prevent incorrect contract initialization.

Recommendations

Add a zero address check for the `nativeToken` parameter in the constructor:

```
checkZeroAddress(nativeToken);
```

Revised constructor:

```
constructor(address gluexTreasury, address nativeToken) {
    checkZeroAddress(gluexTreasury);
    checkZeroAddress(nativeToken);

    _gluexTreasury = gluexTreasury;
    _nativeToken = nativeToken;
}
```

and also in the GluexAaveV3FlashLoanSimple.sol:

```
    constructor(address _router, address _settlementTrigger) {
        require(_router != address(0), "GlueX: zero router");
        require(_settlementTrigger != address(0), "GlueX: zero settlement
trigger");
        gluexRouter = _router;
        settlementTrigger = _settlementTrigger; // The external contract
that owns the underlying effective logic of the settlement
    }
```

[L-02] Unnecessary `payable` Modifiers Cause ETH Sent to Settlement Functions to Be Permanently Stuck in Contract

---

Severity

Low

Impact: Low
Likelihood: Low

Description

In the `GluexAaveV3FlashLoanSimple.sol` contract, the following functions are marked as `payable` but do not process or forward any native tokens (ETH) sent along with their calls:

```
function executeStructuredSettlement(bytes calldata settlementParams)
external payable override { ... }

function executePostRouteCallback(bytes calldata data) external payable
override onlyGluexRouter { ... }

function executePreRouteCallback(bytes calldata data) external payable
override onlyGluexRouter { ... }
```

Within these functions:

- `msg.value` is not used.
- No logic refunds ETH to the sender or forwards it to another address.
- As a result, any ETH accidentally or intentionally sent alongside these function calls will become trapped in the contract, without a mechanism to recover or withdraw it.

Recommendations

- **Remove the `payable` modifier** from all three functions unless they are explicitly intended to handle or forward ETH.

Example fix:

```
function executeStructuredSettlement(bytes calldata settlementParams)
external override { ... }
```

- Alternatively, if ETH payments are expected for future use:
    - Include clear logic to handle `msg.value`.
    - Ensure excess ETH is refunded to the sender.
    - Or route the received ETH to the protocol treasury or designated address.
- Optionally, implement a `withdraw()` function restricted to the protocol owner, allowing accidental ETH recovery if future design considerations require keeping functions `payable`.

[L-03] Incorrect Free Memory Pointer Update in `skipSelector()` May Cause Unexpected Execution

Failures or Malformed Settlements

Severity

Low

Impact: Low
Likelihood: Low

Description

In the `GluexAaveV3FlashLoanSimple.sol` contract, the `skipSelector()` function is used to slice calldata by removing the first 4-byte selector. This function manually copies data using inline assembly and attempts to update Solidity's free memory pointer at the end of the function:

```
mstore(0x40, add(dest, len))
```

This update is incorrect because it fails to account for the 32 bytes used to store the array length before the actual data. As a result, the pointer may point into the middle of the allocated memory space.

While Solidity memory is temporary and resets between external calls, improper management of the free memory pointer can affect any dynamic memory allocations that occur later in the same transaction. Subsequent variables might be allocated into already-used memory, leading to:

- Unintended overwrites of earlier data.
- Corrupted arrays or dynamic variables.
- Unexpected decoding errors or malformed parameters.

In this contract, the result of `skipSelector()` is used immediately within the `executeStructuredSettlement()` function to decode settlement instructions. While Solidity's `abi.decode()` typically operates directly on calldata, improper pointer management can still result in fragile execution where unexpected allocations or decoding failures may occur during settlement operations.

Although this issue does not directly lead to fund loss or a security breach, it increases the risk of unpredictable behavior and failed settlement flows.

Recommendations

Replace the incorrect free memory pointer update:

```
mstore(0x40, add(dest, len))
```

With:

```
mstore(0x40, add(result, add(0x20, len)))
```

This adjustment ensures that the pointer correctly moves past both the array's 32-byte length prefix and the actual data, preserving memory safety for subsequent dynamic allocations.

[L-04] Unrestricted Settlers and Missing Native Token Value Checks Enable Direct Draining of Protocol Native Tokens

---

Severity

Low

Impact: Low
Likelihood: Low

Description

The `settle()` function in `GluexProtocolSettlement.sol` exposes critical flaws when two design issues are combined:

1. **Unrestricted Settler Control**
   Any external contract can act as a settler simply by being the `msg.sender`. The protocol does not whitelist or restrict who can trigger settlement callbacks. This allows attackers to deploy malicious contracts that masquerade as settlers and execute arbitrary callback logic during the settlement process.

2. **Blind Native Token Forwarding Without Sanity Checks**
   The protocol forwards native tokens (`msg.value`) to both pre-route and post-route callbacks based solely on caller-supplied values:

   ```
   IGluexSettler(msg.sender).executePreRouteCallback{value:
   preRouteCallbackParams.value}(preRouteCallbackParams.data);
   ...
   IGluexSettler(msg.sender).executePostRouteCallback{value:
   postRouteCallbackParams.value}(postRouteCallbackParams.data);
   ```

   However:

   - There is no check ensuring that the cumulative value of `preRouteCallbackParams.value` and `postRouteCallbackParams.value` is less than or equal to `msg.value`.

- The protocol assumes that any value requested for forwarding is legitimate.

Combined Risk:

- Attackers can deploy malicious settler contracts.
- They can pass excessive values in callback parameters, causing the protocol to forward significant native token balances directly into attacker-controlled callbacks.

In effect, **anyone can deploy a settler contract and drain the protocol's native tokens using custom callback logic**, exploiting the lack of restrictions and value controls simultaneously.

Recommendations

- **Restrict who can act as a settler:**
  - Introduce a whitelist or role-based access control mechanism to ensure that only trusted, audited settler contracts can interact with the `settle()` function.
  - Example:

```
mapping(address => bool) public whitelistedSettlers;

modifier onlyWhitelistedSettler() {
    require(whitelistedSettlers[msg.sender], "Unauthorized
settler");
    _;
}
```

- **Validate and limit forwarded native tokens:**
  - Ensure that the total forwarded value does not exceed the transaction's `msg.value`:

```
```solidity
require(msg.value >= preRouteCallbackParams.value +
postRouteCallbackParams.value, "Insufficient msg.value");
```
```

[I-01] Ensure Event Emissions Are Implemented and TODO Comments Are Removed

---

Severity

Informational

Description

Within both the `executeStructuredSettlement()` and `executePostRouteCallback()` functions, there are `TODO` comments referencing the need to add event emissions. These events are important for allowing integrators, indexers, and monitoring systems to track settlement activity and borrowing operations.

To improve protocol observability and support external integrations, ensure these events are properly implemented and the corresponding `TODO` comments are removed.

Recommendations

- Define and emit appropriate events in both functions to track key settlement and borrowing activities.
- After implementing, remove the `TODO` comments to reflect completed development.

[I-02] Ensure `executePreRouteCallback()` Reverts to Prevent Unintended Use

Severity

Informational

Description

The `executePreRouteCallback()` function in the contract is currently implemented as a **no-op** and marked as `payable`:

```
function executePreRouteCallback(bytes calldata data) external payable
override onlyGluexRouter {
    // Default: no-op
}
```

Since the function performs no actions:

- Any native tokens sent during calls to this function will be unnecessarily locked in the contract.
- Integrators or developers using this as a reference could mistakenly assume the function performs meaningful operations.
- The `payable` modifier suggests the function expects native value, which it does not process.

And to enforce proper usage and avoid accidental value transfers, the function should **explicitly revert**.

Recommendations

- Modify the function to revert by default:

```
function executePreRouteCallback(bytes calldata) external payable override
onlyGluexRouter {
    revert("GlueX: pre-route callback unsupported");
}
```

- This will:
    - Prevent unnecessary value forwarding.
    - Block meaningless calls.
    - Serve as a clear signal to future developers that the function requires explicit implementation if needed.
      Here's a clear, concise **informational report** for that case:

[I-03] Unnecessary `{value: 0}` in GlueX Router Call

Severity

Informational

Description

In the `executeOperation()` function, the following call is used to execute the router call:

```
(success,) = gluexRouter.call{value: 0}(params);
```

Since the value explicitly passed is zero and the function performs no native token transfers, specifying `{value: 0}` is unnecessary. This introduces minor code clutter.

Recommendations

- Simplify the call statement by removing the redundant value block:

```
(success,) = gluexRouter.call(params);
```

This improves readability and avoids any mistaken assumption about native token involvement in this execution path.

[I-04] Redundant `using SafeERC20` Declaration in `GluexAaveV3FlashLoanSimple`

Severity

Informational

Description

In the `GluexAaveV3FlashLoanSimple` contract, the following line appears:

```
using SafeERC20 for IERC20;
```

However, the contract already inherits from `AaveV3FlashLoaner`, which applies the `using SafeERC20 for IERC20;` directive. In Solidity, such using directives are inherited by child contracts and apply automatically unless explicitly overridden.

Re-declaring the same directive in the child contract:

- Adds unnecessary duplication.
- Provides no functional or readability benefit.
- May suggest to reviewers that a customization is intended when none exists.

Recommendations

- Remove the redundant declaration from `GluexAaveV3FlashLoanSimple`:

```
// using SafeERC20 for IERC20;
```

Since `SafeERC20` functionality is already available via inheritance, removing this improves code cleanliness without affecting functionality.