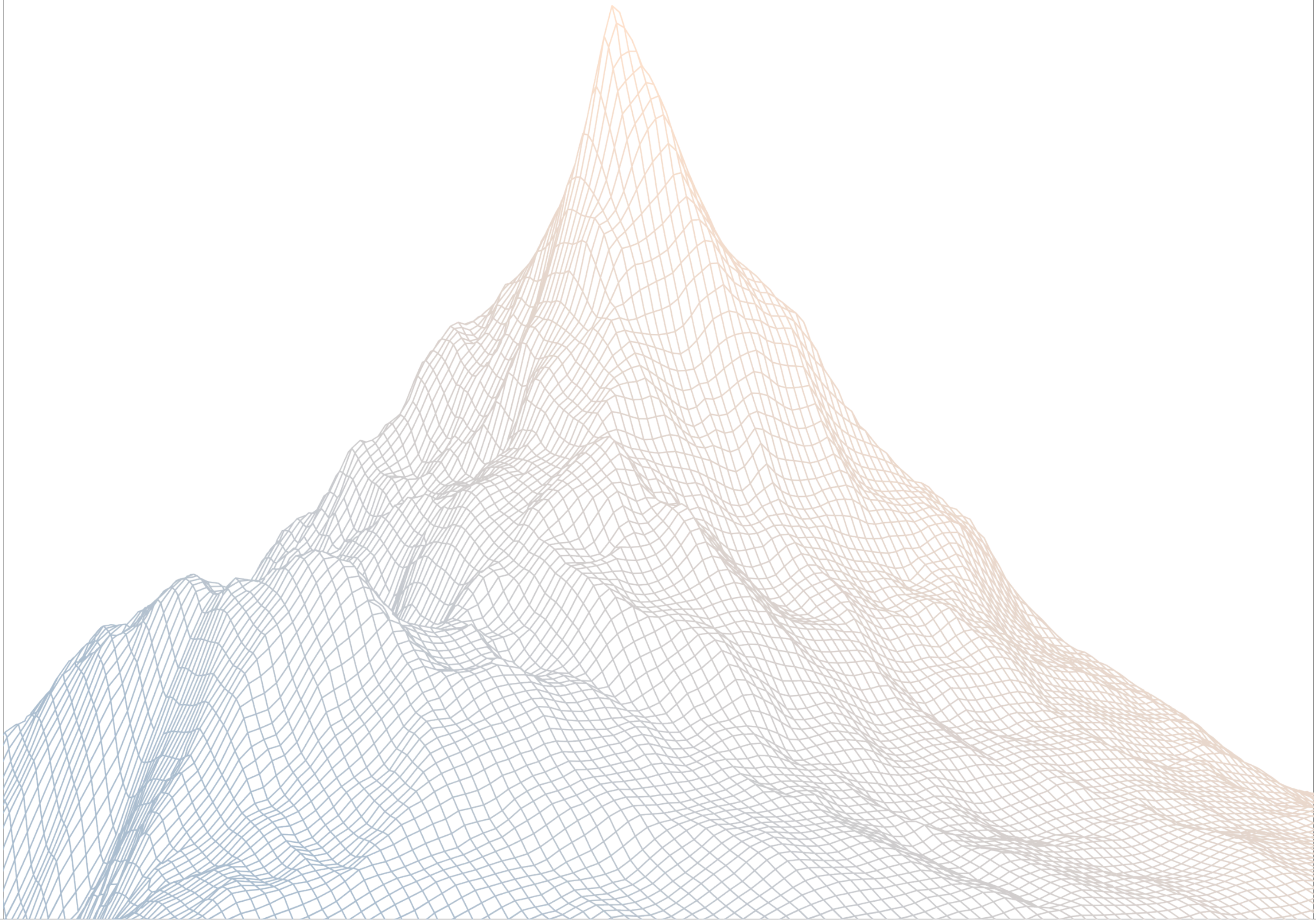
Zenith

Hyperswell

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: March 19th to March 20th, 2025

AUDITED BY:
Arnie
said
sorryNotsorry

Contents

1

Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2 Executive Summary

2.1 About Hyperswell

HyperSwell is the first-ever live implementation of Hyperliquid builder codes on an alternative chain, bringing a groundbreaking approach to decentralized trading. As a next-generation perpetual DEX, HyperSwell seamlessly integrates advanced builder codes to deliver a trading experience that rivals centralized exchanges—without compromising on decentralization. By leveraging these innovations, HyperSwell ensures deep liquidity, low latency, and a smooth, efficient user experience, redefining what's possible in DeFi trading.

2.2 Scope

The engagement involved a review of the following targets:

Target	core-contracts
Repository	https://github.com/HyperSwell-PerpDEX/core-contracts
Commit Hash	`80ef51357e8f731555c9625fbd75cfd5a55a157d`
Files	`contracts/BridgeSwapController.sol`

2.3 Audit Timeline

March 19, 2025	Audit start
March 20, 2025	Audit end
March 21, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	3
Low Risk	2
Informational	0
Total Issues	**5**

3

Findings Summary

ID	Description	Status
M-1	malicious user can create a swapAndBridge order to steal another user's bridged assets.	Resolved
M-2	Not returning excess tokens from swap operation inside lzCompose and swapAndBridge	Acknowledged
M-3	Permit DoS vector on lzCompose and swapAndBridge	Resolved
L-1	feeUSDCAmount Can Be Set above 10e6	Resolved
L-2	SwapBridgeData.receiver is not used as swap receiver inside lzCompose	Acknowledged

4 Findings

4.1 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] malicious user can create a `swapAndBridge` order to steal another user's bridged assets.

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- BridgeSwapController.sol#L122-L166

Description:

user can create and sign the `swapAndBridge` `orderData`, which can later be executed by an executor to fulfill the request. The `swapAndBridge` function will pull USDC from the user, swap it for `erc20Receive`, and bridge it to the specified `bridgeDstEid`.

```
    function swapAndBridge(
        bytes calldata orderData,
        bytes calldata signature
    ) external nonReentrant onlyExecutor payable {
    // ...

        // execute swap
>>>     uint balanceReceived = _executeSwap(usdc, erc20Receive,
    swapBridgeOrder.router, address(this), swapAmount,
    swapBridgeOrder.minSwapAmountOut, swapBridgeOrder.swapData);
        emit Swapped(address(this), address(erc20Receive), swapAmount,
    balanceReceived, block.timestamp);

>>>     (MessagingReceipt memory messagingReceipt, OFTReceipt
    memory oftReceipt) = IOFT(address(erc20Receive)).send{value: msg.value}(
            SendParam(swapBridgeOrder.bridgeDstEid,
    bytes32(uint256(uint160(user))), balanceReceived,
    swapBridgeOrder.minBridgeAmountOut, swapBridgeOrder.bridgeExtraOptions,
    new bytes(0), new bytes(0)),
            MessagingFee(msg.value, 0),
            msg.sender
        );
        if (oftReceipt.amountSentLD < balanceReceived) {
            erc20Receive.safeTransfer(user, balanceReceived
    - oftReceipt.amountSentLD);
        }
        emit SubmitBridge(user, balanceReceived, block.timestamp);

        emit Sent(messagingReceipt.guid, swapBridgeOrder.bridgeDstEid, user,
    address(usdc), swapBridgeOrder.swapAmount, feeUSDCAmount,
    block.timestamp);
    }
```

Inside `_executeSwap`, the router will be validated, the swap will be performed, and the before and after balance will be checked to determine the `balanceReceived`, which will be bridged for the user.

```
    function _executeSwap(
        IERC20 inputToken, IERC20 outputToken, address router,
    address receiver, uint256 amountSwap, uint256 minAmountOut,
    bytes memory swapData
    ) internal returns (uint256 balanceReceived) {
        // _executeSwap(erc20Receive, usdc, bridgeSwapData.router,
    fromAddress, _amountLD, bridgeSwapData.minAmountOut,
    bridgeSwapData.swapData);
        // Execute the token swap
        if (!whitelistRouter[router]) revert InvalidAddress();
        if (minAmountOut < 1) revert InvalidAmount();

        uint balanceBefore = outputToken.balanceOf(receiver);

        inputToken.safeIncreaseAllowance(router, amountSwap);
        // solhint-disable-next-line avoid-low-level-calls
>>>     (bool success, bytes memory res) = router.call(swapData);
        if (!success) {
            string memory reason = LibUtil.getRevertMsg(res);
            revert(string(abi.encodePacked("swap reverted:", reason)));
        }

        balanceReceived = outputToken.balanceOf(receiver) - balanceBefore;
        if (balanceReceived < minAmountOut) revert SwapSlippage();
    }
```

Inside the whitelisted router, Kyberswap Aggregation Router, There is a function that allows provided addresses to perform execution.

```
  function swapSimpleMode(
    IAggregationExecutor caller,
    SwapDescriptionV2 memory desc,
    bytes calldata executorData,
    bytes calldata clientData
  ) public returns (uint256 returnAmount, uint256 gasUsed) {
    uint256 gasBefore = gasleft();

    require(!_isETH(desc.srcToken), 'src is eth, should use normal swap');

    _permit(desc.srcToken, desc.amount, desc.permit);

    address dstReceiver = (desc.dstReceiver == address(0)) ? msg.sender :
    desc.dstReceiver;
    {
      bool isBps = _flagsChecked(desc.flags, _FEE_IN_BPS);
      if (!_flagsChecked(desc.flags, _FEE_ON_DST)) {
        // take fee and deduct total swap amount
        desc.amount = _takeFee(desc.srcToken, msg.sender, desc.feeReceivers,
    desc.feeAmounts, desc.amount, isBps);
      } else {
        dstReceiver = address(this);
      }
    }

    uint256 initialDstBalance = _getBalance(desc.dstToken, dstReceiver);
    uint256 initialSrcBalance = _getBalance(desc.srcToken, msg.sender);
>>> _swapMultiSequencesWithSimpleMode(
      caller,
      address(desc.srcToken),
      desc.amount,
      address(desc.dstToken),
      dstReceiver,
      executorData
    );

 // ...
  }
```

```
  function _swapMultiSequencesWithSimpleMode(
    IAggregationExecutor caller,
    address tokenIn,
    uint256 totalSwapAmount,
    address tokenOut,
    address dstReceiver,
    bytes calldata data
  ) internal {
    SimpleSwapData memory swapData = abi.decode(data, (SimpleSwapData));
    require(swapData.deadline >= block.timestamp, 'ROUTER: Expired');
    require(
      swapData.firstPools.length == swapData.firstSwapAmounts.length &&
        swapData.firstPools.length == swapData.swapDatas.length,
      'invalid swap data length'
    );
    uint256 numberSeq = swapData.firstPools.length;
    for (uint256 i = 0; i < numberSeq; i++) {
      // collect amount to the first pool
      {
        uint256 balanceBefore = _getBalance(IERC20(tokenIn), msg.sender);
        _doTransferERC20(IERC20(tokenIn), msg.sender,
    swapData.firstPools[i], swapData.firstSwapAmounts[i]);
        require(swapData.firstSwapAmounts[i] <= totalSwapAmount, 'invalid
    swap amount');
        uint256 spentAmount = balanceBefore - _getBalance(IERC20(tokenIn),
    msg.sender);
        totalSwapAmount -= spentAmount;
      }
      {
        // solhint-disable-next-line avoid-low-level-calls
        // may take some native tokens for commission fee
>>>     (bool success, bytes memory result) = address(caller).call(
          abi.encodeWithSelector(caller.swapSingleSequence.selector,
    swapData.swapDatas[i])
        );
        if (!success) {
          revert(RevertReasonParser.parse(result, 'swapSingleSequence
    failed: '));
        }
      }
    }
    {
      // solhint-disable-next-line avoid-low-level-calls
      // may take some native tokens for commission fee
>>>   (bool success, bytes memory result) = address(caller).call(
        abi.encodeWithSelector(
          caller.finalTransactionProcessing.selector,
          tokenIn,
          tokenOut,
          dstReceiver,
          swapData.destTokenFeeData
```

```
   )
  );
  if (!success) {
   revert(RevertReasonParser.parse(result, 'finalTransactionProcessing
 failed: '));
  }
 }
}
```

This open attack vector with the following scenario :

1. An attacker monitors unexecuted USDe bridge requests (`lzReceive`) where `BridgeSwapController` is set as the receiver (bridge and swap requests).
2. The attacker then constructs a swap request consisting of a `swapSimpleMode` request to the KyberSwap Aggregation Router, sets a malicious `caller`, and creates fake `swapSingleSequence` functions that execute and call pending `lzReceive` for USDe, causing USDe to be minted to `BridgeSwapController`.
3. After the swap operation is executed, the minted USDe inside `BridgeSwapController` is set as the caller's received USDe from the "swap" operation.

```
  function _executeSwap(
    IERC20 inputToken, IERC20 outputToken, address router,
  address receiver, uint256 amountSwap, uint256 minAmountOut,
  bytes memory swapData
  ) internal returns (uint256 balanceReceived) {
    // _executeSwap(erc20Receive, usdc, bridgeSwapData.router,
  fromAddress, _amountLD, bridgeSwapData.minAmountOut,
  bridgeSwapData.swapData);
    // Execute the token swap
    if (!whitelistRouter[router]) revert InvalidAddress();
    if (minAmountOut < 1) revert InvalidAmount();

    uint balanceBefore = outputToken.balanceOf(receiver);

    inputToken.safeIncreaseAllowance(router, amountSwap);
    // solhint-disable-next-line avoid-low-level-calls
>>> (bool success, bytes memory res) = router.call(swapData);
    if (!success) {
      string memory reason = LibUtil.getRevertMsg(res);
      revert(string(abi.encodePacked("swap reverted:", reason)));
    }

>>> balanceReceived = outputToken.balanceOf(receiver) - balanceBefore;
```

```
        if (balanceReceived < minAmountOut) revert SwapSlippage();
    }
```

4. The attacker steal the bridged assets.

Recommendations:

Consider also restricting the swap `swapData` and ensuring that it interacts only with a valid swap caller or executor.

Hyperswell: Resolved with @7fa341b6d224...

Zenith: Verified.

[M-2] Not returning excess tokens from swap operation inside `lzCompose` and `swapAndBridge`

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- BridgeSwapController.sol#L86-L119
- ridgeSwapController.sol#L122-L148

Description:

Inside `lzCompose` and `swapAndBridge`, it assume that `_executeSwap` execution will use all the provided input token for the swap, which is not always the case.

```
function lzCompose(
    address _oApp,
    bytes32 _guid,
    bytes calldata _message,
    address _executor,
    bytes calldata _executorData
) external nonReentrant payable {
    // ...

    uint balanceReceived = _executeSwap(erc20Receive, usdc,
    bridgeSwapData.router, fromAddress, _amountLD,
    bridgeSwapData.minAmountOut, bridgeSwapData.swapData);
    emit Swapped(fromAddress, address(usdc), _amountLD, balanceReceived,
    block.timestamp);

    // deposit to Hyper Liquid with permit data

    IERC20(usdc).safePermit(fromAddress, address(this),
    bridgeSwapData.amountPermit, bridgeSwapData.deadline,
    bridgeSwapData.permitData);
    usdc.safeTransferFrom(fromAddress, hyperLiquidDeposit, balanceReceived);
    emit DepositedToHyperLiquid(fromAddress, balanceReceived,
    block.timestamp);
    emit Received(_guid, fromAddress, address(erc20Receive),
    balanceReceived, block.timestamp);
}
```

It is possible that the swap does not fully use all the input tokens, resulting in the excess stay in the `BridgeSwapController`. However, these tokens will remain inside the contract as they are not returned to the user.

Recommendations:

Return the excess or unused input tokens to the user.

Hyperswell: Acknowledged

[M-3] Permit DoS vector on `lzCompose` and `swapAndBridge`

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- BridgeSwapController.sol#L114
- BridgeSwapController.sol#L139
- SafeERC20.sol#L24-L37

Description:

Inside `lzCompose` and `swapAndBridge`, the `safePermit` operation is performed to grant `BridgeSwapController` the signed permit required to execute the necessary transfer operation for the swap.

```
function lzCompose(
    address _oApp,
    bytes32 _guid,
    bytes calldata _message,
    address _executor,
    bytes calldata _executorData
) external nonReentrant payable {
    require(_oApp == oApp, "!oApp");
    require(msg.sender == endpoint, "!endpoint");

    // Extract the composed message from the delivered message using the
MsgCodec
    uint256 _amountLD = OFTComposeMsgCodec.amountLD(_message);
    address fromAddress
= address(uint160(uint256(OFTComposeMsgCodec.composeFrom(_message))));
    emit Bridged(fromAddress, _amountLD, block.timestamp);
    BridgeSwapData memory bridgeSwapData;
    // if fromUser submit this txn, use executor data swap
    if (fromAddress == _executor) {
        bridgeSwapData = abi.decode(_executorData, (BridgeSwapData));
    } else {
        bridgeSwapData
= abi.decode(OFTComposeMsgCodec.composeMsg(_message), (BridgeSwapData));
    }
```

```
        // execute swap
        uint balanceReceived = _executeSwap(erc20Receive, usdc,
    bridgeSwapData.router, fromAddress, _amountLD,
    bridgeSwapData.minAmountOut, bridgeSwapData.swapData);
        emit Swapped(fromAddress, address(usdc), _amountLD, balanceReceived,
    block.timestamp);

        // deposit to Hyper Liquid with permit data
>>>     IERC20(usdc).safePermit(fromAddress, address(this),
    bridgeSwapData.amountPermit, bridgeSwapData.deadline,
    bridgeSwapData.permitData);
        usdc.safeTransferFrom(fromAddress, hyperLiquidDeposit,
    balanceReceived);
        emit DepositedToHyperLiquid(fromAddress, balanceReceived,
    block.timestamp);

        emit Received(_guid, fromAddress, address(erc20Receive),
    balanceReceived, block.timestamp);
    }
```

```
function safePermit(
    IERC20 token,
    address owner,
    address spender,
    uint256 value,
    uint256 deadline,
    bytes memory signature
) internal {
    IUSDCPermit tokenPermit = IUSDCPermit(address(token));
    uint256 nonceBefore = tokenPermit.nonces(owner);
    tokenPermit.permit(owner, spender, value, deadline, signature);
    uint256 nonceAfter = tokenPermit.nonces(owner);
    require(nonceAfter == nonceBefore + 1, "SafeERC20: permit did not
    succeed");
}
```

The permit function can be called by anyone as long as it provides valid signed data. This opens a DoS vector, where an attacker can front-run the `lzCompose` and `swapAndBridge` operations by directly calling permit with valid signed data. Causing the operation to revert.

Recommendations:

Consider modifying `safePermit` by wrapping it with a try-catch block. If the permit has already been executed, simply continue the operation.

Hyperswell: Resolved with @7fa341b6d22...

Zenith: Verified.

4.2 Low Risk

A total of 2 low risk findings were identified.

[L-1] feeUSDCAmount Can Be Set above 10e6

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- BridgeSwapController.sol

Description:

`setFeeConfig` has an upper bound for the `feeUSDCAmount`

```
File:  BridgeSwapController.sol
    function setFeeConfig(
        address _feeReceiver,
        uint256 _feeUSDCAmount
    ) external onlyOwner {
>>    if (feeUSDCAmount > 10e6) revert InvalidAmount();
        feeReceiver = _feeReceiver;
        feeUSDCAmount = _feeUSDCAmount;
    }
```

However, this threshold is not implemented during `initialize`;

```
File:  BridgeSwapController.sol
    function initialize(
        address _erc20Receive,
        address _usdc,
        address _endpoint,
        address _oApp,
        address _hyperLiquidDeposit,
        address _feeReceiver,
        uint256 _feeUSDCAmount
    ) external initializer {
        __Ownable_init();
        __ReentrancyGuard_init();

        erc20Receive = IERC20(_erc20Receive);
        usdc = IERC20(_usdc);
        endpoint = _endpoint;
        oApp = _oApp;
        hyperLiquidDeposit = _hyperLiquidDeposit;
        feeReceiver = _feeReceiver;
 >>   feeUSDCAmount = _feeUSDCAmount;
    }
```

It can be set to a greater value.

Recommendations:

Call `setFeeConfig` at `initialize`.

Hyperswell: Resolved with @7fa341b6d22...

Zenith: Verified.

[L-2] `SwapBridgeData.receiver` is not used as swap receiver inside `lzCompose`

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- BridgeSwapController.sol#L110-L115

Description:

When `lzCompose` is executed, it will call `_executeSwap`, providing `fromAddress` as receiver.

```
    function lzCompose(
        address _oApp,
        bytes32 _guid,
        bytes calldata _message,
        address _executor,
        bytes calldata _executorData
    ) external nonReentrant payable {
        require(_oApp == oApp, "!oApp");
        require(msg.sender == endpoint, "!endpoint");

        // Extract the composed message from the delivered message using the
MsgCodec
        uint256 _amountLD = OFTComposeMsgCodec.amountLD(_message);
        address fromAddress
    = address(uint160(uint256(OFTComposeMsgCodec.composeFrom(_message))));
        emit Bridged(fromAddress, _amountLD, block.timestamp);

        BridgeSwapData memory bridgeSwapData;
        // if fromUser submit this txn, use executor data swap
        if (fromAddress == _executor) {
            bridgeSwapData = abi.decode(_executorData, (BridgeSwapData));
        } else {
            bridgeSwapData
    = abi.decode(OFTComposeMsgCodec.composeMsg(_message), (BridgeSwapData));
        }

        // execute swap
>>>     uint balanceReceived = _executeSwap(erc20Receive, usdc,
    bridgeSwapData.router, fromAddress, _amountLD,
    bridgeSwapData.minAmountOut, bridgeSwapData.swapData);
        emit Swapped(fromAddress, address(usdc), _amountLD, balanceReceived,
    block.timestamp);

        // deposit to Hyper Liquid with permit data
>>>     IERC20(usdc).safePermit(fromAddress, address(this),
    bridgeSwapData.amountPermit, bridgeSwapData.deadline,
    bridgeSwapData.permitData);
>>>     usdc.safeTransferFrom(fromAddress, hyperLiquidDeposit,
    balanceReceived);
        emit DepositedToHyperLiquid(fromAddress, balanceReceived,
    block.timestamp);

        emit Received(_guid, fromAddress, address(erc20Receive),
    balanceReceived, block.timestamp);
    }
```

However, inside `BridgeSwapData`, the receiver field is available. If the user sets the `receiver` and expects the operation to use that address as the swap's receiver, the operation will fail.

```
struct BridgeSwapData {
    uint256 minAmountOut;
    address router;
    address receiver; // @audit - receiver is ignored
    bytes swapData;

    bytes permitData;
    uint256 deadline;
    uint256 amountPermit;
}
```

Recommendations:

Either remove the `receiver` field from `BridgeSwapData` or use it as the swap's receiver address.

Hyperswell: Acknowledged