# zkSync v2 contest
# Findings & Analysis Report

2023-01-04

## 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 zkSync v2 smart contract system written in Solidity. The audit contest took place between October 28—November 9 2022.

## Wardens

24 Wardens contributed reports to the zkSync v2 contest:

1. 0x1f8b

2. 0xSmartContract
3. Aymen0909
4. HE1M
5. HardlyCodeMan
6. IIIIIII
7. ReyAdmirado
8. Rolezn
9. Soosh
10. TomJ
11. Tomo
12. brgltd
13. c3phas
14. cccz
15. chaduke
16. codehacker
17. ctf_sec
18. datapunk
19. gogo
20. jayjonah8
21. ladboy233
22. mcwildy
23. pashov
24. rbserver

This contest was judged by Alex the Entreprenerd.

Final report assembled by liveactionllama.

## Summary

The C4 analysis yielded an aggregated total of 2 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 2

received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 13 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 10 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 zkSync v2 contest repository, and is composed of 39 smart contracts written in the Solidity programming language and includes 3,727 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 (2)

## [M-01] `diamondCut` is not protected in case of governor's key leakage

*Submitted by* HE1M*, also found by* codehacker

When the governor proposes a diamondCut, governor must wait for `upgradeNoticePeriod` to be passed, or security council members have to approve the proposal to bypass the notice period, so that the governor can execute the proposal.

```
require(approvedBySecurityCouncil || upgradeNoticePeriodPasse
require(approvedBySecurityCouncil || !diamondStorage.isFrozer
```

If the governor's key is leaked and noticed by zkSync, the attacker must wait for the notice period to execute the already proposed diamondCut with the malicious `_calldata` based on the note below from zkSync, or to propose a new malicious diamondCut. For, both cases, the attacker loses time.

> NOTE: proposeDiamondCut - commits data associated with an upgrade but does not execute it. While the upgrade is associated with facetCuts and (address _initAddress, bytes _calldata) the upgrade will be committed to the facetCuts and _initAddress. This is done on purpose, to leave some freedom to the governor to change calldata for the upgrade between proposing and executing it.

Since, there is a notice period (as zkSync noticed the key leakage, security council member will not approve the proposal, so bypassing the notice period is not possible), there is enough time for zkSync to apply security measures (pausing any deposit/withdraw, reporting in media to not execute any transaction in zkSync, and so on).

But, the attacker can be smarter, just before the proposal be executed by the governor (i.e. the notice period is passed or security council members approved it), the attacker executes the proposal earlier than governor with the malicious `_calldata`. In other words, the attacker front runs the governor.

Therefore, if zkSync notices the governor's key leakage beforehand, there is enough time to protect the project. But, if zkSync does not notice the governor's key leakage,

the attacker can change the `_calldata` into a malicious one in the last moment so that it is not possible to protect the project.

### Proof of Concept

[Diamond.sol#L277](#)
[DiamondCut.sol#L46](#)

### Recommended Mitigation Steps

`_calldata` should be included in the proposed diamondCut:[DiamondCut.sol#L27](#).

Or, at least one of the security council members should approve the `_calldata` during execution of the proposal.

**[miladpiri (zkSync) confirmed and commented](#):**

> It is a valid issue, and the fix is going to be implemented, so we confirm the issue as medium! Thanks.

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

> In contrast to other reports, this shows how a malicious proposal could be injected, bypassing the timelock protection, for this reason (after consulting with a second Judge), I agree with marking it as a distinct finding and agree with Medium Severity.

## [M-02] `BLOCK_PERIOD` is incorrect

*Submitted by* [Soosh](#)

[Config.sol#L47](#)

The `BLOCK_PERIOD` is set to 13 seconds in `Config.sol`.

```
    uint256 constant BLOCK_PERIOD = 13 seconds;
```

Since moving to Proof-of-Stake (PoS) after the Merge, block times on ethereum are fixed at 12 seconds per block (slots).
https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/#:~:text=Whereas%20under%20proof%2Dof%2Dwork,block%20proposer%20in%20every%20slot.

## Impact

This results in incorrect calculation of `PRIORITY_EXPIRATION` which is used to determine when a transaction in the Priority Queue should be considered expired.

```
uint256 constant PRIORITY_EXPIRATION_PERIOD = 3 days;
/// @dev Expiration delta for priority request to be satisfied (
uint256 constant PRIORITY_EXPIRATION = PRIORITY_EXPIRATION_PERIC
```

The time difference can be calulated

```
>>> 3*24*60*60 / 13    # 3 days / 13 sec block period
19938.46153846154
>>> 3*24*60*60 / 12    # 3 days / 12 sec block period
21600.0
>>> 21600 - 19938      # difference in blocks
1662
>>> 1662 * 12 / (60 * 60) # difference in hours
5.54
```

By using block time of 13 seconds, a transaction in the Priority Queue incorrectly expires 5.5 hours earlier than is expected.

5.5 hours is a significant amount of time difference so I believe this issue to be Medium severity.

## Recommended Mitigation Steps

Change the block period to be 12 seconds

```
uint256 constant BLOCK_PERIOD = 12 seconds;
```

miladpiri (zkSync) confirmed and commented:

> This is a valid medium issue! Thanks!

Alex the Entreprenerd (judge) commented:

> The warden has shown how, due to an incorrect configuration, L2Transactions will expire earlier than intended.

> The value would normally be rated a Low Severity, however, because the Warden has shown a more specific impact, I agree with Medium Severity.

## Low Risk and Non-Critical Issues

For this contest, 13 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **HE1M** received the top score from the judge.

*The following wardens also submitted reports:* brgltd, rbserver, pashov, ladboy233, 0xSmartContract, datapunk, cccz, ctf_sec, Tomo, Rolezn, jayjonah8, *and* chaduke.

### [01]

The critical parameters in `initialize(...)` are not set safely:

- `s.governor` should be set to `msg.sender`, because a wrong governor address will result in loss of access to all other parts, and later changing the governor to the correct address.
- `_l2BootloaderBytecodeHash` should be validated like `L2ContractHelper.validateBytecodeHash(_l2BootloaderBytecodeHash)` as in `GovernanceFacet`
- `_l2DefaultAccountBytecodeHash` should be validated like `L2ContractHelper.validateBytecodeHash(_l2DefaultAccountBytecodeHash)` as in `GovernanceFacet`

DiamondInit.sol#L39
DiamondInit.sol#L58
DiamondInit.sol#L59

## [02]

Better to have also `approveEmergencyDiamondCutAsSecurityCouncilMemberBySignature`, in case the security council members do not have access to ethereum blockchain or in case it is needed to approve just in one transaction by batch of signatures (to bypass the notice period).

## [03]

Better to have config facet, in case some update is needed in the `config.sol`. Therefore, it is not necessary to redeploy the facets that imported config (like `Executor` and `Mailbox`).

## [04]

`L2_LOG_BYTES` is not correct, it should be `L2_TO_L1_LOG_SERIALIZE_SIZE`
Config.sol#L19

## [05]

It is not needed to have modifier `senderCanCallFunction` for the function `deposit` in both `L1ERC20Bridge` and `L1ETHBridge`, because they call the function `requestL2Transaction` in the `MailBox` that has already such modifier.
L1EthBridge.sol#L92
Mailbox.sol#L112

## [06]

For each deposit of an ERC20 token, the information of the token is read and packed and sent to L2. Even if this token is used before for deposit, again this information is sent to L2, which is waste of gas.
L1ERC20Bridge.sol#L164-L169
L1ERC20Bridge.sol#L155

## [07]

When a block is committed, its hash will be stored in `storedBlockHashes`:
Executor.sol#L164

If this block is reverted, it is not removed from `storedBlockHashes`:
Executor.sol#L336

The vulnerability is that in the `GettersFacet`, the function `storedBlockHash(...)` will return the hash of a reverted block if this block number is given as its parameter, while it should return 0.
Getters.sol#L86

Alex the Entreprenerd (judge) commented:

> **The critical parameters in initialize(...) are not set safely:**

> Logically equivalent to address(0) check, Low

> **approveEmergencyDiamondCutAsSecurityCouncilMemberBySignature**

> Not sure what this means, but if it means one caller gets to approve emerengency cuts, this is a glaring security risk.

> **Better to have config facet, in case some update is needed in the config.sol**

> Not convinced by this one either, ultimately it's called config but it's just a bunch of contact / base contract

> **L2*LOG*BYTES is not correct, it should be `L2_TO_L1_LOG_SERIALIZE_SIZE`**

> Non-Critical

> **It is not needed to have modifier senderCanCallFunction for the function deposit in both L1ERC20Bridge and L1ETHBridge, because they call the function requestL2Transaction in the MailBox that has already such modifier.**

> Not convinced in lack of detail, if you call contract X and contract X calls contract Y, then the check is necessary on both contracts

> **For each deposit of an ERC20 token, the information**

> Unclear what you'd do and where the savings would be.

> **When a block is committed, its hash will be stored in storedBlockHashes:**

> See #204 , Low

**Alex the Entreprenerd (judge) commented:**

> 2 Low, 1 Non-Critical

> Report was pretty good, but the downgraded findings add a lot of points to this QA.

> While the Warden has sent a few false positives, I think the value they offered for this contest warrants them winning the best QA report.

*(Note: please see warden's original submission for links to the referenced downgraded findings)*

**miladpiri (zkSync) commented:**

> Thanks, @Alex the Entreprenerd for your comment, that we are now looking at this report more seriously.

> IMO, No.2 "approveEmergencyDiamondCutAsSecurityCouncilMemberBySignature" can be a security issue. This is what we are now implementing to make security council members be able to approve the proposal by signature.

> The security issue is that in case the security council members do not have access to approve on-chain, the governor must wait for their approval to bypass the notice period. In case the proposal should be executed instantly (in case of a critical bug in our protocol), this delay can be dangerous to our protocol.

> By having approval by signature, the security members can approve a proposal off-chain, and the governor can execute the instant upgrade on behalf of them by using their signatures, so all-time access to the chain is not necessary for security council members.

> Moreover, in case a critical proposal should be executed silently, it is necessary to be able to approve by signature. Because, the governor, in one batch transaction, proposes the proposal, and approves the upgrade on behalf of the security council members by using their signatures, and then executes the proposal. Since all the above actions are done in one transaction, there is no security risk of vulnerability leak. But, in the current structure, when the governor proposes the proposal, should wait for security members' approval, this delay can be transparent to a malicious user to investigate the governor's proposal, and gets a clue what the vulnerability is and then exploits the protocol.

> This report worths to be upgraded in terms of severity! Thanks.

**Alex the Entreprenerd (judge) commented**:

> Thank you for your insight, I have sent the contest to triage, took note of your feedback and am sharing with other Judges and Wardens.

**Alex the Entreprenerd (judge) commented**:

> Similarly to `#48` , I have shared this finding with 2 Judges and a Top warden and we all agree that this is effectively the same finding, with similar impact.

> As such Low Severity is the most appropriate.

**miladpiri (zkSync) commented**:

> Thanks for the follow-up!

## Gas Optimizations

For this contest, 10 reports were submitted by wardens detailing gas optimizations. The **report highlighted below** by **0x1f8b** received the top score from the judge.

*The following wardens also submitted reports:* Aymen0909, c3phas, TomJ, gogo, lllllll, ReyAdmirado, mcwildy, Rolezn, *and* HardlyCodeMan.

**Disclosure Note from 0x1f8b:**

*First of all, please note that there are "known issues" that were not referenced in the report, these lines were included in this report because they were not present in the public one.*

## [G-01] Use `require` instead of `assert`

The `assert()` and `require()` functions are a part of the error handling aspect in Solidity. Solidity makes use of state-reverting error handling exceptions. This means all changes made to the contract on that call or any sub-calls are undone if an error is thrown. It also flags an error.

They are quite similar as both check for conditions and if they are not met, would throw an error.

The big difference between the two is that the `assert()` function when false, **uses up all the remaining gas and reverts all the changes made.**

Meanwhile, a `require()` function when false, also reverts back all the changes made to the contract but **does refund all the remaining gas fees we offered to pay**. This is the most common Solidity function used by developers for debugging and error handling.

**Affected source code:**

- DiamondCut.sol:16

## [G-02] Avoid compound assignment operator in state variables

Using compound assignment operators for state variables (like `State += X` or `State -= X` ...) it's more expensive than using operator assignment (like `State = State + X` or `State = State - X` ...).

Proof of concept (*without optimizations*):

```
pragma solidity 0.8.15;

contract TesterA {
uint private _a;
function testShort() public {
_a += 1;
}
}

contract TesterB {
uint private _a;
function testLong() public {
_a = _a + 1;
}
}
```

Gas saving executing: **13 per entry**

```
TesterA.testShort: 43507
TesterB.testLong:  43494
```

**Affected source code:**

- DiamondCut.sol:29

Total gas saved: 13 * 1 = 13

## [G-03] Use `calldata` instead of `memory`

Some methods are declared as `external` but the arguments are defined as `memory` instead of as `calldata`.

By marking the function as `external` it is possible to use `calldata` in the arguments shown below and save significant gas.

**Recommended change:**

```
-   function decodeString(bytes memory _input) external pure ret
+   function decodeString(bytes calldata _input) external pure r
        (result) = abi.decode(_input, (string));
    }
```

**Affected source code:**

- ExternalDecoder.sol:10-12
- ExternalDecoder.sol:15-17

## [G-04] Shift right or left instead of dividing or multiply by 2

Shifting one to the right will calculate a division by two.

he `SHR` opcode only requires 3 gas, compared to the `DIV` opcode's consumption of 5. Additionally, shifting is used to get around Solidity's division operation's division-by-0 prohibition.

**Proof of concept (*without optimizations*):**

```
pragma solidity 0.8.16;

contract TesterA {
function testDiv(uint a) public returns (uint) { return a / 2; }
}

contract TesterB {
function testShift(uint a) public returns (uint) { return a >> 1
}
```

Gas saving executing: **172 per entry**

```
TesterA.testDiv:    21965
TesterB.testShift:  21793
```

The same optimization can be used to multiply by 2, using the left shift.

```
pragma solidity 0.8.16;

contract TesterA {
function testMul(uint a) public returns (uint) { return a * 2; }
}

contract TesterB {
function testShift(uint a) public returns (uint) { return a << 1
}
```

Gas saving executing: **201 per entry**

```
TesterA.testMul:    21994
TesterB.testShift:  21793
```

**Affected source code:**

`/`:

- Merkle.sol:34

Total gas saved: (172 * 1) = 172

## [G-05] There's no need to set default values for variables

If a variable is not set/initialized, the default value is assumed (0, `false`, 0x0 ... depending on the data type). You are simply wasting gas if you directly initialize it with its default value.

**Proof of concept (*without optimizations*):**

```
pragma solidity 0.8.15;

contract TesterA {
function testInit() public view returns (uint) { uint a = 0; ret
}

contract TesterB {
```

```
function testNoInit() public view returns (uint) { uint a; retur
}
```

Gas saving executing: **8 per entry**

```
TesterA.testInit:   21392
TesterB.testNoInit: 21384
```

**Affected source code:**

- L1EthBridge.sol:33
- L2ETHBridge.sol:28

Total gas saved: 8 * 2 = 16

## [G-06] Unnecessary cast in `Mailbox.serializeL2Transaction`

It's possible to remove the following casts:

```
    function serializeL2Transaction(
        uint256 _txId,
        uint256 _l2Value,
        address _sender,
        address _contractAddressL2,
        bytes calldata _calldata,
        uint256 _ergsLimit,
        bytes[] calldata _factoryDeps
    ) public pure returns (L2CanonicalTransaction memory) {
        return
            L2CanonicalTransaction({
                txType: PRIORITY_OPERATION_L2_TX_TYPE,
                from: uint256(uint160(_sender)),
                to: uint256(uint160(_contractAddressL2)),
                ergsLimit: _ergsLimit,
                ergsPerPubdataByteLimit: uint256(1),
                maxFeePerErg: uint256(0),
                maxPriorityFeePerErg: uint256(0),
                paymaster: uint256(0),
```

```
-                reserved: [uint256(_txId), _l2Value, 0, 0, 0, 0]
+                reserved: [_txId, _l2Value, 0, 0, 0, 0],
                 data: _calldata,
                 signature: new bytes(0),
                 factoryDeps: _hashFactoryDeps(_factoryDeps),
                 paymasterInput: new bytes(0),
                 reservedDynamic: new bytes(0)
             });
         }
```

**Affected source code:**

- Mailbox.sol:206

## [G-07] Gas saving using `immutable`

It's possible to avoid storage access a save gas using `immutable` keyword for the following variables:

It's also better to remove the initial values, because they will be set during the constructor.

**Affected source code:**

- L2ERC20Bridge.sol:19
- L2ERC20Bridge.sol:23
- L2ERC20Bridge.sol:26
- L2ETHBridge.sol:22

## [G-08] Reorder structure layout

The following structs could be optimized moving the position of certains values in order to save slot storages:

`StoredBlockInfo` in IExecutor.sol#L15-L24

```
    struct StoredBlockInfo {
        uint64 blockNumber;
```

```
+       uint64 indexRepeatedStorageChanges;
        bytes32 blockHash;
-       uint64 indexRepeatedStorageChanges;
        uint256 numberOfLayer1Txs;
        bytes32 priorityOperationsHash;
        bytes32 l2LogsTreeRoot;
        uint256 timestamp;
        bytes32 commitment;
    }
```

`AppStorage` in [Storage.sol:69-106]

```
    struct AppStorage {
        /// @dev Storage of variables needed for diamond cut face
        DiamondCutStorage diamondCutStorage;
        /// @notice Address which will exercise governance over t
        address governor;
+       bool zkPorterIsAvailable;
        /// @notice Address that governor proposed as one that wi
        address pendingGovernor;
        /// @notice List of permitted validators
        mapping(address => bool) validators;
        /// @dev Verifier contract. Used to verify aggregated pro
        Verifier verifier;
        /// @notice Total number of executed blocks i.e. blocks[t
        uint256 totalBlocksExecuted;
        /// @notice Total number of proved blocks i.e. blocks[tot
        uint256 totalBlocksVerified;
        /// @notice Total number of committed blocks i.e. blocks[
        uint256 totalBlocksCommitted;
        /// @dev Stored hashed StoredBlock for block number
        mapping(uint256 => bytes32) storedBlockHashes;
        /// @dev Stored root hashes of L2 -> L1 logs
        mapping(uint256 => bytes32) l2LogsRootHashes;
1
2       PriorityQueue.Queue priorityQueue;
3       /// @dev The smart contract that manages the list with pe
4       IAllowList allowList;
5       /// @notice Part of the configuration parameters of ZKP c
6       VerifierParams verifierParams;
7       /// @notice Bytecode hash of bootloader program.
8       /// @dev Used as an input to zkp-circuit.
9       bytes32 l2BootloaderBytecodeHash;
10      /// @notice Bytecode hash of default account (bytecode fo
```

```
11      /// @dev Used as an input to zkp-circuit.
12      bytes32 l2DefaultAccountBytecodeHash;
13      /// @dev Indicates that the porter may be touched on L2 t
14      /// @dev Used as an input to zkp-circuit.
15  -   bool zkPorterIsAvailable;
16  }
```

## [G-09] Use `require` instead of `assert`

The `assert()` and `require()` functions are a part of the error handling aspect in Solidity. Solidity makes use of state-reverting error handling exceptions. This means all changes made to the contract on that call or any sub-calls are undone if an error is thrown. It also flags an error.

They are quite similar as both check for conditions and if they are not met, would throw an error.

The big difference between the two is that the `assert()` function when false, **uses up all the remaining gas and reverts all the changes made.**

Meanwhile, a `require()` function when false, also reverts back all the changes made to the contract but **does refund all the remaining gas fees we offered to pay.** This is the most common Solidity function used by developers for debugging and error handling.

**Affected source code:**

- DiamondCut.sol:16

Alex the Entreprenerd (judge) commented:

> **[G-01] Use require instead of assert**
> Valid but will only run in constructor so ignoring

> **[G-02] Avoid compound assignment operator in state variables**
> Awarding 34 as that's what I get when changing the in-scope tests

> **[G-03] Use calldata instead of memory**
> Cannot test the benchmark, but it's roughly in the hundreds, let's say 200 gas

> **[G-04] Shift right or left instead of dividing or multiply by 2**
> This mostly saves gas because of the unchecked, let's say 100 gas

> **[G-05] There's no need to set default values for variables**
> Those are constants, will be inlined by the compiler

> **[G-06] Unnecessary cast in Mailbox.serializeL2Transaction**
> The cast should be optimized away by the compiler as no supporting variable is declared

> **[G-07] Gas saving using immutable**
> 2.1k per var
> 8.4k

> **[G-08] Reorder structure layout**
> 2k each as it will save one extra cold Slot Load
> 4k

> Pretty good, but would benefit by using benchmarks from the codebase in-scope.

> 10734

Alex the Entreprenerd (judge) commented:

> Ultimately offered the strongest savings via immutables and did not offer false positives vs other reports, also found the packing refactoring which will offer great savings to end users.

## Disclosures

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