Introducing Code4rena Blue: Dedicated defense. Competitive bounties. Independent judging.

Learn more →

zkSync Era System Contracts contest Findings & Analysis Report

2023-06-14

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 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 outlined in this document, C4 conducted an analysis of the zkSync Era System Contracts smart contract system written in Solidity. The audit took place between March 10—March 19 2023.

Wardens

19 Wardens contributed reports to the zkSync Era System Contracts audit:

1. 0x73696d616f
2. 0xSmartContract
3. Dravee
4. Franfran

5. HE1M
6. Jeiwan
7. Madalad
8. Ruhum
9. bin2chen
10. brgltd
11. bshramin
12. gjaldon
13. joestakey
14. minaminao
15. rbserver
16. ronnyx2017
17. rvierdiiev
18. supernova
19. unforgiven

This audit was judged by Alex the Entreprenerd.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 6 unique vulnerabilities. Of these vulnerabilities, 1 received a risk rating in the category of HIGH severity and 5 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 14 reports detailing issues with a risk rating of LOW severity or non-critical.

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

Note: while the bootloader/bootloader.yul *was out of scope for this audit, the zkSync team decided to reward an additional bounty for valid related vulnerabilities. There were 3 such vulnerabilities found:*

- Paymaster context can override a transaction hash in memory
- Bootloader can refund less gas than user paid for L2 tx
- Operator can cause funds to be stolen by manipulating gas fee refund

Scope

The code under review can be found within the C4 zkSync Era System Contracts audit repository, and is composed of 8 libraries, 17 interfaces, 1 abstract, 1 constant, and 13 smart contracts written in the Solidity programming language and includes 2,418 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on 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

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (1)

[H-01] The call to `MsgValueSimulator` with non zero `msg.value` will call to sender itself which will bypass the `onlySelf` check

Submitted by ronnyx2017

First, I need to clarify, there may be more serious ways to exploit this issue. Due to the lack of time and documents, I cannot complete further exploit. The current exploit has only achieved the impact in the title. I will expand the possibility of further exploit in the poc chapter.

The call to MsgValueSimulator with non zero msg.value will call to sender itself with the msg.data. It means that if you can make a contract or a custom account call to specified address with non zero msg.value (that's very common in withdrawal functions and smart contract wallets), you can make the contract/account call itself. And if you can also control the calldata, you can make the contract/account call its functions by itself.

It will bypass some security check with the msg.sender, or break the accounting logic of some contracts which use the msg.sender as account name.

For example the `onlySelf` modifier in the ContractDepolyer contract:

```
    modifier onlySelf() {
        require(msg.sender == address(this), "Callable only by self'
        _;
    }
```

Proof of Concept

The `MsgValueSimulator` use the `mimicCall` to forward the original call.

```
    return EfficientCall.mimicCall(gasleft(), to, _data, msg.sender,
```

And if the `to` address is the `MsgValueSimulator` address, it will go back to the `MsgValueSimulator.fallback` function again.

The fallback function will Extract the `value` to send, isSystemCall flag and the `to` address from the extraAbi params(r3,r4,r5) in the `_getAbiParams` function. But it's different from the first call to the `MsgValueSimulator`. The account uses `EfficientCall.rawCall` function to call the `MsgValueSimulator.fallback` in the first call. For example, code in `DefaultAccount._execute`:

```
bool success = EfficientCall.rawCall(gas, to, value, data);
```

The rawCall will simulate `system_call_byref` opcode to call the `MsgValueSimulator`. And the `system_call_byref` will write the r3-r5 registers which are read as the above extraAbi params.

But the second call is sent by `EfficientCall.mimicCall`, as the return value explained in the document https://github.com/code-423n4/2023-03-zksync/blob/main/docs/VM-specific_v1.3.0_opcodes_simulation.pdf, `mimicCall` will mess up the registers and will use r1-r4 for standard ABI convention and r5 for the extra who*to*mimic arg. So extraAbi params(r3-r5) read by `_getAbiParams` will be messy data. It can lead to very serious consequences, because the r3 will be used as the msg.value, and the r4 will be used as the `to` address in the final `mimicCall`. It means that the contract will send a different(greater) value to a different address, which is unexpected in the original call.

I really don't know how to write a complex test to verify register changes in the era-compiler-tester. So to find out how to control the registers, I use the repo https://github.com/matter-labs/zksync-era and replace the etc/system-contracts/ codes with the lastest version in the audit, and write an integration test.

```
import { TestMaster } from '../src/index';
import * as zksync from 'zksync-web3';
import { BigNumber } from 'ethers';

describe('ETH token checks', () => {
    let testMaster: TestMaster;
    let alice: zksync.Wallet;
    let bob: zksync.Wallet;

    beforeAll(() => {
        testMaster = TestMaster.getInstance(__filename);
        alice = testMaster.mainAccount();
        bob = testMaster.newEmptyAccount();
    });

    test('Can perform a transfer (legacy)', async () => {
        const LEGACY_TX_TYPE = 0;
        const value = BigNumber.from(30000);
        const MSG_VALUE_SYSTEM_CONTRACT = "0x0000000000000000000(
        console.log(await alice.getBalance());
        console.log(await alice.provider.getBalance(MSG_VALUE_SY

        let block = await alice.provider.getBlock("latest");
        console.log("block gas limit", block.gasLimit);
        let tx_gasLimit = block.gasLimit.div(8);
        console.log("tx_gasLimit", tx_gasLimit);
        console.log("gas price", await alice.getGasPrice());

        try {
            let tx = await alice.sendTransaction({ type: LEGACY_
            let txp = await tx.wait();
            console.log("success");
            console.log(txp["logs"]);
        } catch (err ) {
            console.log("fail");
            console.log(err);
            console.log('--------');
            console.log(err["receipt"]["logs"]);
        }

        console.log(await alice.getBalance());
        console.log(await alice.provider.getBalance(MSG_VALUE_SY
        console.log(await alice.getNonce());
    });

    afterAll(async () => {
        await testMaster.deinitialize();
    });
});
```

The L2EthToken Transfer event logs:

```
    {
        transactionIndex: 0,
        blockNumber: 25,
        transactionHash: '0x997b6536c802620e56f8c1b54a0bd3092dfe
        address: '0x000000000000000000000000000000000000800A',
        topics: [
          '0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f
          '0x0000000000000000000000006a8b37bcf2decff1452fccedc14
          '0x0000000000000000000000000000000000000000000000000000(
        ],
        data: '0x000000000000000000000000000000000000000000000000
        logIndex: 1,
        blockHash: '0xafb60d1285fc9ac08db01b02df01f6cbb668918d98
      },
      {
        transactionIndex: 0,
        blockNumber: 25,
        transactionHash: '0x997b6536c802620e56f8c1b54a0bd3092dfe
        address: '0x000000000000000000000000000000000000800A',
        topics: [
          '0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f
          '0x0000000000000000000000006a8b37bcf2decff1452fccedc14
          '0x0000000000000000000000006a8b37bcf2decff1452fccedc14
        ],
        data: '0x00000000000000000000000000000000000000000000002129
        logIndex: 2,
        blockHash: '0xafb60d1285fc9ac08db01b02df01f6cbb668918d98
      }
```

There are two l2 eth token transaction in addition to gas processing. And the value sent to the `MsgValueSimulator` will stuck in the contract forever.

I found that the r4(to) is always msg.sender, the r5(mask) is always 0x1, and if the length of the calldata is 0, the r3(value) will be `0x2129c0000000a00000000` , and if the length > 0, r3(value) will be `0x215800000000a00000000 + calldata.length << 96` . So in this case, the balance of the sender should be at least 0x2129c0000000a00000000 wei to finish the whole transaction whitout reverting.

I did not find any document about the `standard ABI convention` mentioned in the VM-specific*v1.3.0*\opcodes*simulation.pdf and the r5 is also not really the extra who*\to_mimic arg. I didn't make a more serious exploit due to lack of time. I'd like more documentation about register call conventions to verify the possibility of manipulating registers.

Recommended Mitigation Steps

Check the `to` address in the `MsgValueSimulator` contract. The `to` address must not be the `MsgValueSimulator` address.

Note: extensive discussion took place regarding this issue. Final comments are included below. For full details, please see the original submission.

vladbochok (zkSync) commented:

> Hey @ronnyx2017 & @Alex the Entreprenerd,

> I managed to reproduce the issue. @ronnyx2017 is right, if Alice calls `msgValueSimulator` with `msgValueSimulator` as a recipient then:

1. Alice (contract) transferred funds to the `msgValueSimulator`
2. `msgValueSimulator` reenter itself with a changed register:
3. value = rawFatPointer
4. isSystemCall = isSystemCall (was set by Alice)
5. to = Alice.address
6. Alice reenters self contract with the same `calldata` as was sent to the `msgValueSimulator` and `value == rawFatPointer`.
7. Alice sends `rawFatPointer` wei to herself.

> Please note the `fatPointer` is the struct:

```
pub struct FatPointer {
    pub offset: u32,
    pub memory_page: u32,
    pub start: u32,
    pub length: u32,
}
```

> And its raw representation:

```
rawFatPointer = length || start || memory_page || offset
```

> Depending on the use case, a user could manipulate the `msg.value` of the reentrant call. However if `length > 0`, the `rawFatPointer = msg.value >=`

> `2^96`. So if an attacker manipulates `length`, the result `msg.value` will be very large, so the attack is realistically impossible. Just as note, `2^96 wei == 79228162514 Ether == $100 trillion`.

> So the length of the data should be 0, but manipulating other data is still possible.

> I see the impact of a non-unauthorized call to itself fallback function. It is indeed pretty bad, even though I don't know any smart contract that would suffer from this in practice.

> All in all, I confirm the issue and appreciate that deep research, thanks a lot @ronnyx2017!

vladbochok (zkSync) commented:

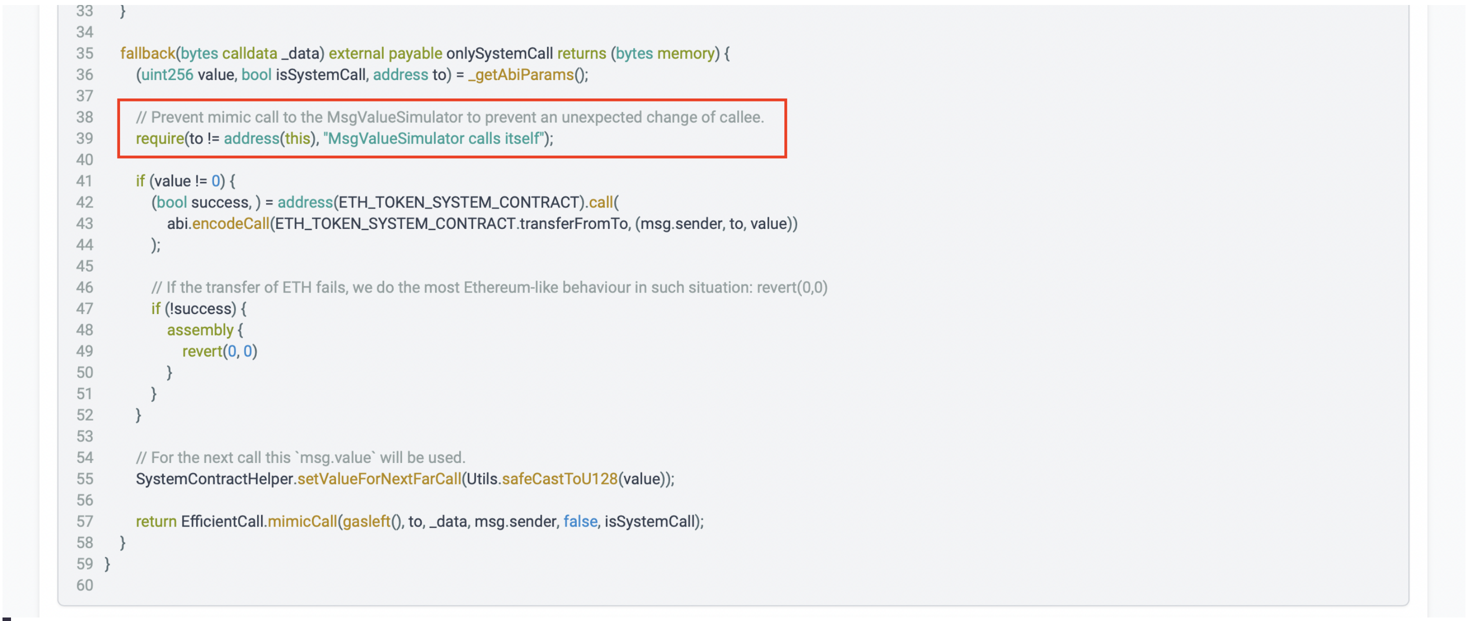
> For a note here is the test that we add to our `compiler-tester` to reproduce the issue.

```
pragma solidity ^0.8.0;

// The same copy of the system contracts that was on the scope.
import "./system-contracts/libraries/EfficientCall.sol";

contract Main {
    /// @dev The address of msgValueSimulator system contract.
    address constant MSG_VALUE_SIMULATOR_ADDRESS = address(0x80(


    /// @dev Number of times that fallback function was called.
    uint256 fallbackEntrantCounter;

    function test() external payable {
        // Reset counter, after the call to msgValueSimulator it
        fallbackEntrantCounter = 0;

        require(msg.value >= 2, "msg.value should be at least 2

        // The same pattern as on `DefaultAccount`
        bool success = EfficientCall.rawCall(gasleft(), MSG_VALU
        if (!success) {
            EfficientCall.propagateRevert();
        }
```

```
            require(fallbackEntrantCounter == 1, "Fallback function
        }

        fallback() external payable {
            fallbackEntrantCounter++;
         }
    }
```

[vladbochok (zkSync) commented](#):

> Last but not least, even though the impact of the issue wasn't clear to us after triaging the report, the fix was done immediately after the end of the audit, before the launch. So this (and others) issues are not in production.



```
    }

    fallback(bytes calldata _data) external payable onlySystemCall returns (bytes memory) {
        (uint256 value, bool isSystemCall, address to) = _getAbiParams();

        // Prevent mimic call to the MsgValueSimulator to prevent an unexpected change of callee.
        require(to != address(this), "MsgValueSimulator calls itself");

        if (value != 0) {
            (bool success, ) = address(ETH_TOKEN_SYSTEM_CONTRACT).call(
                abi.encodeCall(ETH_TOKEN_SYSTEM_CONTRACT.transferFromTo, (msg.sender, to, value))
            );

            // If the transfer of ETH fails, we do the most Ethereum-like behaviour in such situation: revert(0,0)
            if (!success) {
                assembly {
                    revert(0, 0)
                }
            }
        }

        // For the next call this `msg.value` will be used.
        SystemContractHelper.setValueForNextFarCall(Utils.safeCastToU128(value));

        return EfficientCall.mimicCall(gasleft(), to, _data, msg.sender, false, isSystemCall);
    }
}
```

> https://explorer.zksync.io/address/0x0000000000000000000000000000000000008009#contract

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

> Thank you @vladbochok for the extra detail and am glad this was already addressed.

> I do believe self-calling opens up to a category of exploits, especially for contracts that for example use try/catch or have "unusual" behaviour around transfers.

> I believe we can agree that the finding is unique and at least of medium severity -> Incorrect behaviour, which can conditionally lose funds.

> We must agree that the operation also can be viewed as account hijacking, in the sense that we can impersonate the receiving contract and then have it call itself.

> These lead me to believe that a higher severity should be appropriate.

> Have asked for advice to other judges with the goal of clarifying if there was sufficient information in the original submission, I believe the initial POC was valid but I want to get their perspective.

> Glad this was found and sorted.

[Alex the Entreprenerd (judge) commented]:

> While plenty of discussion has happened, the original finding has shown a valid POC that shows the following impact:

- By performing a call with value, we can forge a call that will cause the target contract to call itself

> The discussion after that helped demonstrate the report's validity and the Sponsor has already mitigated the potential risk.

> The ability for a specific contract to call self can be met with some skepticism in terms of its impact, however, I believe that in different scenarios, the severity would easily be raised to High.

> For example:

- Bridge contracts that call to self
- Contract that calls to self to use try/catch
- Vault Contracts, can be tricked into minting empty shares (no-op transfers), if the caller and the payer are the same (quirkiness of DAI)

> If those contracts were in-scope and the setup demonstrated in the finding was not patched, the finding would have easily been rated as High Severity.

> In this case, those contracts are not in-scope, so I would maintain a Medium Severity, because that's reliant on the specific integrators using that pattern.

> In contrast to `isSystem` which breaks an invariant on fully in-scope contracts, without the ability of causing damage, I have reason to believe that this specific vulnerability could have caused higher degrees of damage, for example:

- Contracts that allow to call or delegate call
- Vault Contracts as shown above
- Contracts where there is no expectation that the contract can call itself (as it may mess up the balance, accounting, etc..)

> Given the following considerations, I have asked myself whether this is a type of risk that would in any way be imputable to the integrator as a quirk, and at this time I cannot justify that.

> For the logic above, because the finding has shown a way to break a very strong expectation that a contract cannot call itself unless programmed for it, considering this as `msg.sender` spoofing, although limited to `self` calls, considering the potential risks for integrators, and the breaking of expectations for EVM systems, I am raising the finding to High Severity because I believe this would have not been a risk that the Sponsor would have wanted any user nor developer to take.

> The Sponsor has already mitigated the finding at the time of writing

Medium Risk Findings (5)

[M-01] deploying contracts with `forceDeployOnAddress` will break contracts when `callConstructor` is false

Submitted by [unforgiven](), *also found by* [Franfran](), [bin2chen](), [HE1M](), *and* [rvierdiiev]()

https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/ContractDeployer.sol#L212-L227
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/ContractDeployer.sol#L302-L306

When function `forceDeployOnAddress()` used for deploying contract and `callConstructor` is false, then contract's bytecodehash would stay in constructing state and calling the contract won't be possible. it can cause protocol and other contracts that are using it to break and if they call that address and sends some funds, then those funds would be lost. the issue is critical because the updated contract(which is updated by calling `forceDeployOnAddress()`) can be part of important process like bridging or sending messages or withdrawing funds.

Proof of Concept

This is `forceDeployOnAddress()` code in ContractDeployer:

```
    /// @notice The method that can be used to forcefully deploy
    /// @param _deployment Information about the forced deployme
    /// @param _sender The `msg.sender` inside the constructor c
    function forceDeployOnAddress(ForceDeployment calldata _depl
        _ensureBytecodeIsKnown(_deployment.bytecodeHash);
        _storeConstructingByteCodeHashOnAddress(_deployment.newA

        AccountInfo memory newAccountInfo;
        newAccountInfo.supportedAAVersion = AccountAbstractionVe
        // Accounts have sequential nonces by default.
        newAccountInfo.nonceOrdering = AccountNonceOrdering.Sequ
        _storeAccountInfo(_deployment.newAddress, newAccountInfo

        if (_deployment.callConstructor) {
            _constructContract(_sender, _deployment.newAddress,
        }

        emit ContractDeployed(_sender, _deployment.bytecodeHash,
    }
```

As you can see in the second line code calls `_storeConstructingByteCodeHashOnAddress()` and it would set constructing bytecode hash the address(the second byte is 1), and when `_deployment.callConstructor` is false, code won't call `_constructContract()` (which sets the address's bytecode hash as constructed after calling constructor) and contract bytecode hash would stay in constructing state after the deployment.

The constructing bytecode state is there to prevent other contracts to call this contract when this contract's constructor is called. Constructing state means that "if anyone else call the contract, it will behaves like a contract being constructing (EmptyContract/EOA)". so contract won't be callable if it stays in the Constructing state. This can cause serious issues, like this scenario:

If important contracts like L1Messenger, L2EthToken, MsgValueSimulator, … needed upgrade and their code upgraded with this deployer function(and admin didn't want to call constructor as initiating the contract again would break it), then the contract logics won't be callable by others but it would still behave like EmptyContract, the transaction won't revert and other contracts logics won't get interrupted but they don't work properly, for example user would spend funds(send to the updated contract) but because logics won't get executed the funds would be lost.

The real impact may be different based on the target contract that is being deployed by this function(when `callConstructor` =false), but in each time the contract deployment would break the contract and deployment would be faulted. the issue can happen every time and using this function to upgrade system contracts without calling constructor is common. for example imagine there is a contract, that have constructor that initialize the state. It is supposed to be called only once, for the first deployment. But protocol want to redeploy contract, that state of the contract is the same as before force deployment. So they want to skip the constructing phrase.

Tools Used

VIM

Recommended Mitigation Steps

Even when `callConstructor` is false, and code doesn't call the constructor, code should set the address's bytecode hash to Constructed state after the deployment.

miladpiri (zkSync) disagreed with severity and commented:

> The issue is real and fixed.

> The severity is **Medium**.

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

> The Warden has shown how, the `forceDeployOnAddress` function allows calling a contract in a way that can brick it, because the issue is notable but reliant on a mistake, while I have considered Low Severity (user mistake), I believe Medium Severity to be the most appropriate, because the system is not behaving in the intended way.

[M-02] User transactions can call system contracts directly

Submitted by Jeiwan*, also found by* ronnyx2017

User transaction can call system contracts directly, which shouldn't be allowed to not invoke potentially dangerous operations.

Proof of Concept

The DefaultAccount.executeTransaction executes a user transaction after it was validated. The function calls _execute under the hood. The `_execute` function makes two different calls depending on the destination address of a transaction:

1. if the `ContractDeployer` is called, it'll pass the call to the contract via the system call (`ContractDeployer` is a system contract and can be executed only via system calls);
2. if any other contract is called, it'll execute the call via `EfficientCall.rawCall`.

`EfficientCall.rawCall` in its turn also makes two different calls:

1. If `msg.value` of the transaction is 0, it'll make a regular call.
2. If there's some ETH sent with the transaction (i.e. `msg.value` is positive), it'll pass the call to the `MsgValueSimulator` contract. `MsgValueSimulator` is a system contract, thus the `isSystem` flag will be set in the far call ABI (notice the `true` in the last argument of `_loadFarCallABIIntoActivePtr`). However, it'll also set the forward mask to 1 (the value of MSG_VALUE_SIMULATOR_IS_SYSTEM_BIT). `MsgValueSimulator` will extract the mask and will set the `isSystemCall` flag to true—it'll then pass the `isSystemCall` flag to the subsequent call, making the call a system one.

To sum it up, if a transaction calls a contract that's not `ContractDeployer` and sends ETH, the call will be a system one, which will let it call the system contracts. However, users shouldn't be allowed to call system contracts directly to not invoke potentially dangerous operations. As per the documentation:

> Some of the system contracts can act on behalf of the user or have a very important impact on the behavior of the account. That's why we wanted to make it clear that users can not invoke potentially dangerous operations by doing a simple EVM-like call. Whenever a user wants to invoke some of the operations which we considered dangerous, they must explicitly provide isSystem flag with them.

However, since most system contracts are harmless, there's no direct high severity impact on the system, thus I think the issue is a medium severity.

Recommended Mitigation Steps

In the `EfficientCall.rawCall` function, consider setting the forward mask to 0. The behaviour of the function is similar to that of the `msgValueSimulatorMimicCall` function from the bootloader:

1. since the `MsgValueSimulator` contract is called, the `isSystemCall` flag should be set only for this call;
2. the `isSystemCall` flag should be forwarded by `MsgValueSimulator` only if the destination contract is `ContractDeployer`.

It looks that the second part of the `EfficientCall.rawCall` function was copied from the `SystemContractsCaller.systemCall` function, which is intended to call system contracts and which sets the forward mask to 1 when calling `MsgValueSimulator`. However, `rawCall` shouldn't forward the `isSystemCall` flag.

miladpiri (zkSync) confirmed and commented:

> It is an issue, though realisticially most of the methods are non-payable.

Alex the Entreprenerd (judge) commented:

> The Warden has shown a way to bypass the security checks that would prevent a end user to be able to call system contracts.

> In lack of a loss of deposits, I agree with Medium Severity.

[M-03] `DefaultAccount#fallback` lack payable

Submitted by [bin2chen](), *also found by* [minaminao]()

Fallback lack `payable` ,will lead to differences from the mainnet, and many existing protocols may not work.

Proof of Concept

DefaultAccount Defined as follows:

```
DefaultAccount

The implementation of the default account abstraction. This is t

Contains the minimal implementation of our account abstraction p
When anyone (except bootloader) calls/delegate calls it, it beha
```

If there is no code for the address, the DefaultAccount `#fallback` method will be executed, which is compatible with the behavior of the mainnet.

But at present, `fallback` is not `payable` .
The code is as follows

```
contract DefaultAccount is IAccount {
    using TransactionHelper for *;
..
    fallback() external { //<---------without payable
        // fallback of default account shouldn't be called by bo
        assert(msg.sender != BOOTLOADER_FORMAL_ADDRESS);

        // If the contract is called directly, behave like an EO
    }
    receive() external payable {
        // If the contract is called directly, behave like an E(
    }
```

Which will lead to differences from the mainnet.

For example, the mainnet execution of the method with value will return true, and the corresponding value will be transfer but DefaultAccount.sol will return false.

It is quite common for `call ()` to with `value`. If it is not compatible, many existing protocols may not work.

Mainnet code example, executing 0x0 call with value can be successful:

```
  function test() external {
    vm.deal(address(this),1000);
    console.log("before value:",address(0x0).balance);
    (bool result,bytes memory datas) = address(0x0).call{value:1
    console.log("call result:",result);
    console.log("after value:",address(0x0).balance);
  }
```

```
$ forge test -vvv

[PASS] test() (gas: 42361)
Logs:
  before value: 0
  call result: true
  after value: 10
```

Simulate DefaultAccount `#fallback` without payable, it will fail:

```
    contract DefaultAccount {
        fallback() external {
        }
        receive() external payable {
        }
    }

    function test() external {
        DefaultAccount defaultAccount = new DefaultAccount();
        vm.deal(address(this),1000);
        console.log("before value:",address(defaultAccount).bala
        (bool result,bytes memory datas) = address(defaultAccour
        console.log("call result:",result);
        console.log("after value:",address(defaultAccount).balar
    }
```

```
$ forge test -vvv

[PASS] test() (gas: 62533)
Logs:
  before value: 0
  call result: false
  after value: 0
```

Recommended Mitigation Steps

```
- function test() external {
+ function test() external payable {
    vm.deal(address(this),1000);
    console.log("before value:",address(0x0).balance);
    (bool result,bytes memory datas) = address(0x0).call{value:1
    console.log("call result:",result);
    console.log("after value:",address(0x0).balance);
  }
```

vladbochok (zkSync) confirmed

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, due to the lack of the `payable` modifer, contracts that sent value as well as a message would revert when triggering the `fallback`.

> While this could have been very severe for Smart Contracts, because the contract in question is for Account Abstraction, and that could break only certain transfers,

> I agree with Medium Severity.

[M-04] Time-sensitive contracts deployed on zkSync

Submitted by HE1M, also found by minaminao, rvierdiiev, and 0x73696d616f

Time-sensitive contracts will be impacted if deployed on zkSync.

Proof of Concept

Many contracts use `block.number` to measure the time as the miners were able to manipulate the `timestamp` (the `timestamp` could be easily gamed over short intervals). So, it was assumed that `block.number` is a safer and more accurate source of measuring time than `timestamp`.

For instance, if a defi project sets 144000 block interval to release the interest, it means approximately `144000 * 12 = 20 days`. Please note that each block in Ethereum takes almost 12 second.

If the same defi project is deployed on zkSync, it will not operate as expected. Because there is no time-bound for the blocks in zkSync (the interval may be 30 seconds or 1 week). So, the time to release the interest can be between 50 days to 2762 days.

Since, it is assumed that zkSync is Ethereum compatible, any deployed contracts on Ethereum may deploy their contract in zkSync without noting such big difference.

Even if the contracts use `timestamp` to measure the time, there will be another issue. In the contract `SystemContext.sol`, it is possible to set new block with the same `timestamp` as previous block, but with incremented block number.
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/SystemContext.sol#L116

In other words, new blocks are created but their time is frozen. Please note that freezing time can not be lasted for a long time, because when committing block their `timestamp` will be validated against a defined boundary.

Recommended Mitigation Steps

It should be explicitly mentioned that block intervals in zkSync are not compatible with Ethereum. So, time-sensitive contracts will be noted.

Moreover, the equal sign should be removed in the following line: https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/SystemContext.sol#L116

miladpiri (zkSync) confirmed and commented:

> Two issues are stated:

1. Block timestamp issue (which is duplicate of some other reports). dup #31 (**Low**)
2. Block creation rate is not consistent with Ethereum. (not duplicate) (**Medium**)

> The judges can decide better how to distinguish them.

Alex the Entreprenerd (judge) commented:

> The warden has shown how the zkEVM could differ from the EVM in how block timing is enforced, because blocks can happen at inconsistent times, protocols contract could have their time assumptions broken.

> Because this is a finding that has been addressed by other L2s, and causes inconsistent behaviour vs the EVM, I agree with Medium Severity.

[M-05] Losing fund during force deployment

Submitted by HE1M

During force deployment, if the fund is not properly transferred to the to-be-force-deployed contract, the fund will remain in the contract `ContractDeployer` and can not easily be recovered.

Proof of Concept

The function `forceDeployOnAddresses` in contract `ContractDeployer` is used only during an upgrade to set bytecodes on specific addresses.
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/ContractDeployer.sol#L232

The ETH sent to this function will be used to initialize to-be-force-deployed contracts. The ETH sent should be equal to the aggregated value needed for each contract.
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/ContractDeployer.sol#L240

Then the function externally calls itself, and send the required value to itself.
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/ContractDeployer.sol#L245

If any of this call is unsuccessful, the whole transaction will not revert, and the loop continues to deploy all the contract on the provided `newAddress`.

If for any reason, the deployment was not successful, the transferred ETH will remain in `ContractDeployer`, and can not be used for the next deployments (because the aggregated amount is compared with `msg.value` not the ETH balance of the contract). In other words, `FORCE_DEPLOYER` fund will be in `ContractDeployer`, and it can not be easily recoverred.

The possibility of unsuccessful deployment is not low:

It can happen if the bytecode is not known already.
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/ContractDeployer.sol#L213
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/ContractDeployer.sol#L296

It can happen during storing constructing bytecode hash.
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/ContractDeployer.sol#L214
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/AccountCodeStorage.sol#L36

It can happen during constructing contract and transferring the value.
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/ContractDeployer.sol#L223

Recommended Mitigation Steps

By using try/catch, the fund can be transferred to an address that the governor has control to be used later.

```
    function forceDeployOnAddresses(ForceDeployment[] calldata _depl
            external
            payable
        {
            // remaining of the code

            for (uint256 i = 0; i < deploymentsLength; ++i) {
                try
                    this.forceDeployOnAddress{value: _deployments[i]
                        _deployments[i],
                        msg.sender
                    )
                {} catch {
                    ETH_TOKEN_SYSTEM_CONTRACT.transferFromTo(
                        address(this),
                        SomeAddress,
                        _deployments[i].value
                    );
                }
            }
        }
```

Alex the Entreprenerd (judge) commented:

> Loss of funds due to reverts, keeping separate for now.

> See #95 for loss of value due to not calling constructor.

miladpiri (zkSync) confirmed and commented:

> The goal was to force deploy multiple of contracts (especially system contracts during the upgrade). If any deployment was unsuccessful, the whole transaction should not be reverted. So, the fund reverted during the failed deployment should be transferred to a valid address (not stay in `ContractDeployer`) as suggested by the warden. The recommended mitigation is also good.

> Severity is **Medium**.

Alex the Entreprenerd (judge) commented:

> Per the Sponsor's comment, the Warden has shown how, due to a lack of sweep on revert, funds sent in a sequence of multiple deployments can be lost when one of the deployment fails.

> This is in contrast to having the entire deployment reverting.

> Because the behavior is unintended and funds can be lost conditionally on a revert, I believe Medium Severity to be appropriate.

Low Risk and Non-Critical Issues

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

The following wardens also submitted reports: gjaldon, brgltd, joestakey, 0xSmartContract, supernova, Dravee, rbserver, minaminao, bshramin, Madalad, HE1M, *and* rvierdiiev.

[1]

Function `DefaultAccount._validateTransaction()` shouln't check `trx.value` for required balance, maybe user wanted the transaction to fail. also maybe paymaster is going to transfer required balance later.

```
        // The fact there is are enough balance for the account
        // should be checked explicitly to prevent user paying f
        // transaction that wouldn't be included on Ethereum.
        uint256 totalRequiredBalance = _transaction.totalRequire
        require(totalRequiredBalance <= address(this).balance, '
```

https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/DefaultAccount.sol#L102-L103

[2]

Function SystemContext.setGasPrice() comments are wrong.

```
    /// @notice Set the current tx origin.
    /// @param _gasPrice The new tx gasPrice.
    function setGasPrice(uint256 _gasPrice) external onlyBootloa
        gasPrice = _gasPrice;
    }
```

https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/SystemContext.sol#L64-L68

[3]

In functions `unsafeOverrideBlock()` and `setNewBlock()` of SystemContext, code should check that timestamp is less than `BLOCK_INFO_BLOCK_NUMBER_PART`, otherwise it can overflow and change the block number when combining them to calculate block info.

```
        // Setting new block number and timestamp
        currentBlockInfo = (currentBlockNumber + 1) * BLOCK_INFO
        currentBlockInfo = (number) * BLOCK_INFO_BLOCK_NUMBER_PA
```

https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/SystemContext.sol#L109-L128
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/SystemContext.sol#L132-L135

[4]

Function `BytecodeCompressor.publishCompressedBytecode()` shouldn't be payable as it doesn't have any logic for transferred ETH. If users send eth by mistake their funds would be lost.

```
    function publishCompressedBytecode(
        bytes calldata _bytecode,
        bytes calldata _rawCompressedData
    ) external payable returns (bytes32 bytecodeHash) {
```

https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/BytecodeCompressor.sol#L35-L38

[5]

Function `BytecodeCompressor.publishCompressedBytecode()` should check that the bytecode doesn't published before, it's possible to publish multiple compressed format for single bytecodes which can create issue for indexers and 3rd parties. If a bytecode has published before it's not necessary to publish it again. Also user or other protocol contract may lose funds if it calls this publish multiple times by mistake.
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/BytecodeCompressor.sol#L35-L68

[6]

The check for system deployer address in `DefaultAccount._validateTransaction()` is not correct as it convert `DEPLOYER_SYSTEM_CONTRACT` to uint256 and compare it to the `_transaction.to`. If `_transaction.to` was equal to `2^180 + DEPLOYER_SYSTEM_CONTRACT` then the check would be bypassed but the target address is in fact `DEPLOYER_SYSTEM_CONTRACT`. Code should convert `_transaction.to` to address and then check it with `DEPLOYER_SYSTEM_CONTRACT`.

```
        if (_transaction.to == uint256(uint160(address(DEPLOYER_
            require(_transaction.data.length >= 4, "Invalid call
        }
```

https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/DefaultAccount.sol#L95-L97

[7]

Code of `AccountCodeStorage.getCodeHash()` and `AccountCodeStorage.getCodeSize()` would return wrong values for inputs that are larger than `2^161-1` because code convert the input to uint160 and then return value for that address so if the input was bigger than max value of the uint160 then it when casting happens the calculated address can belong to another contract address. For example, there return value for `2^161 + Contract1_Address` will be the `Contract1_Address` information while there is no contract in `2^161 + Contract1_Address`. Code should make sure that the input uint256 is less than `2^161`.

```
    function getCodeSize(uint256 _input) external view override
        // We consider the account bytecode size of the last 20
        // according to the spec "If EXTCODESIZE of A is X, ther
        address account = address(uint160(_input));
        bytes32 codeHash = getRawCodeHash(account);
  ....
  ....
```

```
        function getCodeHash(uint256 _input) external view override
            // We consider the account bytecode hash of the last 20
            // according to the spec "If EXTCODEHASH of A is X, ther
            address account = address(uint160(_input));
            if (uint160(account) <= CURRENT_MAX_PRECOMPILE_ADDRESS)
                return EMPTY_STRING_KECCAK;
            }
    ....
    ....
```

https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/AccountCodeStorage.sol#L74-L77
https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/AccountCodeStorage.sol#L102-L106

[8]

Function `TransactionHelper.isEthToken()` should convert the input to the address and compare it to the `ETH_TOKEN_SYSTEM_CONTRACT`, in current implementation value `2^161 + ETH_TOKEN_SYSTEM_CONTRACT` would be not considered as ethToken but when it is converted to the uint160 and address it would be as `ETH_TOKEN_SYSTEM_CONTRACT`. Any logic depending on this function's return value can be vulnerable as it would be possible to supply uint256(Input1) that bypass `!isEthToken(Input1)` check but in fact the uint160(input1) is ETH token.

```
        function isEthToken(uint256 _addr) internal pure returns (bc
            return _addr == uint256(uint160(address(ETH_TOKEN_SYSTEM
        }
```

https://github.com/code-423n4/2023-03-zksync/blob/21d9a364a4a75adfa6f1e038232d8c0f39858a64/contracts/libraries/TransactionHelper.sol#L93-L95

miladpiri (zkSync) commented:

> Numbers 6, 7, 8 are interesting.
> They are good suggestions and useful, but mostly will not be implemented.

Alex the Entreprenerd (judge) commented:

1. Refactoring
2. Refactoring
3. Refactoring
4. Low
5. Refactoring
6. Refactoring
7. Refactoring
8. Low

Alex the Entreprenerd (judge) commented:

> Best report by far, going for strong impact and unique insights.

Note: the warden's downgraded findings were also considered when scoring. (for further details, see issues: 192, 187, 184, 180, 178, and 174)

Disclosures

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

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit 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.