Design

The most important design decision for Pimlico's ERC20Paymaster was permissionlessness — you do not have to interact with any hosted APIs to leverage the paymaster. This has the tradeoff however that token approvals to the paymaster cannot be done during the execution phrase of the UserOperation, and must rather be done during the validation phase or in a previous UserOperation. We felt like this was a good tradeoff to make, because token approvals should only be a one-time friction for users under normal circumstances.

It is important to note that while Pimlico's paymaster is permissionless, it is not decentralized. The owner of the paymaster is able to do a couple very limited actions. These are further explained in the Admin functions section of this page. It was however designed so it can easily be made decentralized in the future.

Oracle

The ERC20 paymaster leverages Chainlink for its price oracles, and uses a combination of the ERC20 Token to USD and Native Token to USD prices to calculate the ERC20 Token to Native Token price. This means that the paymaster is easily deployable on any chain for any token that has Chainlink support, but it also means that if Chainlink is compromised, the price can be arbitrarily different from the real token price for the end user.

Paymaster-specific functions

Paymasters under theERC-4337 specification must implement two functions.

validatePaymasterUserOp allows the paymaster to verify whether the paymaster is willing to pay for the User Operation with custom logic. The paymaster is restricted during this phrase in what it can do. Most importantly, it must comply with the banned opcode and banned external storage access rules of the ERC. For this reason, actions like fetching oracle price state are not possible as they access external storage that is not associated with the account.

postOp is called by the EntryPoint on the paymaster after making the main execution call if anycontext is returned by thevalidatePaymasterUserOp function. Since our paymaster always returns context, this will always be called.

validatePayasterUserOp

Below is the fullvalidatePaymasterUserOp function of our paymaster:

```

///@noticeValidates a paymaster user operation and calculates the required token amount for the transaction. ///@paramuserOpThe user operation data. ///@paramrequiredPreFundThe amount of tokens required for pre-funding. ///@returncontextThe context containing the token amount and user sender address (if applicable). ///@returnvalidationResultA uint256 value indicating the result of the validation (always 0 in this implementation). function_validatePaymasterUserOp(UserOperationcalldatauserOp,bytes32,uint256requiredPreFund) internal override returns(bytesmemorycontext,uint256validationResult) { unchecked{ uint256cachedPrice=previousPrice; require(cachedPrice!=0,"PP-ERC20 : price not set"); uint256length=userOp.paymasterAndData.length-20; // 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffdf is the mask for the last 6 bits 011111 which mean length should be 100000(32) || 000000(0) require( length&0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffdf==0, "PP-ERC20 : invalid data length" ); //NOTE: we assumed that nativeAsset's decimals is 18, if there is any nativeAsset with different decimals, need to change the 1e18 to the correct decimals uint256tokenAmount=(requiredPreFund+ (REFUND_POSTOP_COST)*userOp.maxFeePerGas)*priceMarkup *cachedPrice/(1e18*priceDenominator); if(length==32) { require( tokenAmount<=uint256(bytes32(userOp.paymasterAndData[20:52])),"PP-ERC20 : token amount too high" ); } SafeTransferLib.safeTransferFrom(address(token), userOp.sender,address(this), tokenAmount); context=abi.encodePacked(tokenAmount, userOp.sender); // No return here since validationData == 0 and we have context saved in memory validationResult=0; } }

```

The code above does the following, in order:

1. Fetches the last price — (set either by a manual price update or a previous UserOperation's postOp call)
2. Validate paymasterAndData length — the paymaster makes sure the paymasterAndData consists only of either: 1. the paymaster address itself (with no other data) or 2. the paymaster address concatenated with a uint256 number that represents the maximum amount of ERC20 tokens the wallet is willing to pay
3. Calculate the token amount — use the last price, the requiredPreFund amount, and the gasPrice to calculate the amount of ERC20 tokens that will be taken from the sender
4. Take the tokens and return the context, implicitly agreeing to sponsor the UserOperation — (if the paymaster does not revert, it by default agrees to sponsoring the UserOperation).

Note that we do not fetch the live oracle price during the validation step. Fetching the price at this stage would break the ERC-4337 external storage access rules. Instead, we delay doing that until thepostOp function. This way, we do a pay-it-forward system, where each UserOperation updates the price for the UserOperation coming after it.

As a safety measure in case there is a significant price change between two UserOperations, Pimlico or any third party is able to manually update the price and bring it into the local contract storage by calling theupdatePrice() function.

We will be maintaining automated off-chain watchers that will automatically call theupdatePrice() function on the supported paymasters in case the price deviates from the last price beyond a certain threshold.

postOp

Below is the fullpostOp function of our paymaster:

```

///@noticePerforms post-operation tasks, such as updating the token price and refunding excess tokens. ///@devThis function is called after a user operation has been executed or reverted. ///@parammodeThe post-operation mode (either successful or reverted). ///@paramcontextThe context containing the token amount and user sender address. ///@paramactualGasCostThe actual gas cost of the transaction.
function_postOp(PostOpModemode,bytescalldatacontext,uint256actualGasCost)internaloverride{
if(mode==PostOpMode.postOpReverted) { return;// Do nothing here to not revert the whole bundle and harm reputation }
unchecked{ uint192tokenPrice=fetchPrice(tokenOracle); uint192nativeAsset=fetchPrice(nativeAssetOracle);
uint256cachedPrice=previousPrice; uint192price=nativeAsset*uint192(tokenDecimals)/tokenPrice;*
*uint256cachedUpdateThreshold=priceUpdateThreshold; if(*
*uint256(price)*priceDenominator/cachedPrice>priceDenominator+cachedUpdateThreshold
||uint256(price)*priceDenominator/cachedPriceactualTokenNeeded) { // If the initially provided token amount is greater than the actual amount needed, refund the difference SafeTransferLib.safeTransfer( address(token), address(bytes20(context[32:52])), uint256(bytes32(context[0:32]))-actualTokenNeeded ); }// If the token amount is not greater than the actual amount needed, no refund occurs

emitUserOperationSponsored(address(bytes20(context[32:52])), actualTokenNeeded, actualGasCost); } }

```

The code above does the following, in order:

1. Checks whether thepostOp
2. is being called by the outerpostOp
3. context, i.e. if we have already reverted thepostOp
4. once. If so, it simply returns and doesn't revert the whole UserOperation.
5. Fetch the live prices for the ERC20 Token / USD and NativeToken / USD prices, and use it to calculate the latest ERC20 Token / NativeToken price.
6. If If the price has deviated more than thecachedUpdateThreshold
7. , then this new price is stored in thepreviousPrice
8. storage variable to be used by the next UserOperation that uses this paymaster. Otherwise, if the price doesn't deviate beyond the threshold, don't put the new price in storage. This is to save an SSTORE whenever possible.
9. Refund the excess tokens taken by the paymaster during validation based on the amount of gas actually used during execution.
10. Emit aUserOperationSponsored
11. event so paymaster usage can be easily indexed off-chain.

Admin functions

The owner of the paymaster does not have access to user balances and has been designed with hardcoded limitations to what it can change. No admin upgradeable proxies are used.

There are only two admin-controlled functions in the paymaster.

withdrawToken

Below is the fullwithdrawToken function of our paymaster:

```

///@noticeAllows the contract owner to withdraw a specified amount of tokens from the contract. ///@paramtoThe address to transfer the tokens to. ///@paramamountThe amount of tokens to transfer.
functionwithdrawToken(addressto,uint256amount)externalonlyOwner{ SafeTransferLib.safeTransfer(address(token), to, amount); }
```

```

This function enables the owner to withdraw the ERC20 tokens that have been accumulated by the paymaster to a specified address. We will frequently use this to withdraw the ERC20 tokens received by the paymaster in order to swap them back to native tokens, which we can deposit back to the paymaster.

## updateConfig

Below is the fullupdateConfig function of our paymaster:

```

///@noticeUpdates the price markup and price update threshold configurations. ///@param_priceMarkupThe new price markup percentage (1e6 = 100%). ///@param_updateThresholdThe new price update threshold percentage (1e6 = 100%). functionupdateConfig(uint32_priceMarkup,uint32_updateThreshold)externalonlyOwner{ require(_priceMarkup<=120e4,"PP-ERC20 : price markup too high"); require(_priceMarkup>=1e6,"PP-ERC20 : price markeup too low"); require(_updateThreshold<=1e6,"PP-ERC20 : update threshold too high"); priceMarkup=_priceMarkup; priceUpdateThreshold=_updateThreshold; emitConfigUpdated(_priceMarkup, _updateThreshold); }

```

TheupdateConfig function is able to make the following updates to the paymaster contract:

1. Changing thepriceMarkup
2. to any number between 0% and 20% on top of the price of the ERC20 token price. The larger the number, the larger the fee the owner makes per UserOperation
3. Changing theupdateThreshold
4. , which represents at what percentage price difference from thepreviousPrice
5. thepostOp
6. function will store the new price in storage. The smaller this number, the more regularly a UserOperation will pay-it-forward and update the price for the next UserOperation, but simultaneously the more often it will make the caller incur the added gas cost of the SSTORE call.

ThepriceMarkup serves to compensate the owner for maintaining the infrastructure (like the frequent manual price updates), as well as the exchange rate movement risk and slippage risk incurred when swapping back the tokens to the native token. If you feel thepriceMarkup used by our canonical PimlicoERC20Paymaster (of which we are the owners) is unfair, we provide an easy way for you to deploy your own version, which can be found in the repository.

Assuming an adversarial scenario where the owner of the paymaster becomes compromised, the worst-case-scenario is that the hacker sets thepriceMarkup to the maximum of 20% (meaning from that point on users will pay a premium of 20% on top of the price of the ERC20 tokens), or all of the native tokens previously deposited by the owner are drained (meaning the paymaster becomes unusable as it has no native tokens to sponsor with). They cannot get any direct access to user balances, and they are not able to increase the markup to an arbitrarily high number.
```