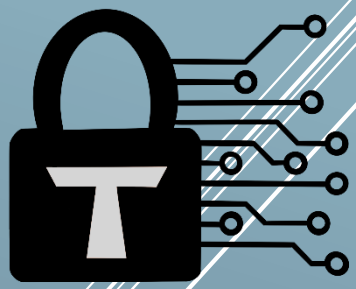
Trust Security

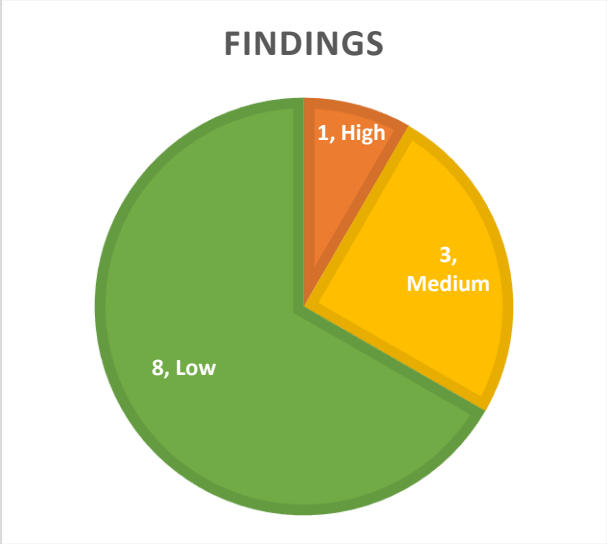


Smart Contract Audit

Optimism Bedrock upgrade

11/01/2024

Executive summary

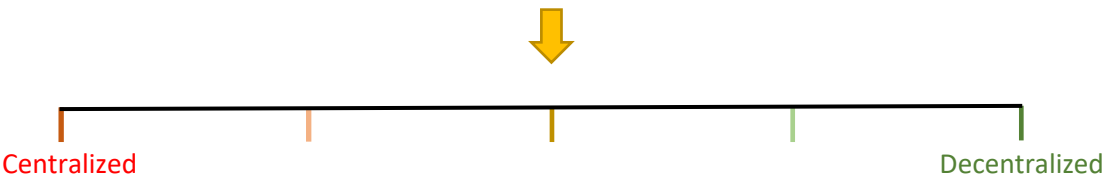


Category	Rollups
Audited file count	8
Auditor	Trust Gjaldon
Time period	11/12-25/12

Findings

Severity	Total	Fixed	Acknowledged
High	1	1	-
Medium	3	-	3
Low	8	1	7

Centralization score



Signature

Digitally signed by Trust
Date: 2024.01.11 15:34:54 +02'00'

Document properties

Versioning

Version	Date	Description
0.1	25/12/23	Client report
0.2	11/01/23	Mitigation review

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

The scope is all changes made in the contracts below, since the Sherlock contest commit.

- SuperchainConfig.sol
- L1CrossDomainMessenger.sol
- L1ERC721Bridge.sol
- L1StandardBridge.sol
- OptimismPortal.sol
- CrossDomainMessenger.sol
- ERC721Bridge.sol
- StandardBridge.sol

Specifically, the following mechanisms have been given special attention:

- Mitigation of contest's reported gas issues and additional gas-related flaws
- Examination of storage slots and the impact of upgrades on their safety
- Changes in reentrancy protection of cross-chain messaging
- Integration of the new SuperchainConfig contract, including the pausing functionality.
- Upgrade procedure of Bedrock contracts and potential side-effects

Repository details

- **Repository URL:** https://github.com/ethereum-optimism/optimism
- **Commit hash:** d1651bb22645ebd41ac4bb2ab4786f9a56fc1003
- **Mitigation review commit hash:** 81b56fee40f96c798174115c375f41c3d2ff9d40

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is a leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is serving as a Code4rena judge.

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

Gjaldon transitioned to Web3 after 10+ years working as a Web2 engineer. His first foray into Web3 was achieving first place in a smart contracts hackathon and then later securing a project grant to write a contract for Compound III. He shifted to Web3 security and in 3 months achieved top 2-5 in two contests with unique High and Medium findings and joined exclusive top-tier auditing firms.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	The code is modularized well to reduce complexity.
Documentation	**Excellent**	Project is mostly very well documented.
Best practices	**Excellent**	Project consistently adheres to industry standards.
Centralization risks	**Moderate**	A compromised multisig account can cause irrecoverable damage.

Findings

High severity findings

TRST-H-1 Anyone can execute a withdrawal twice by abusing the upgrade procedure

- **Category:** Reentrancy attacks, frontrunning attacks, initialization flaws
- **Source:** CrossDomainMessenger.sol
- **Status:** Fixed

Description

In Optimism architecture, every withdrawal can be performed once. The recommended way is to use the CrossDomainMessenger which protects against any delivery issues by storing failed messages, so they may be replayed. In *CrossDomainMessenger::relayMessage()*, the following code serves as a reentrancy guard:

```
        if (
            !SafeCall.hasMinGas(_minGasLimit, RELAY_RESERVED_GAS +
RELAY_GAS_CHECK_BUFFER)
                || xDomainMsgSender != Constants.DEFAULT_L2_SENDER
        ) {
```

It checks that **xDomainMsgSender** is not the default L2 sender. If it is not, then it will proceed to fail the message. Note that because **successfulMessages[versionedHash]** is checked before the external call and set to **true** after it, the code pattern is otherwise susceptible to reentrancy attacks.

The *CrossDomainMessenger* also has some initialization code that is run every time it is upgraded. This initializer resets the **xDomainMsgSender** and is what opens the exploit enabling an attacker to steal funds via reentrancy.

To perform the exploit, the following steps must be taken by the attacker:

1. Wait for a signed upgrade transaction from Optimism. This was intended to be delivered by the public mempool.
2. Once the signed upgrade transaction is available, front-run by running *it inside a withdrawal transaction*.
3. The attacker's withdrawal payload will call their own contract which would run the upgrade transaction and then re-enter *relayMessage()* with their own same withdrawal message.
4. The reentrancy will succeed since *xDomainMsgSender* has been reset by the upgrade.

The attacker must also satisfy the following with their withdrawal message:

1. It must be a failed withdrawal so they can re-enter relayMessage(). A fresh withdrawal cannot reenter as it checks **failedMessages[versionedHash]** is **true**.
2. It must have a value set. The amount for value is the amount that they will able to drain from the contract.

The funds at risk for this exploit is the ETH balance of the L1CrossDomainMessenger contract. It will have ETH from all the failed messages with attached msg.value that are pending replay.

Recommended mitigation

The following mitigations will address the issue:

- When setting the state of **xDomainMsgSender** in *__CrossDomainMessenger_init()*, verify it is previously **zero** (meaning it is freshly deployed, not upgraded).
- Add another check that **successfulMessages** for the message is still false after the external call. **successfulMessages** only becomes true at this point if the external call has successfully re-entered *relayMessage()*.
- Set **successfulMessages[versionedHash] = true** before the external call similar to the replay protection that exists in *OptimismPortal*.

Team response

Fixed.

Mitigation Review

The fix implements two of the recommended methods of mitigating the issue, which is sufficient to address the reentrancy issue. The changes are the following:

- On initialize, **xDomainMsgSender** is no longer reset to zero when it has already been previously set.
- An assertion is added to ensure that **successfulMessages** is still false after the external call.

The fix also includes an added protective measure against reentrancy for *OptimismPortal*.

Medium severity findings

TRST-M-1 Anyone can make a victim's withdrawal TX revert, delaying withdrawals and making them more expensive

- **Category:** Griefing attacks, reentrancy attacks
- **Source:** CrossDomainMessenger.sol
- **Status:** Acknowledged

Description

An attacker can grief other users by forcing their withdrawals to fail for their initial submissions. This leads to the victims having to manually replay their own withdrawals and cover the gas costs for these transactions on top of the reverting transaction.

Below are the steps to execute the griefing:

1. Attacker sends a withdrawal message that is run in the *L1CrossDomainMessenger*.
2. The withdrawal message calls the attacker's contract which then calls *OptimismPortal::finalizeWithdrawalTransaction()* to finalize the withdrawal of the target user.

3. When the target user's withdrawal is relayed, it reenters *relayMessage* which fails the message due to the reentrancy guard.
4. Since the call to *L1CrossDomainMessenger::relayMessage* did not revert, the withdrawal is considered finalized in the *OptimismPortal*.

Recommended mitigation

Consider refactoring to revert in a reentrant flow, while adding safeguards in place not to brick transactions. Note that due to the fragile nature of the code, there may be dangerous side effects.

Team response

Acknowledged. This will be tracked and addressed in future upgrades.

TRST-M-2 Insufficient calldata gas stipend could make initial delivery fail

- **Category:** Gas-related issues
- **Source:** CrossDomainMessenger.sol
- **Status:** Acknowledged

Description

When sending deposit transactions, a user pays for L2 gas costs in L1. The total gas user pays for is computed via *baseGas()*:

```
function baseGas(bytes calldata _message, uint32 _minGasLimit) public pure returns
(uint64) {
    return
    // Constant overhead
    RELAY_CONSTANT_OVERHEAD // 200_000
    // Calldata overhead
    + (uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) // messageLength * 16
    // Dynamic overhead (EIP-150)
    + ((_minGasLimit * MIN_GAS_DYNAMIC_OVERHEAD_NUMERATOR) /
MIN_GAS_DYNAMIC_OVERHEAD_DENOMINATOR) // (minGasLimit * 64) / 63
    // Gas reserved for the worst-case cost of 3/5 of the `CALL` opcode's dynamic gas
    // factors. (Conservative)
    + RELAY_CALL_OVERHEAD // 40_000
    // Relay reserved gas (to ensure execution of `relayMessage` completes after the
    // subcontext finishes executing) (Conservative)
    // @audit this reservedGas and gasCheckBuffer may not be needed here
    + RELAY_RESERVED_GAS // 40_000
    // Gas reserved for the execution between the `hasMinGas` check and the `CALL`
    // opcode. (Conservative)
    + RELAY_GAS_CHECK_BUFFER; // 5_000
}
```

Note that in the above calculation, the calldata overhead accounts for only one external call. However, when the deposit transaction is executed in L2, this transaction involves at least 2

external calls. The first external call is the call to *relayMessage()* and the second external call is the call to the target address.

```
    xDomainMsgSender = _sender;
    bool success = SafeCall.call(_target, gasleft() - RELAY_RESERVED_GAS, _value,
_message);
    xDomainMsgSender = Constants.DEFAULT_L2_SENDER;
```

As far as Optimism is concerned, **minGasLimit** should be the amount available when running the first instruction in the user's contract, ignoring calldata costs. This means that *baseGas()* is incorrect and its calldata overhead is insufficient.

Note that since one copy of calldata costs is accounted for, it is extremely unlikely for the *relayMessage()* call to revert before storing the delivery status. It will presumably fail the *SafeCall.hasMinGas()* check and store the failure immediately. Therefore, impact is limited to delay and extra gas spending of the withdrawal process.

Recommended mitigation

The correct calculation for calldata overhead costs is:

```
(uint64(_message.length) * 2 + 5) * MIN_GAS_CALLDATA_OVERHEAD
```

This takes into consideration the calldata spending of *relayMessage* (The arbitrary payload as well as fixed parameters) as well as the target contract calldata (only payload).

Team response

Acknowledged. This will be tracked and addressed in future upgrades.

TRST-M-3 Messages of over 50k bytes can be permanently lost due to unaccounted gas costs

- **Category:** Gas-related issues
- **Source:** CrossDomainMessenger.sol
- **Status:** Acknowledged

Description

The *baseGas()* calculation is intended to account for all the overhead costs for executing a deposit or withdrawal transaction.

```
function baseGas(bytes calldata _message, uint32 _minGasLimit) public pure returns
(uint64) {
    return
    // Constant overhead
    RELAY_CONSTANT_OVERHEAD // 200_000
    // Calldata overhead
    + (uint64(_message.length) * MIN_GAS_CALLDATA_OVERHEAD) // messageLength * 16
    // Dynamic overhead (EIP-150)
    + ((_minGasLimit * MIN_GAS_DYNAMIC_OVERHEAD_NUMERATOR) /
MIN_GAS_DYNAMIC_OVERHEAD_DENOMINATOR) // (minGasLimit * 64) / 63
    // Gas reserved for the worst-case cost of 3/5 of the `CALL` opcode's dynamic gas
    // factors. (Conservative)
    + RELAY_CALL_OVERHEAD // 40_000
    // Relay reserved gas (to ensure execution of `relayMessage` completes after the
    // subcontext finishes executing) (Conservative)
    // @audit this reservedGas and gasCheckBuffer may not be needed here
    + RELAY_RESERVED_GAS // 40_000
    // Gas reserved for the execution between the `hasMinGas` check and the `CALL`
    // opcode. (Conservative)
    + RELAY_GAS_CHECK_BUFFER; // 5_000
}
```

The *RELAY_CONSTANT_OVERHEAD* (200K gas units) should cover all the costs in *relayMessage()* up to the *hasMinGas()* check. It is critical that *relayMessage()* has enough gas to at least store the transaction hash in the **failedMessages** mapping and return. Otherwise, in the case of withdrawals, the transaction could be permanently lost.

There are dynamic gas costs related to hashing that have not been sufficiently accounted for in *baseGas()*. Listed below are the non-negligible operations in terms of gas, in *relayMessage()*:

```
require(paused() == false, "CrossDomainMessenger: paused");
```

- 2 cold SLOADs and 1 cold address CALL - ~7000 gas
- 1 or 2 hashing rounds – gas cost is dependent on the length of the message being hashed

```
assert(!failedMessages[versionedHash])
```

- Cold SLOAD - ~2100 gas

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

- Cold SLOAD - ~2100 gas

```
failedMessages[versionedHash] = true;
```

- Warm zero to non-zero SSTORE - ~20,000 gas

```
emit FailedRelayedMessage(versionedHash);
```

- LOG1 - ~750 gas

Total gas costs - ~32,000 + hash costs

The available gas from *baseGas()* is:

200k + 40k + 40k + 5k + minGasLimit * 64/63 = ~285k + minGasLimit

Note that the calldata overhead component of *baseGas()* is spent on the calldata cost of the call from *OptimismPortal* to *CrossDomainMessenger*, so it is not included in the calculation above.

To simplify the computations, a **minGasLimit** of 0 will be assumed, since a user could specify it as such and expect the TX to always be replayable through the CrossDomainMessenger security guarantees. Given the above costs and gas provided by the user, the gas cost of the hashing operations must equal or exceed ~253,000 gas for *relayMessage()* to always revert due to an OOG error.

To simulate the hashing functions, the following code can be used:

```
function test_demo() public {
    uint256[] memory input = new uint256[](1563);
    for (uint256 i; i < input.length; i++) {
        input[i] = type(uint256).max;
    }
    bytes memory data = abi.encodePacked(input);
    console.log("Data length: ", data.length);
    (bool success,) = address(this).call(abi.encodeWithSelector(this.encodeCrossDomainMessageV1.selector, data));
}

function encodeCrossDomainMessageV1(bytes memory _data) public {
    uint size;
    uint offset;
    assembly { offset := _data }
    size = offset + _data.length;
    console.log("Starting memory size: ", size);
    uint256 startingGas = gasleft();
    console.log("Starting gas: ", startingGas);
    bytes memory b = abi.encodeWithSignature("aaaa",_data);
    assembly { offset := b }
    size = offset + b.length;
    bytes32 kec = keccak256(b);
    bytes memory c = abi.encodeWithSignature("aaaa",_data);
    assembly { offset := c }
    size = offset + c.length;
    kec = keccak256(c);
    console.log("Total gas used: ", startingGas - gasleft());
    console.log("Completed memory size: ", size);
}
```

In the above simulation (a legacy transaction), the input provided has a size of 50,016 bytes (1563 32-byte elements in the array). The operations related to the hashing function end up consuming a total of 278,012 gas units. Given input data of 50,000 bytes, that is enough for *relayMessage()* to permanently fail for a withdrawal with 0 **minGasLimit**. Tweaking the input to a size of 120,000 bytes, which is the maximum data size allowed in

OptimismPortal::depositTransaction, the operations would consume gas totaling 791,575 units.

There are two operations responsible for the dynamic gas costs of the hashing functions. Below is a simplified version of the hashing function:

```
return keccak256(
    abi.encodeWithSignature(
        "relayMessage(address,address,bytes,uint256)",
        _target,
        _sender,
        _data,
        _nonce
    )
);
```

Keccak256 is largely the SHA3 opcode which has a gas cost that grows linearly based on the size of the message being hashed. This is the actual hashing operation.

However, responsible for the larger chunk of gas usage is *abi.encodeWithSignature()* since its output is *bytes* data that is always stored in memory. The way Solidity works when working with dynamic data is that it always stores (MSTORE) new data in an unused offset to avoid data corruption. This leads to memory expansion which is a very costly operation that grows quadratically based on the size of expanded memory.

With the hashing function, memory is expanded by the size of the *_message* parameter in *relayMessage()*. When the withdrawal is a legacy withdrawal, the memory expansion is twice the size of the *_message* since two hashing functions are executed. Note that a 100k v1 payload would not cost the same as a 50k v0 payload, as although the loop length and memory expansion size are the same, the memory size starting point is higher for a v1 payload, making the quadratic cost higher.

Recommended mitigation

Limit the *sendMessage()* payload size using an upper bound calculated from simulation of **v1** and **v0** transactions, making sure to leave margin for inaccuracies.

Team response

Acknowledged. This will be tracked and addressed in future upgrades.

Low severity findings

TRST-L-1 User is forced to overpay for deposit gas due to calldata gas calculation

- **Category:** Gas-related issues
- **Source:** OptimismPortal.sol
- **Status:** Acknowledged

Description

Gas consumption for non-zero bytes in data passed in transactions is 16 while it is 4 for zero bytes. However, OptimismPortal::minimumGasLimit does not apply this distinction.

```
    function minimumGasLimit(uint64 _byteCount) public pure returns (uint64) {
        return _byteCount * 16 + 21000;
    }
```

This leads to users overpaying for gas.

Recommended mitigation

The following computation can instead be used for minimum gas limit:

```
    function minimumGasLimit(bytes memory _data) public pure returns (uint256) {
        uint256 intrinsicGas = 21000;
        uint256 nonzeroGas = 16;
        uint256 zeroGas = 4;
        uint256 gas = intrinsicGas;
        if (_data.length > 0) {
            uint256 nz = 0;
            for (uint256 i = 0; i < _data.length; i++) {
                if (_data[i] != 0) {
                    nz++;
                }
            }
            gas += nz * nonzeroGas + (_data.length - nz) * zeroGas;
        }
        return gas;
    }
```

Note that the above calculation is also used by Scroll. However, it would be fair to consider the heavier gas costs of this loop and opt out of its use.

Team response

Acknowledged. This will be tracked and addressed in future upgrades.

TRST-L-2 The metering logic in OptimismPortal will be incorrect immediately after the upgrade

- **Category:** Initialization flaws
- **Source:** OptimismPortal.sol
- **Status:** Fixed

Description

During upgrades, *OptimismPortal::initialize* is called:

```
    function initialize(SuperchainConfig _superchainConfig) public initializer {
        l2Sender = Constants.DEFAULT_L2_SENDER;
        superchainConfig = _superchainConfig;
        __ResourceMetering_init();
    }
```

The initializer in *ResourceMetering* does the following:

```
    function __ResourceMetering_init() internal onlyInitializing {
        params = ResourceParams({
            prevBaseFee: 1 gwei,
            prevBoughtGas: 0,
            prevBlockNum: uint64(block.number)
        });
    }
```

Note that the **prevBoughtGas** is set to 0. It is used for recording all the gas that has been previously bought within the current block and is used to ensure that the *maxResourceLimit* is not exceeded.

```
    params.prevBoughtGas += _amount;
    require(
        int256(uint256(params.prevBoughtGas)) <=
int256(uint256(config.maxResourceLimit)),
        "ResourceMetering: cannot buy more gas than available gas limit"
    );
```

In effect, upgrades reset **prevBoughtGas** and allow users to go beyond the **maxResourceLimit**.

Recommended mitigation

Remove the resetting of the gas market in *ResourceMetering*'s initializer.

Team response

Fixed.

Mitigation Review

The **ResourceParams** in *ResourceMetering's* initializer is no longer reset when it has already been previously set.

TRST-L-3 A storage slot is accidentally skipped

- **Category:** Storage collision issues
- **Source:** CrossDomainMessenger.sol
- **Status:** Acknowledged

Description

A previous version of *CrossDomainMessenger* had a **reentrancyLocks** state variable. This older version had the following gap size:

```
/**
 * @notice A mapping of hashes to reentrancy locks.
 */
 mapping(bytes32 => bool) internal reentrancyLocks;
 /**
  * @notice Reserve extra slots in the storage layout for future upgrades.
  *         A gap size of 41 was chosen here, so that the first slot used in a child
contract
  *         would be a multiple of 50.
  */
 uint256[41] private __gap;
```

The current version of *CrossDomainMessenger* no longer has the **reentrancyLocks** state variable. However, its gap size has increased to 44.

```
    /// @notice Reserve extra slots in the storage layout for future upgrades.
    ///         A gap size of 44 was chosen here, so that the first slot used in a
child contract
    ///         would be 1 plus a multiple of 50.
    uint256[44] private __gap;
```

The latest *CrossDomainMessenger* is now using 1 more storage slot than it should.

Recommended mitigation

The latest *CrossDomainMessenger* should have a gap size of 43 so that its total storage slots would be a multiple of 50. With a gap size of 43, its total storage slots would be 250, which is a multiple of 50.

Team response

Acknowledged. This will be tracked and addressed in future upgrades.

TRST-L-4 Upgrades are resetting the state of unused storage slots

- **Category:** Storage collision issues
- **Source:** l1.go
- **Status:** Acknowledged

Description

L1 upgrades are a 2-step process which involves replacing the implementation contract with the *StorageSetter* and directly manipulating data in storage slots as the first step. This is necessary to reset the *_initialized* slot to be able to re-initialize the Proxy contracts.

There is an issue in the upgrade logic for some of the L1 contracts since they are resetting state for storage slots that are unused. The table below details the issues:

Contract	Issue
OptimismMintableERC20Factory	Does not use storage so no need to use StorageSetter
OptimismPortal	Slot 53 is used by `spacer_53_0_1` and `superchainConfig`. It is being cleared here. These are noops since slots 54 and 55 are unused
SystemConfig	Slot 106 is unused and all the other EIP-1967 slots no longer exist in SystemConfig. The following can be removed
L2OutputOracle	Only 4 slots are used. These are noops since slots and 4 and 5 are unused. Note that immutables do not use storage and are hardcoded into the contract during deployment
L1StandardBridge	Slot 3 belongs to `__gap` and this is a noop. It can be removed. `messenger` is an immutable and can not be modified.
L1CrossDomainMessenger	This can be removed since it's only modifying data in the `__gap`

Relevant references:

- OptimismMintableERC20Factory

- OptimismPortal

- SystemConfig

- L2OutputOracle

- L1StandardBridge

- L1CrossDomainMessenger

Recommended mitigation

Remove all the state manipulation code that are no longer necessary and are being applied to state variables that no longer exist. Leaving these may lead to issues in the future related to corrupt data.

Team response

Acknowledged. This will be tracked and addressed in future upgrades.

TRST-L-5 The gas buffer set is insufficient, leading to risks of unexpected reverts

- **Category:** Gas-related flaws
- **Source:** OptimismPortal.sol
- **Status:** Acknowledged

Description

The *CrossDomainMessenger* introduced a *RELAY_GAS_CHECK_BUFFER* which has a value of 5000. This buffer represents the gas that needs to be reserved for the execution between the *hasMinGas()* check and the external call in *relayMessage()*.

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

The assumption is that this 5000 buffer is enough to cover the gas required in the following code between *hasMinGas()* and *SafeCall.call()*:

```
    if (
        !SafeCall.hasMinGas(_minGasLimit, RELAY_RESERVED_GAS + RELAY_GAS_CHECK_BUFFER)
            || xDomainMsgSender != Constants.DEFAULT_L2_SENDER
    ) {
        failedMessages[versionedHash] = true;
        emit FailedRelayedMessage(versionedHash);
        // Revert in this case if the transaction was triggered by the estimation
address. 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");
        }
        return;
    }
    xDomainMsgSender = _sender;
```

A breakdown of the gas costs for the execution between the gas check and the external call follows:

- Cold SLOAD of **xDomainMsgSender** in **xDomainMsgSender != Constants.DEFAULT_L2_SENDER** - 2100 gas
- Non-zero to non-zero SSTORE of **xDomainMsgSender** in **xDomainMsgSender = _sender** - 2900 gas
- Other opcodes for comparisons, additions, multiplications and jumps - ~200-300 gas

The gas check buffer is insufficient by a few hundred gas units.

Recommended mitigation

Increase the gas check buffer by 1000 gas to 6000 gas for additional safety.

Team response

Acknowledged. This will be tracked and addressed in future upgrades.

TRST-L-6 OptimismPortal consumes all forwarded gas even if the TX is undeliverable

- **Category:** Gas-related flaws

- **Source:** OptimismPortal.sol
- **Status:** Acknowledged

Description

If a user does not forward enough gas to cover the gas cost of their deposit when calling *depositTransaction()*, then all the forwarded gas will be burned and the transaction will be reverted. A check can be implemented to prevent this unnecessary gas loss at the expense of the user.

Suppose a user calls *OptimismPortal::depositTransaction()* and the following parameters apply:

L2 base fee – 10 gwei

L1 base fee – 50 gwei

gasLimit – 1M

gas forwarded by the caller – 150,000

The total gas that should be burned on L1 is: `1e6 * 10e9 / 50e9 = 200,000`

With the above parameters, the 150,000 gas forward by the caller is not enough to cover the gas cost of 200,000. However, the metering function will still proceed with burning all the 150,000 gas of the caller and revert with an OOG (out-of-gas) error.

The issue lies in *Burn.gas()*:

```
    function gas(uint256 _amount) internal view {
        uint256 i = 0;
        uint256 initialGas = gasleft();
        while (initialGas - gasleft() < _amount) {
            ++i;
        }
    }
```

For simplicity, assume the **_amount** is the total gas cost of 200,000 and *gasleft()* is zero to get the maximum amount of the lefthand-side of the condition. Since the condition `150,000 < 200,000` will always be true, all the 150,000 gas will be burned and it will attempt to burn more but will revert due to no more gas left.

In fact, any deposit transaction that has insufficient gas forwarded with it will lose all that gas.

Recommended mitigation

A check can be added to *Burn.gas()* to prevent unnecessary burning when the forwarded gas is insufficient.

```
    function gas(uint256 _amount) internal view {
        uint256 i = 0;
        uint256 initialGas = gasleft();

        require(initialGas > _amount, " not enough gas");

        while (initialGas - gasleft() < _amount) {
            ++i;
        }
    }
```

Team response

Acknowledged. This will be tracked and addressed in future upgrades.

TRST-L-7 Users can underpay gas for contract creations, which would make them fail

- **Category:** Gas-related flaws
- **Source:** OptimismPortal.sol
- **Status:** Acknowledged

Description

For deposit transactions, a minimum gas limit is enforced to ensure that gas consumption on L2 has been paid for by users. However, this minimum gas limit only accounts for the data length and intrinsic TX cost (21000 gas units):

```
    function minimumGasLimit(uint64 _byteCount) public pure returns (uint64) {
        return _byteCount * 16 + 21000;
    }
```

This minimum gas limit does not account for the case of contract creations where it has a minimum fixed cost of 32000 gas on top of the inherent 21k cost. This leads to users possibly attempting to deploy contracts with a gas amount that is guaranteed to be insufficient.

Recommended mitigation

The minimum gas limit should account for the fixed costs and dynamic costs of contract creation when the deposit transaction is a contract creation.

Team response

Acknowledged. This will be tracked and addressed in future upgrades.

TRST-L-8 Different ERC20Factory addresses between chains makes it impossible to deploy ERC20s with the same address on all the chains

- **Category:** Deployments
- **Source:** OptimismMintableERC20Factory.sol
- **Status:** Acknowledged

Description

OptimismMintableERC20Factory deploys *OptimisMintableERC20* with the use of CREATE2. The intention for this is to enable deploying a token contract with the same address on all the OP Stack chains. For the contract to have the same address on all the chains, the following details must be consistent on every deployment on each chain:

1. Sender address – This would be the address of the *OptimismMintableERC20Factory*.
2. Bytecode of the *OptimismMintableERC20* contract
3. Constructor arguments passed to the *OptimismMintableERC20* contract which are ***BRIDGE***, **_remoteToken**, **_name**, **_symbol**, and **_decimals**.

Currently, the address for the *OptimismMintableERC20Factory* contract is different between Optimism and Base. This would make it impossible to have the same address for any ERC20 contracts deployed with the Factory in those chains.

Also worth noting is that the counterpart for Optimism's USDC, would be USDbC on Base. USDbC has a different name and symbol for its token. This difference in the constructor arguments would lead to different token addresses. The current implementation of the Factory does not allow for differences in the name, symbol, and decimals of the token if they are to have the same address on different chains.

Recommended mitigation

The *OptimismMintableERC20Factory* must be deployed to the same address on all the chains to enable token deployers to deploy the same address for their token. If there is value to enabling using different names, symbols, and decimals for the tokens while deploying to the same address, these details must instead be stored in state variables and not passed as constructor arguments. If not passed as constructor arguments, these details can instead be set in and fetched from a different contract.

Team response

Acknowledged.

Additional recommendations

Including constructor arguments in the salt is redundant

The *OptimismMintableERC20Factory* deploys *OptimismMintableERC20* contracts via CREATE2 for deterministic addresses to enable a token to have the same address across all OP Stack chains. The **salt** input to CREATE2 includes the construct arguments such as **_remoteToken**, **_name**, **_symbol**, and **_decimals**. This is redundant since the constructor arguments already

affect CREATE2 address derivation. In this case, **salt** could be a zero value or a value set by the user instead.

Centralization risks

Only risks introduced by the upgrade in scope will be detailed below.

TRST-CR-1 The SuperchainConfig guardian can pause all Superchains

The upgrade delegates responsibility for pausing withdrawals to the SuperchainConfig contract. This means all chains share the pause button, and therefore to handle an issue in one particular chain would require pausing all Superchains. It is acknowledged that the chosen model presents an advantage, whereby all chains which presumably share the same source code, can be paused in tandem, should a code-level emergency arise.