Debugging

If you find that your bundles aren't landing when you think they should, we have some tips to help you figure out why.

Note: Some examples you see in the code here are based on thelimit order bot tutorial . The same tactics can be used to debug any MEV-Share bundle.

Simulate your bundle

The simplest way to find out what happened with your bundle is to simulate it. MEV-Share nodes offer themev_simBundle json rpc . Typically, MEV-Share Nodes will not return simulation results unless the private transaction being backrun has already landed on-chain.

Using a client

Themev-share-client library has a functionsimulateBundle which executes your bundle in a virtual environment based on the block which the bundle was targeting.

In our project, we can simply call thesimulateBundle function with our original bundle. The block we choose is the block before the first transaction (the one we tried to backrun) lands on-chain, because we want the state as close as possible to when the transaction actually landed; this is the most accurate representation of the state of the blockchain (i.e. prices) when our bundle was supposed to have landed.

// after you call sendBundle(bundleParams) let simResult :

SimBundleLogs

=

await mevshare . simulateBundle (bundleParams) console . log ("simResult" , simResult) Here's a breakdown of the simulation result:

type

SimBundleLogs

=

{ / *True if the simulation executes without error.*/ success :

boolean , / *The error message, if there is an error; otherwise undefined.*/ error ? :

string , / *The block that the simulation derived its state from.*/ stateBlock :

number , / *(profit) / (gasUsed)*/ mevGasPrice : bigint , /*Coinbase profit; the amount paid to the builder after user receives kickback.* / profit : bigint , /*Total ETH paid by searcher to coinbase (gas fees + coinbase transfers)*/ refundableValue : bigint , / *Total gas used by transactions in the bundle.*/ gasUsed : bigint , logs ? :

Array < { /*ETH transaction logs.* / txLogs ? :

Array < LogParams

, / *Logs for nested bundles.* / bundleLogs ? :

Array < SimBundleLogs

, }

, } LogParams are defined in theEthers documentation .

Simulating private Transactions The backend endpoint for simulating bundles (mev_simBundle) only accepts signed transactions, not private transactions (specified by{hash}). The mev-share-client library automatically waits for the transactions in a bundle specified by{hash} to land on-chain by querying for the full signed transactions withprovider.getTransaction before it callsmev_simBundle . If the private transaction(s) in your bundle don't land on-chain, you won't be able to simulate your bundle with them. There are several things you want to look for when debugging your bundles:

'invalid inclusion'

When simulating your bundles, you might see a result like this:

{ success :

false , error :

'invalid inclusion' , stateBlock :

17674041 , mevGasPrice :

0n , profit :

0n , refundableValue :

0n , gasUsed :

0n , logs :

undefined } This happens whenbundleParams.inclusion.block (andbundleParams.inclusion.maxBlock if you set it) target a block (range) that doesn't overlap with the simulation state block.

To remedy this, you can adjust yourbundleParams when you callsimulateBundle to target more blocks — Protect transactions target 25 blocks starting from when they were submitted, so you may want to set your bundleParams like this:

const bundleParams =

{ inclusion :

{ block : currentBlockNumber +

1

} , body :

[{ hash : pendingTxHash } , { tx : backrunSignedTx , canRevert :

false

}] } // ... sendBundle (bundleParams) // ... simulateBundle ({ ... bundleParams , inclusion :

{ ... bundleParams . inclusion , maxBlock : bundleParams . inclusion . block

+

24 , } }) We would just set themaxBlock parameter toinclusion.block + 25 , but we have to subtract 1 sinceinclusion.block wascurrentBlockNumber + 1 when we initially defined it.

You can also simply modify your originalbundleParams to target more blocks, but note that this opens up the possibility of bundles using an outdated price landing on-chain.

'tx failed'

{ success :

false , error :

'tx failed' , stateBlock :

17674503 , mevGasPrice :

0n , profit :

472614000000000n , refundableValue :

472614000000000n , gasUsed :

157538n , logs :

undefined } This happens when one of your transactions reverts. Since oursimulateBundle function waits for private transactions to land on chain before simulating the bundle, we know that in our example, this error means that our backrun transaction failed.

If you see this error when running the bot, it most likely means that the transaction you tried to backrun didn't affect the price

enough to meet our target. This is expected — it's a feature, not a bug!

However, if you want to be sure, a good way to further verify what's happening is to simulate transactions in a local development environment. Toolkits such asfoundry orhardhat give you tools to compile contracts, create local nodes that fork their state from public nodes (e.g. mainnet, goerli), and simulate transactions locally, with stack traces to show you where things went wrong.

Did it pay enough gas?

One of the most common reasons a bundle doesn't land is that it didn't pay enough gas. Of course, you need to pay at least the base fee (the current minimum gas price on Ethereum), but you may also be facing competition. If there are other searchers trying to include the same transaction as you in a bundle, then only one of these bundles can land, because the transaction specified in multiple bundles can only be included on-chain once. The one that lands is determined by who pays the most.

Increasing the priority fee (maxPriorityFeePerGas) on your backrun transaction is a reliable way to improve your chances of inclusion. Note that we also increasemaxFeePerGas by the same amount — this ensures that the builder/validator gets the tip, and it isn't consumed by the base fee. See thisblog post from Blocknative for a detailed explanation of gas fees on Ethereum.

Here's one example of how we can set higher gas fees in our code:

async

function

getSignedBackrunTx (outputAmount : bigint , nonce :

number , gasTip : bigint)

{ const backrunTx =

await uniswapRouterContract . swapExactTokensForTokens . populateTransaction (SELL_TOKEN_AMOUNT , outputAmount ,

[SELL_TOKEN_ADDRESS ,

BUY_TOKEN_ADDRESS] , executorWallet . address ,

9999999999n) const backrunTxFull =

{ ... backrunTx , chainId :

1 , maxFeePerGas :

MAX_GAS_PRICE

*

GWEI

+ gasTip , maxPriorityFeePerGas :

MAX_PRIORITY_FEE

*

GWEI

+ gasTip , gasLimit :

TX_GAS_LIMIT , nonce : nonce } return executorWallet . signTransaction (backrunTxFull) } Then we'd call it like this:

// tip: +5 gwei per gas const tip =

5n

*

GWEI ; const backrunTx =

await

getSignedBackrunTx (outputAmount , nonce , tip) You'll notice that we set our gas prices (maxFeePerGas andmaxPriorityFeePerGas) to constant values. If you don't mind waiting for the gas price on Ethereum to go down, then you can safely ignore these errors. However, if you want to ensure that your trade goes through regardless of the gas price, then you'll have to track the base fee on Ethereum and adjust your transactions' gas prices accordingly.

Here's how we can do that in our project:

async

function

getSignedBackrunTx (outputAmount : bigint , nonce :

number , tip : bigint)

{ const gasFees =

await provider . getFeeData ( ) const backrunTx =

await uniswapRouterContract . swapExactTokensForTokens . populateTransaction (SELL_TOKEN_AMOUNT , outputAmount ,

[SELL_TOKEN_ADDRESS ,

BUY_TOKEN_ADDRESS] , executorWallet . address ,

9999999999n) const backrunTxFull =

{ ... backrunTx , chainId :

1 , maxFeePerGas : gasFees . maxFeePerGas !

+ tip , maxPriorityFeePerGas : gasFees . maxPriorityFeePerGas !

+ tip , gasLimit :

TX_GAS_LIMIT , nonce : nonce } return executorWallet . signTransaction (backrunTxFull) } We start off by fetching the gas fees usingprovider.getFeeData — this queries the RPC provider. You may want to do this in a background thread (once per block) to avoid making redundant calls. But it's OK for demonstration's sake. InbackrunTxFull , we set our gas parameters withgasFees . We use the! suffix operator to coerce the value, which is possibly null, into a bigint. This is safe to do when you're sure that your network has EIP-1559 integrated, which Ethereum mainnet does. The values will only be null on old, deprecated networks.

Did any transactions fail? If so, which one(s)?

It's possible that one of the transactions in your bundle encountered an error and was not able to execute. If this is the case, you'll seefalse in thesuccess parameter of the simulation response, along with an error message, and some data about gas usage and payment.

For example, a simulation result for a bundle including a transaction with its gas price set too low might look like this:

{ success :

false , error :

'max fee per gas less than block base fee: address 0x2326Bd2F29a6004D31344a1FE2329F2C13284f0d, maxFeePerGas: 2000000000 baseFee: 13077974866' , stateBlock :

17674455 , mevGasPrice :

0n , profit :

288942000000000n , refundableValue :

288942000000000n , gasUsed :

192628n , logs :

undefined } IfmevGasPrice is 0, it just means that the transaction didn't use any gas (in this case, because it reverted).

When a simulation succeeds, it looks something like this:

{ success :

true , error :

undefined , stateBlock :

17674443 , mevGasPrice :

9485937386n , profit :

2731874079688143n , refundableValue :

2731874079688143n , gasUsed :

287992n , logs :

[

{ txLogs :

[Array]

} ,

{ txLogs :

[Array]

}

] } This means that your bundlecould have landed in a block, but it isn't a guarantee. If yourmevGasPrice is not high enough, your bundle may not be included in a block. Refer back toDid it pay enough gas? for info on setting your gas price.

There is also one more factor that often comes into play…

Did the Flashbots builder have a chance to include my bundle?

Another common reason a bundle doesn't land is simply because the builders on MEV-Share didn't have an opportunity to build a block for the desired slot. For more details about how builders work, and why this happens, seeSearching Post-Merge . You can check to see if your target block was built by any of the builders connected to MEV-Share by checking the block'sminer parameter. The Flashbots builder address is0xDAFEA492D9c6733ae3d56b7Ed1ADB60692c98Bc5

async

function

isBlockBuiltByFlashbots (blockNum :

number)

{ const flashbotsCoinbase =

"0xDAFEA492D9c6733ae3d56b7Ed1ADB60692c98Bc5" const block =

await provider . getBlock (blockNum) return block ?. miner === flashbotsCoinbase } With this function, look at each block frombundleParams.inclusion.block throughbundleParams.inclusion.maxBlock (if you set it) for a bundle to determine whether it was possible for the bundle to land in that block. Edit this page Last updatedonJan 30, 2024 Previous Sending Bundles Next Introduction