

Withdrawal Flow

In Optimism terminology, a withdrawal is a transaction sent from L2 (OP Mainnet, OP Sepolia etc.) to L1 (Ethereum mainnet, Sepolia, etc.).

Withdrawals require the user to submit three transactions:

1. Withdrawal initiating transaction
2. , which the user submits on L2.
3. Withdrawal proving transaction
4. , which the user submits on L1 to prove that the withdrawal is legitimate (based on a merkle patricia trie root that commits to the state of the `L2ToL1MessagePasser`
5. 's storage on L2)
6. Withdrawal finalizing transaction
7. , which the user submits on L1 after the fault challenge period has passed, to actually run the transaction on L1.

You can see an example of how to do this [in the bridging tutorials](#).

Withdrawal initiating transaction

1. On L2 somebody (either an externally owned account (EOA) directly or a contract acting on behalf of an EOA) calls the [sendMessage \(opens in a new tab\)](#)
2. function of [L2CrossDomainMessenger \(opens in a new tab\)](#)
3. .
4. This function accepts three parameters:
5.
 - `_target`
6.
 - , target address on L1.
7.
 - `_message`
8.
 - , the L1 transaction's calldata, formatted as per the [ABI \(opens in a new tab\)](#)
9.
 - of the target address.
10.
 - `_minGasLimit`
11.
 - , The minimum amount of gas that the withdrawal finalizing transaction can provide to the withdrawal transaction. This is enforced by the `SafeCall`
12.
 - library, and if the minimum amount of gas cannot be met at the time of the external call from the `OptimismPortal`
13.
 - `->L1CrossDomainMessenger`
14.
 - , the finalization transaction will revert to allow for re-attempting with a higher gas limit. In order to account for the gas consumed in the `L1CrossDomainMessenger.relayMessage`
15.
 - function's execution, extra gas will be added on top of the `_minGasLimit`
16.
 - value by the `CrossDomainMessenger.baseGas`
17.
 - function when `sendMessage`
18.
 - is called on L2.
19. `sendMessage`
20. is a generic function that is used in both cross domain messengers.
21. It calls [sendMessage \(opens in a new tab\)](#)
22. , which is specific to [L2CrossDomainMessenger \(opens in a new tab\)](#)
23. .
24. `_sendMessage`
25. calls [initiateWithdrawal \(opens in a new tab\)](#)
26. on [L2ToL1MessagePasser \(opens in a new tab\)](#)
27. .
28. This function calculates the hash of the raw withdrawal fields.
29. It then [marks that hash as a sent message in `sentMessages` \(opens in a new tab\)](#)
30. and [emits the fields with the hash in a `MessagePassed` event \(opens in a new tab\)](#)

31. .

32. The raw withdrawal fields are:

33.

- nonce

34.

- - A single use value to prevent two otherwise identical withdrawals from hashing to the same value

35.

- sender

36.

- - The L2 address that initiated the transfer, typically [L2CrossDomainMessenger \(opens in a new tab\)](#)

37.

- target

38.

- - The L1 target address

39.

- value

40.

- - The amount of WEI transferred by this transaction

41.

- gasLimit

42.

- - Gas limit for the transaction, the system guarantees that at least this amount of gas will be available to the transaction on L1.

43.

- Note that if the gas limit is not enough, or if the L1 finalizing transaction does not have enough gas to provide that gas limit, the finalizing transaction returns a failure, it does not revert.

44.

- data

45.

- - The calldata for the withdrawal transaction

46. Whenop-proposer

47. [proposes a new output\(opens in a new tab\)](#)

48. , the output proposal includes [the output root\(opens in a new tab\)](#)

49. , provided as part of the block byop-node

50. .

51. This new output root commits to the state of thesentMessages

52. mapping in theL2ToL1MessagePasser

53. contract's storage on L2, and it can be used to prove the presence of a pending withdrawal within it.

Withdrawal proving transaction

Once an output root that includes theMessagePassed event is published to L1, the next step is to prove that the message hash really is in L2. Typically this is done[by the SDK\(opens in a new tab\)](#) .

Offchain processing

1. A user calls the SDK's[CrossDomainMessenger.proveMessage\(\) \(opens in a new tab\)](#)
2. with the hash of the L2 message.
3. This function calls[CrossDomainMessenger.populateTransaction.proveMessage\(\) \(opens in a new tab\)](#)
4. .
5. To get from the L2 transaction hash to the raw withdrawal fields, the SDK uses[toLowLevelMessage \(opens in a new tab\)](#)
6. .
7. It gets them from theMessagePassed
8. event in the receipt.
9. To get the proof, the SDK uses[getBedrockMessageProof \(opens in a new tab\)](#)
10. .
11. Finally, the SDK calls[OptimismPortal.proveWithdrawalTransaction\(\) \(opens in a new tab\)](#)
12. on L1.

Onchain processing

[OptimismPortal.proveWithdrawalTransaction\(\)](#) (opens in a new tab) runs a few sanity checks. Then it verifies that in `L2ToL1MessagePasser.sentMessages` on L2 the hash for the withdrawal is turned on, and that this proof has not been submitted before. If everything checks out, it writes the output root, the timestamp, and the L2 output index to which it applies `improveWithdrawals` and emits an event.

The next step is to wait the fault challenge period, to ensure that the L2 output root used in the proof is legitimate, and that the proof itself is legitimate and not a hack.

Withdrawal finalizing transaction

Finally, once the fault challenge period passes, the withdrawal can be finalized and executed on L1.

Offchain processing

1. A user calls the SDK's [CrossDomainMessenger.finalizeMessage\(\)](#) (opens in a new tab)
2. with the hash of the L1 message.
3. This function calls [CrossDomainMessenger.populateTransaction.finalizeMessage\(\)](#) (opens in a new tab)
4. .
5. To get from the L2 transaction hash to the raw withdrawal fields, the SDK uses [toLowLevelMessage](#) (opens in a new tab)
6. .
7. It gets them from the `MessagePassed`
8. event in the receipt.
9. Finally, the SDK calls [OptimismPortal.finalizeWithdrawalTransaction\(\)](#) (opens in a new tab)
10. on L1.

Onchain processing

1. [OptimismPortal.finalizeWithdrawalTransaction\(\)](#) (opens in a new tab)
2. runs several checks. The interesting ones are:
3.
 - [Verify the proof has already been submitted](#)(opens in a new tab)
4.
 - .
5.
 - [Verify the proof has been submitted long enough ago that the fault challenge period has already passed](#)(opens in a new tab)
6.
 - .
7.
 - [Verify that the proof applies to the current output root for that block \(the output root for a block can be changed by the fault challenge process\)](#)(opens in a new tab)
8.
 - .
9.
 - [Verify that the current output root for that block was proposed long enough ago that the fault challenge period has passed](#)(opens in a new tab)
10.
 - .
11.
 - [Verify that the transaction has not been finalized before to prevent replay attacks](#)(opens in a new tab)
12.
 - .
13. If any of these checks fail, the transaction reverts.
14. Mark the withdrawal as finalized in `finalizedWithdrawals`
15. .
16. Run the actual withdrawal transaction (call the `target`
17. contract with the `calldata` in `data`
18.).
19. Emit a [WithdrawalFinalized](#) (opens in a new tab)
20. event.

[Transaction Flow Overview](#)