PoolTogether contest
Findings & Analysis Report

2023-01-20

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the PoolTogether smart contract system written in Solidity. The audit contest took place between December 1—December 5 2022.

Wardens

19 Wardens contributed reports to the PoolTogether contest:

1. 0x4non
2. 0x52
3. 0xSmartContract
4. AkshaySrivastav

5. Chom
6. Madalad
7. Rolezn
8. Tricko
9. adriro
10. cccz
11. cryptonue
12. csanuragjain
13. enckrish
14. gzeon
15. hihen
16. joestakey
17. ktg
18. ladboy233
19. neko_nyaa

This contest was judged by Alex the Entreprenerd.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 3 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 3 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 4 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 8 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 PoolTogether contest repository, and is composed of 7 smart contracts written in the Solidity programming language and includes 279 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

Medium Risk Findings (3)

[M-01] An attacker can make users unable to cancel their L1 calls on Ethereum To Arbitrum

Submitted by ktg, *also found by* 0x52

EthereumToArbitrumRelayer.sol#L118-#L127

When someone wants to make calls to Arbitrum from Ethereum, first they call `relayCalls` to fingerprint their data and then anyone else can call `processCalls` to process the calls. According to the doc in Inbox source code https://github.com/OffchainLabs/nitro/blob/1f32bec6b9b228bb2fab4bfa0286771

6f65d0c5c/contracts/src/bridge/Inbox.sol#L427, function `createRetryableTicket` has one parameter called `callValueRefundAddress` and this is the address that is granted the option to `cancel` a `Retryable`. In `EthereumToArbitrumRelayer.sol` it's currently set as `msg.sender` (5th parameter) which is whoever make the call to function `processCall`:

```
    uint256 _ticketID = inbox.createRetryableTicket{ value: msg.valu
            address(executor),
            0,
            _maxSubmissionCost,
            msg.sender,
            msg.sender,
            _gasLimit,
            _gasPriceBid,
            _data
        );
```

This implementation allows an attacker to remove the possibility of a user to cancel their calls, which is an important mechanism to be properly implemented. This scenario demonstrates how this could happen:

- User A call `relayCalls` to fingerprint their calls
- User B call `processCalls` to process user A's calls.
- User A now changes his mind and wants to cancel his calls but he's unable to since `callValueRefundAddress` is set to user B's address, now user B is the one who decides whether to cancel user A's calls or not, which should be user A's option.
- Another common case is when users's calls failed, anyone can try to `redeem` it, according to the doc https://developer.arbitrum.io/arbos/l1-to-l2-messaging. So if a someone calls `processCalls` to process others's calls and it fails, the owner of the calls now cannot cancel their calls and anyone else can redeem (reexecute) them.

It should be noted here that `EthereumToArbitrumRelayer.sol` provides no other functionality to cancel users's calls, but it seems to rely only on Arbitrum's Retryable cancel mechanism to do so.

Recommended Mitigation Steps

Currently, anyone can process others's calls by calling `processCalls` functions and I think this does not pose any security risk as long as the user who actually fingerprinted these calls can reserve their rights to cancel it if they want to. Therefore, I recommend changing `callValueRefundAddress` in `createRetryableTicket` to `_sender`, this combines with event `ProcessedCalls(_nonce, msg.sender, _ticketID)` emitted at the end of `processCalls` function will allow a user to be notified if their calls has been processed by anyone else and they can cancel it in L2 using `_ticketID`.

[Alex the Entreprenerd (judge) commented]():

> Relayer has privilege to cancel arbitrum txs, I think there may be a similar finding, but for now will keep separate.

[PierrickGT (PoolTogether) confirmed and commented]():

> Very nice find! I've fixed the issue in the following PR:
> https://github.com/pooltogether/ERC5164/pull/10

[Alex the Entreprenerd (judge) commented]():

> Worth flagging that allowing the caller to pass an arbitrary address may not solve, as I'd argue the only address that would rationally prevent the grief is the `_sender`, not fully sure if that is sufficient though.

[Alex the Entreprenerd (judge) commented]():

> The Warden has shown a specific scenario in which Arbitrum tickets, which are meant to be cancellable by the caller / the sender, can be griefed.

> Because this breaks the expectations of the code, and denies a functionality which was explicitly added, I agree with Medium Severity.

[M-02] When a smart contract calls

`CrossChainRelayerArbitrum.processCalls`, excess submission fees may be lost

Submitted by cccz, also found by joestakey, enckrish, and Chom

When the user calls CrossChainRelayerArbitrum.processCalls, ETH is sent as the submission fee.

According to the documentation : https://github.com/OffchainLabs/arbitrum/blob/master/docs/L1_L2_Messages.md#retryable-transaction-lifecycle

```
Credit-Back Address: Address to which all excess gas is creditec
...
Submission fee is collected: submission fee is deducted from the
```

The excess submission fee is refunded to the address on L2 of the `excessFeeRefundAddress` provided when calling `createRetryableTicket`.

```
     * @notice Put a message in the L2 inbox that can be reexecu
     * @dev all msg.value will deposited to callValueRefundAddre
     * @param destAddr destination L2 contract address
     * @param l2CallValue call value for retryable L2 message
     * @param  maxSubmissionCost Max gas deducted from user's L2
     * @param excessFeeRefundAddress maxgas x gasprice - executi
     * @param callValueRefundAddress l2Callvalue gets credited b
     * @param maxGas Max gas deducted from user's L2 balance to
     * @param gasPriceBid price bid for L2 execution
     * @param data ABI encoded data of L2 message
     * @return unique id for retryable transaction (keccak256(re
     */
    function createRetryableTicket(
        address destAddr,
        uint256 l2CallValue,
        uint256 maxSubmissionCost,
        address excessFeeRefundAddress,
        address callValueRefundAddress,
        uint256 maxGas,
        uint256 gasPriceBid,
        bytes calldata data
    ) external payable virtual override onlyWhitelisted returns
```

In `CrossChainRelayerArbitrum.processCalls`, `excessFeeRefundAddress == msg.sender`.

```
    uint256 _ticketID = inbox.createRetryableTicket{ value: msg.
      address(executor),
      0,
      _maxSubmissionCost,
      msg.sender,    // @audit : excessFeeRefundAddress
      msg.sender,   // @audit: callValueRefundAddress
      _gasLimit,
      _gasPriceBid,
      _data
    );
```

For EOA accounts, the excess submission fees are correctly refunded to their address on L2. However, for smart contracts, since there may not exist a corresponding address on L2, these excess submission fees will be lost.

Also, since the `callValueRefundAddress` is also `msg.sender`, according to the documentation, if the Retryable Ticket is cancelled or expired, then the smart contract caller may lose all the submission fees

```
    If the Retryable Ticket is cancelled or expires before it is rec
```

Proof of Concept

https://github.com/pooltogether/ERC5164/blob/5647bd84f2a6d1a37f41394874d567e45a97bf48/src/ethereum-arbitrum/EthereumToArbitrumRelayer.sol#L118-L127

https://github.com/OffchainLabs/arbitrum/blob/master/packages/arb-bridge-eth/contracts/bridge/Inbox.sol#L333-L354

Recommended Mitigation Steps

Consider allowing the user to specify `excessFeeRefundAddress` and `callValueRefundAddress` when calling `CrossChainRelayerArbitrum.processCalls`.

Alex the Entreprenerd (judge) commented:

> Making primary for quality of info.

> Ultimately boils down to the idea that contracts won't get a refund. Will have to think about whether this Med (submitted as such), or Low (self-inflicted).

PierrickGT (PoolTogether) confirmed and commented:

> Fixed in this PR: https://github.com/code-423n4/2022-12-pooltogether-findings/issues/63

> The `processCalls` function was intended to be called by an EOA only but it's true that a contract may want to call it while providing the required `_gasLimit`, `_maxSubmissionCost` and `_gasPriceBid` by an EOA.
> Passing a `refundAddress` variable will allow a contract to refund the EOA that called it.

> Regarding the severity, I think 2 (Med Risk) is appropriate since the contract would leak value.

Alex the Entreprenerd (judge) commented:

> Agree with finding, am conflicted on severity:

- Low -> User sends more than necessary
- Med -> Behaviour, is inconsistent to expected / intended functionality

> Will think about it further.

Alex the Entreprenerd (judge) commented:

> More specifically, the fact that the system wants to allow refunds and has a bug that prevents that, which would qualify as Medium. (We care if you send more, we

> will send it back, but because of bug we cannot)

> While the pre-condition, in case of a less sophisticated system, would most likely be Low (we don't care if you send more, don't send more)

Alex the Entreprenerd (judge) commented:

> The Warden has shown an incorrect implementation, which can cause excess fees to be lost.

> While the loss of excess fees could be considered Low Severity (self-inflicted), the integration mistake is worth flagging and warrants the increased severity.

[M-03] `CrossChainExecutor` contracts do not update the necessary states for failing transactions

Submitted by AkshaySrivastav*, also found by* ladboy233, hihen*, and* csanuragjain

EthereumToOptimismExecutor.sol#L45-L59
EthereumToArbitrumExecutor.sol#L31-L45

The `CrossChainExecutorArbitrum` and `CrossChainExecutorOptimism` contracts both use `CallLib` library to invoke `Call` s on external contract. As per the `CallLib` library implementation, any failing `Call` results in the entire transaction getting reverted.

The `CrossChainExecutor` contracts does not store whether the calls in `CallLib.Call[]` were already attempted which failed.

This creates several issues for `CrossChainExecutor` contracts.

1. Offchain components can be tricked to submit failing `Call[]` s again and again. This can be used to drain the offchain component of gas.
2. Once a failing `Call[]` was invoked (which failed) and if again the same `Call[]` is invoked, the transaction should revert with `CallsAlreadyExecuted` error but it reverts with `CallFailure` error.

3. It is difficult to determine whether a to-be executed `Call[]` is pending or the invocation was already tried but failed.

PoCs for the above issues are listed below.

Proof of Concept

Scenario 1

```
contract Foo {
    function bar() public {
        for(uint256 i; ; i++) {}
    }
}
```

- The attacker relays the `Foo.bar()` call in the `CrossChainRelayer` contract with `maxGasLimit` as the `_gasLimit` parameter.
- The transport layer tries to invoke the `Foo.bar()` call by calling the `CrossChainExecutor.executeCalls()` . This transaction reverts costing the transport layer client `maxGasLimit` gas.
- Since no state updates were performed in `CrossChainExecutor` , the transport layer still assumes the relayed call as pending which needs to be executed. The transport layer client again tries to execute the pending relayed call which reverts again.
- Repeated execution of the above steps can deplete the gas reserves of transport layer client.

Scenario 2

```
contract Foo {
    function bar() public {
        revert();
    }
}
```

- The attacker relays the `Foo.bar()` call in the `CrossChainRelayer` contract.
- The transport layer tries to invoke the `Foo.bar()` call by calling the `CrossChainExecutor.executeCalls()`. This transaction gets reverted.
- Since the relayed calls still seems as pending, the transport layer tries to invoke the `Foo.bar()` call again. This call should get reverted with `CallsAlreadyExecuted` error but it gets reverted with `CallFailure` error.

Recommended Mitigation Steps

The `CrossChainExecutor` contract should store whether a relayed call was attempted to be executed to make sure the execution cannot be tried again.

The `CallLib` library can be changed to not completely revert the transaction when any individual `Call` gets failed.

[Alex the Entreprenerd (judge) commented]:

> Not convinced by High Severity but the fact that you cannot determine whether calls were already attempted seems valid.

[PierrickGT (PoolTogether) confirmed, but disagreed with severity and commented]:

> Indeed, in the current implementation, it's pretty difficult to know which calls succeeded and which calls failed.

> So we've added two events:

- `event CallSuccess(uint256 callIndex, bytes successData);`
- `event CallFailure(uint256 callIndex, bytes errorData);`

> When a Call fails, we emit the `CallFailure` event and exit early the loop going through the batch calls. `CallLib.executeCalls` will return `false` and then the transaction will revert with the custom error `ExecuteCallsFailed`.

> If all calls have executed successfully, `CallLib.executeCalls` will return `true` and then the `ExecutedCalls` event will be emitted.

> This way, it's possible to know which calls succeeded and which didn't.
> If one Call fails, the entire transaction must revert cause the user may have intended to execute all the calls in one transaction and maybe some calls depends on others to succeed.

> I think this issue should be labeled as 2 (Med Risk) since it would indeed have been difficult for the transport layer client to figure out why the transaction failed and if it was worth replaying in the future.

[Alex the Entreprenerd (judge) commented]():

> I think the finding was well thought out and can tell it helped shaped the protocol.

> I believe Medium severity could be reasonably marked, however I think Low Severity to be the most appropriate one.

> Specifically:

- No loss of funds (beside gas happens)
- Similar architectures (e.g Chainlink Keepers), share the similar "cannot tell if failed or not"
- The responsibility for determining if the tx will fail is on the caller (relayer)

> For those reasons I believe QA Low (Notable finding for Relayer / Service Operators) to be the most appropriate.

> I will flag this during triage to get more opinions.

[Alex the Entreprenerd (judge) commented]():

> Some additional thinking I'm having is that a failed tx could remain un-broadcasted for an indefinite amount of time, and this could create issues for the receiving contract if / when the contract is made to not revert.

> Specifically the fact that a failed tx can be relayed in the future (no expiration) seems to create a risk that can cause loss of value, which leads to me believe there is a valid argument for Medium Severity.

[Alex the Entreprenerd (judge) decreased severity to Medium and commented]():

> After further thinking, I believe the most appropriate severity is Medium.

> The Warden has shown how the code allows the execution of old failed txs, while that is fine, I believe the lack of expiry can create situations in which a old message could be broadcasted and the broadcasting of it could cause a non idempotent behavior.

> The simplest example I can think of would be an unpause tx, that fails up until a set of contracts are paused, which would put a paused system (probably because of an exploit or the need for immediate stop) back into the unpaused state.

> While the externalities are multiple, I believe because:

- The general nature of the system
- The lack of expiration for old calls

> That Medium Severity is the most appropriate.

> Personally I would recommend considering a way to make calls expire after some time to avoid potential gotchas (or integrators may want to verify that via a nonce system or similar)

Low Risk and Non-Critical Issues

For this contest, 4 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xSmartContract** received the top score from the judge.

The following wardens also submitted reports: cryptonue, ladboy233, and gzeon.

Summary

Low Risk Issues List

Number	Issues Details	Context
[L-01]	Hard-coding the `maxGasLimit` variable may cause problems in the future	3

Total 1 issue

Non-Critical Issues List

Number	Issues Details	Context
[N-01]	For functions, follow Solidity standard naming conventions	4
[N-02]	Use a more recent version of Solidity	All contracts
[N-03]	For modern and more readable code; update import usages	6
[N-04]	Use of `bytes.concat()` instead of `abi.encodePacked(,)`	1
[N-05]	Missing Event for critical parameters change	4

Total 5 issues

[L-01] Hard-coding the `maxGasLimit` variable may cause problems in the future

The variable `maxGasLimit` is defined as immutable and its value is assigned in `constructor` but cannot be changed later

EVM-Based blockchains are hardforked and there is no such thing as Gas Limit etc. values may change, this has happened in the past, so it is recommended to have this value updated in the future

```
3 results - 3 files

src/ethereum-arbitrum/EthereumToArbitrumRelayer.sol:
  36    /// @notice Gas limit provided for free on Arbitrum.
  37:   uint256 public immutable maxGasLimit;
  38

src/ethereum-optimism/EthereumToOptimismRelayer.sol:
  25    /// @notice Gas limit provided for free on Optimism.
  26:   uint256 public immutable maxGasLimit;
  27

src/ethereum-polygon/EthereumToPolygonRelayer.sol:
  19    /// @notice Gas limit provided for free on Polygon.
  20:   uint256 public immutable maxGasLimit;
```

[N-01] For functions, follow Solidity standard naming conventions

Context:

```
4 results - 4 files

src/ethereum-arbitrum/EthereumToArbitrumRelayer.sol:
  40:   uint256 internal nonce;

src/ethereum-optimism/EthereumToOptimismRelayer.sol:
  29:   uint256 internal nonce;

src/ethereum-polygon/EthereumToPolygonRelayer.sol:
  23:   uint256 internal nonce;

src/libraries/CallLib.sol:
  48      */
  49:   function executeCalls(
  50:     uint256 _nonce,
  51:     address _sender,
  52:     Call[] memory _calls,
  53:     bool _executedNonce
  54:   ) internal {
```

The above codes don’t follow Solidity’s standard naming convention,

internal and private functions : the mixedCase format starting with an underscore (_mixedCase starting with an underscore)

[N-02] Use a more recent version of Solidity

Context:
All contracts

Description:
For security, it is best practice to use the latest Solidity version.
For the security fix list in the versions;
https://github.com/ethereum/solidity/blob/develop/Changelog.md

Recommendation:
Old version of Solidity is used , newer version can be used `(0.8.17)`

[N-03] For modern and more readable code; update import usages

Context:

```
6 results - 5 files

src/ethereum-arbitrum/EthereumToArbitrumExecutor.sol:
  7: import "../interfaces/ICrossChainExecutor.sol";
  8: import "../libraries/CallLib.sol";

src/ethereum-arbitrum/EthereumToArbitrumRelayer.sol:
  9: import "../libraries/CallLib.sol";

src/ethereum-optimism/EthereumToOptimismRelayer.sol:
  9: import "../libraries/CallLib.sol";

src/ethereum-polygon/EthereumToPolygonExecutor.sol:
  7: import "../libraries/CallLib.sol";

src/ethereum-polygon/EthereumToPolygonRelayer.sol:
  9: import "../libraries/CallLib.sol";
```

Description:
Solidity code is also cleaner in another way that might not be noticeable: the struct Point. We were importing it previously with global import but not using it. The Point struct `polluted the source code` with an unnecessary object we were not using because we did not need it.
This was breaking the rule of modularity and modular programming: `only import what you need` Specific imports with curly braces allow us to apply this rule better.

Recommendation:

```
import {contract1 , contract2} from "filename.sol";
```

A good example from the ArtGobblers project;

```
import {Owned} from "solmate/auth/Owned.sol";
import {ERC721} from "solmate/tokens/ERC721.sol";
import {LibString} from "solmate/utils/LibString.sol";
import {MerkleProofLib} from "solmate/utils/MerkleProofLib.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLi
import {ERC1155, ERC1155TokenReceiver} from "solmate/tokens/ERC1
import {toWadUnsafe, toDaysWadUnsafe} from "solmate/utils/Signe
```

[N-04] Use of `bytes.concat()` instead of `abi.encodePacked(,)`

```
src/libraries/CallLib.sol:
      Call memory _call = _calls[_callIndex];

      (bool _success, bytes memory _returnData) = _call.ta
        abi.encodePacked(_call.data, _nonce, _sender)
      );
```

Rather than using `abi.encodePacked` for appending bytes, since version 0.8.4, `bytes.concat()` is enabled

Since version 0.8.4 for appending bytes, `bytes.concat()` can be used instead of `abi.encodePacked(,)`.

[N-05] Missing Event for critical parameters change

```
src/ethereum-arbitrum/EthereumToArbitrumExecutor.sol:
     */
    function setRelayer(ICrossChainRelayer _relayer) externa
      require(address(relayer) == address(0), "Executor/rela
      relayer = _relayer;
    }

src/ethereum-optimism/EthereumToOptimismExecutor.sol:
     */
    function setRelayer(ICrossChainRelayer _relayer) externa
```

```
  67:      require(address(relayer) == address(0), "Executor/rela
  68:      relayer = _relayer;
  69:    }
  70

src/ethereum-arbitrum/EthereumToArbitrumRelayer.sol:
  138      */
  139:    function setExecutor(ICrossChainExecutor _executor) ext
  140:      require(address(executor) == address(0), "Relayer/exe
  141:      executor = _executor;
  142:    }
  143

src/ethereum-optimism/EthereumToOptimismRelayer.sol:
  84      */
  85:    function setExecutor(ICrossChainExecutor _executor) exte
  86:      require(address(executor) == address(0), "Relayer/exec
  87:      executor = _executor;
  88:    }
  89  }
```

Description:
Events help non-contract tools to track changes, and events prevent users from being surprised by changes.

Recommendation:
Add Event-Emit.

PierrickGT (PoolTogether) confirmed and commented:

> I've confirmed the issue because some suggestions have been fixed.

Alex the Entreprenerd (judge) commented:

> **[L-01] | Hard-coding the maxGasLimit variable may cause problems in the future | 3**
> Low

> **[N-01] | For functions, follow Solidity standard naming conventions | 4**
> Refactoring

> **[N-02] | Use a more recent version of Solidity | All contracts**
> Non-Critical

> **[N-03] | For modern and more readable code; update import usages | 6**
> Refactoring

> **[N-04] | Ùse of bytes.concat() instead of abi.encodePacked(,) | 1**
> Non-Critical

> **[N-05] | Missing Event for critical parameters change**
> Non-Critical

Gas Optimizations

For this contest, 8 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Tricko** received the top score from the judge.

The following wardens also submitted reports: adriro, cryptonue, Madalad, AkshaySrivastav, Rolezn, neko_nyaa, and 0x4non.

[G-01] Use bitmaps to save gas

During calls to `executeCalls` (and `processMessageFromRoot` in the Polygon executor) the `executed` mapping is updated, setting the bool flag to true, but because the way the EVM works it allocates an entire storage slot (256 bits) every time a new flag is set. `SSTORE` opcode cost up to 20000 gas for uninitialized slots like these. Consider using bitmaps instead, this enables you to convert the `mapping(uint256 => bool)` to a `mapping(uint256 => uint256)` and pay the cost of slot initialization just once every 256 `nonces` added, so the high gas costs are amortized over many calls.

Each group of 256 sequential `nonces` values (0-255, 256-511, ...) are stored together in a single uint256, where each bit represents a bool. The correct index of each nonce inside each bitmap can be calculated by `nonce mod 256`. An exemplified implementation is shown below (For complete implementations see Modifications section)

```
    mapping(uint256 => uint256) public executed;

    //setting executed nonce value to true
    uint256 baseIndex = _nonce / 256;
    uint256 index = _nonce - (256 * baseIndex);
    uint256 mask = 1 << index;
    executed[baseIndex] |= mask;

    //getting the executed bool flag for a specific nonce
    uint256 baseIndex = _nonce / 256;
    uint256 index = _nonce - (256 * baseIndex);
    uint256 mask = 1 << index;
    bool _executedNonce = (executed[baseIndex] & mask) != 0
```

The full extent of the gas reduction from these changes cannot be seen using the tests present in the repo, as during the tests `executeCalls` are called at most a few times, so the effect of amortization cannot be seen, underestimating the effect of these changes. Extra tests were added to enable benchmarking more realistic conditions (See modifications section below), where `executeCalls` is called 100 times in sequence, simulating many user interacting with the contract during normal operating conditions. Using those benchmarks, we obtain the following results.

	avg. gas (before modification)	avg. gas (after modification)	gas diff
CrossChainExecutorArbitrum - executeCalls	31322	9980	-21342 (-68.1%)
CrossChainExecutorOptimism - executeCalls	32241	10899	-21342 (-66.2%)
CrossChainExecutorPolygon - processMessageFromRoot	32279	10940	-21339 (-66.1%)
Total	95842	31819	-64023 (-66.8%)

*Values obtained by running `forge test --match-test Benchmark --gas-report`

Modifications

https://github.com/pooltogether/ERC5164/blob/5647bd84f2a6d1a37f41394874d567e45a97bf48/src/ethereum-arbitrum/EthereumToArbitrumExecutor.sol

```
diff --git a/EthereumToArbitrumExecutor.sol.orig b/EthereumToArb
index bfee411..cbe299b 100644
--- a/EthereumToArbitrumExecutor.sol.orig
+++ b/EthereumToArbitrumExecutor.sol
@@ -23,7 +23,7 @@ contract CrossChainExecutorArbitrum is ICross(
     *          nonce => boolean
     * @dev Ensure that batch of calls cannot be replayed once th
     */
-  mapping(uint256 => bool) public executed;
+  mapping(uint256 => uint256) public executed;

   /* ============ External Functions ============ */

@@ -36,8 +36,9 @@ contract CrossChainExecutorArbitrum is ICross(
     ICrossChainRelayer _relayer = relayer;
     _isAuthorized(_relayer);

-    bool _executedNonce = executed[_nonce];
-    executed[_nonce] = true;
+    (uint256 baseIndex, uint256 mask) = _getBaseIndexAndMask(_r
+    bool _executedNonce = (executed[baseIndex] & mask) != 0;
+    executed[baseIndex] |= mask;

     CallLib.executeCalls(_nonce, _sender, _calls, _executedNonc

@@ -54,6 +55,11 @@ contract CrossChainExecutorArbitrum is ICross
     relayer = _relayer;
   }

+  function isExecuted(uint256 _nonce) external view returns (bo
+    (uint256 baseIndex, uint256 mask) = _getBaseIndexAndMask(_r
+    return (executed[baseIndex] & mask) != 0;
+  }
+
   /* ============ Internal Functions ============ */

   /**
@@ -67,4 +73,11 @@ contract CrossChainExecutorArbitrum is ICross
       "Executor/sender-unauthorized"
     );
   }
+
+   function _getBaseIndexAndMask(uint256 _nonce) internal view r
+     uint256 baseIndex = _nonce / 256;
+     uint256 index = _nonce - (256 * baseIndex);
+     uint256 mask = 1 << index;
+     return (baseIndex, mask);
+   }
 }
```

https://github.com/pooltogether/ERC5164/blob/5647bd84f2a6d1a37f41394874d567e45a97bf48/src/ethereum-optimism/EthereumToOptimismExecutor.sol

```
diff --git a/EthereumToOptimismExecutor.sol.orig b/EthereumToOpt
index 1aba9c1..6763b22 100644
--- a/EthereumToOptimismExecutor.sol.orig
+++ b/EthereumToOptimismExecutor.sol
@@ -26,7 +26,7 @@ contract CrossChainExecutorOptimism is ICross(
     *          nonce => boolean
     * @dev Ensure that batch of calls cannot be replayed once th
     */
-   mapping(uint256 => bool) public executed;
+   mapping(uint256 => uint256) public executed;

    /* ============ Constructor ============ */

@@ -50,8 +50,9 @@ contract CrossChainExecutorOptimism is ICross(
      ICrossChainRelayer _relayer = relayer;
      _isAuthorized(_relayer);

-     bool _executedNonce = executed[_nonce];
-     executed[_nonce] = true;
+     (uint256 baseIndex, uint256 mask) = _getBaseIndexAndMask(_r
+     bool _executedNonce = (executed[baseIndex] & mask) != 0;
+     executed[baseIndex] |= mask;

      CallLib.executeCalls(_nonce, _sender, _calls, _executedNonc

@@ -68,6 +69,11 @@ contract CrossChainExecutorOptimism is ICross
      relayer = _relayer;
    }

+   function isExecuted(uint256 _nonce) external view returns (bo
+     (uint256 baseIndex, uint256 mask) = _getBaseIndexAndMask(_r
+     return (executed[baseIndex] & mask) != 0;
+   }
+
   /* ============ Internal Functions ============ */

   /**
@@ -83,4 +89,11 @@ contract CrossChainExecutorOptimism is ICross
       "Executor/sender-unauthorized"
     );
   }
+
+  function _getBaseIndexAndMask(uint256 _nonce) internal view r
+    uint256 baseIndex = _nonce / 256;
+    uint256 index = _nonce - (256 * baseIndex);
+    uint256 mask = 1 << index;
+    return (baseIndex, mask);
+  }
 }
```

https://github.com/pooltogether/ERC5164/blob/5647bd84f2a6d1a37f41394874d567e45a97bf48/src/ethereum-polygon/EthereumToPolygonExecutor.sol

```
diff --git a/EthereumToPolygonExecutor.sol.orig b/EthereumToPoly
index 29bc54f..9c0df8c 100644
--- a/EthereumToPolygonExecutor.sol.orig
+++ b/EthereumToPolygonExecutor.sol
@@ -28,7 +28,7 @@ contract CrossChainExecutorPolygon is FxBaseCh
    *          nonce => boolean
    * @dev Ensure that batch of calls cannot be replayed once th
    */
-  mapping(uint256 => bool) public executed;
+  mapping(uint256 => uint256) public executed;

   /* ============ Constructor ============ */

@@ -51,11 +51,24 @@ contract CrossChainExecutorPolygon is FxBase
       (uint256, address, CallLib.Call[])
     );

-    bool _executedNonce = executed[_nonce];
-    executed[_nonce] = true;
+    (uint256 baseIndex, uint256 mask) = _getBaseIndexAndMask(_r
+    bool _executedNonce = (executed[baseIndex] & mask) != 0;
+    executed[baseIndex] |= mask;

    CallLib.executeCalls(_nonce, _callsSender, _calls, _execute
      emit ExecutedCalls(_sender, _nonce);
    }
+
+   function isExecuted(uint256 _nonce) external view returns (bo
+     (uint256 baseIndex, uint256 mask) = _getBaseIndexAndMask(_n
+     return (executed[baseIndex] & mask) != 0;
+   }
+
+   function _getBaseIndexAndMask(uint256 _nonce) internal view r
+     uint256 baseIndex = _nonce / 256;
+     uint256 index = _nonce - (256 * baseIndex);
+     uint256 mask = 1 << index;
+     return (baseIndex, mask);
+   }
  }
```

https://github.com/pooltogether/ERC5164/blob/5647bd84f2a6d1a37f41394874d567e45a97bf48/test/unit/ethereum-arbitrum/EthereumToArbitrumExecutor.t.sol

```
diff --git a/EthereumToArbitrumExecutor.t.sol.orig b/EthereumToA
index 942d92d..37e7161 100644
--- a/EthereumToArbitrumExecutor.t.sol.orig
+++ b/EthereumToArbitrumExecutor.t.sol
@@ -70,7 +70,7 @@ contract CrossChainExecutorArbitrumUnitTest is

      executor.executeCalls(nonce, sender, calls);

-     assertTrue(executor.executed(nonce));
+     assertTrue(executor.isExecuted(nonce));
    }

   function testExecuteCallsAlreadyExecuted() public {
@@ -90,6 +90,17 @@ contract CrossChainExecutorArbitrumUnitTest i
      executor.executeCalls(nonce, sender, calls);
    }

+   function testExecuteCallsBenchmark() public {
+     setRelayer();
+
+     vm.startPrank(relayerAlias);
+
+     for (uint256 i; i < 100; i++) {
+       executor.executeCalls(nonce, sender, calls);
+       nonce++;
+     }
+   }
+
    /* ============ Setters ============ */

    function testSetRelayer() public {
```

https://github.com/pooltogether/ERC5164/blob/5647bd84f2a6d1a37f41394874d567e45a97bf48/test/fork/EthereumToOptimismFork.t.sol

```
diff --git a/EthereumToOptimismFork.t.sol.orig b/EthereumToOptim
index be84235..ed2d2e7 100644
--- a/EthereumToOptimismFork.t.sol.orig
+++ b/EthereumToOptimismFork.t.sol
@@ -194,6 +194,39 @@ contract EthereumToOptimismForkTest is Test
     assertEq(greeter.greet(), l1Greeting);
   }

+  function testExecuteCallsBenchmark() public {
+    deployAll();
+    setAll();
+
+    vm.selectFork(optimismFork);
+
+    CallLib.Call[] memory _calls = new CallLib.Call[](1);
+
+    _calls[0] = CallLib.Call({
+      target: address(greeter),
+      data: abi.encodeWithSignature("setGreeting(string)", l1Gr
+    });
+
+    L2CrossDomainMessenger l2Bridge = L2CrossDomainMessenger(l2
+
+    vm.startPrank(AddressAliasHelper.applyL1ToL2Alias(proxyOVMI
+
+    for (uint256 i; i < 100; i++) {
+      l2Bridge.relayMessage(
+        address(executor),
+        address(relayer),
+        abi.encodeWithSignature(
+          "executeCalls(uint256,address,(address,bytes)[])",
+          nonce,
+          address(this),
+           _calls
+         ),
+         l2Bridge.messageNonce() + 1
+       );
+       nonce++;
+     }
+   }
+
    function testGasLimitTooHigh() public {
      deployAll();
      setAll();
```

https://github.com/pooltogether/ERC5164/blob/5647bd84f2a6d1a37f41394874d567e45a97bf48/test/fork/EthereumToPolygonFork.t.sol

```
diff --git a/EthereumToPolygonFork.t.sol.orig b/EthereumToPolygc
index 01913e6..eaad5f2 100644
--- a/EthereumToPolygonFork.t.sol.orig
+++ b/EthereumToPolygonFork.t.sol
@@ -179,6 +179,31 @@ contract EthereumToPolygonForkTest is Test
      assertEq(greeter.greet(), l1Greeting);
    }

+   function testExecuteCallsBenchmark() public {
+     deployAll();
+     setAll();
+
+     vm.selectFork(polygonFork);
+
+     CallLib.Call[] memory _calls = new CallLib.Call[](1);
+
+     _calls[0] = CallLib.Call({
+       target: address(greeter),
+       data: abi.encodeWithSignature("setGreeting(string)", l1Gr
+     });
+
+     vm.startPrank(fxChild);
+
+     for (uint256 i; i < 100; i++) {
+       executor.processMessageFromRoot(
+         1,
+         address(relayer),
+         abi.encode(nonce, address(this), _calls)
+       );
+       nonce++;
+     }
+   }
+
    function testGasLimitTooHigh() public {
      deployAll();
```

PierrickGT (PoolTogether) acknowledged and commented:

> Very interesting optimization but it does complexify the code quite a bit.
> So for this reason, I've acknowledged the issue but we won't implement the suggestion.

Alex the Entreprenerd (judge) commented:

> I believe that in the spirit of Gas Reports, the report has shown:

- A refactoring with code snippets
- It's fully benchmarked

> Am accepting the finding as valid, and because it effectively saves one SSTORE on each operation, I believe 20k gas to be the appropriate amount saved

> 20_000

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top