# Polygon zkEVM Contracts Security Review


**Auditors**

Gerard Persoon, Lead Security Researcher

0xLeastwood, Lead Security Researcher

Csanuragjain, Security Researcher

Xiaoming90, Security Researcher

Pashov Krum, Associate Security Researcher

**Report prepared by:** Pablo Misirov



Report Version 1

March 27, 2023

# Contents

# 1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

# 2 Introduction

Smart contract implementation which will be used by the Polygon-Hermez zkEVM.

*Disclaimer*: This security review does not guarantee against a hack. It is a snapshot in time of zkEVM-Contracts according to the specific commit. Any modifications to the code will require a new security review.

# 3 Risk classification

| Severity level | Impact: High | Impact: Medium | Impact: Low |
|---|---|---|---|
| **Likelihood: high** | Critical | High | Medium |
| **Likelihood: medium** | High | Medium | Low |
| **Likelihood: low** | Medium | Low | Low |

## 3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

## 3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

## 3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

# 4 Executive Summary

Over the course of 13 days in total, Polygon engaged with Spearbit to review the zkevm-contracts protocol. In this period of time a total of **68** issues were found.

### Summary

| | |
|---|---|
| **Project Name** | Polygon |
| **Repository** | zkevm-contracts |
| **Commit** | 5de59e...f899 |
| **Type of Project** | Cross Chain, Bridge |
| **Audit Timeline** | Jan 9 - Jan 25 |
| **Two week fix period** | Jan 25 - Feb 8 |

### Issues Found

| Severity | Count | Fixed | Acknowledged |
|---|---|---|---|
| Critical Risk | 0 | 0 | 0 |
| High Risk | 0 | 0 | 0 |
| Medium Risk | 3 | 3 | 0 |
| Low Risk | 16 | 10 | 6 |
| Gas Optimizations | 19 | 18 | 1 |
| Informational | 30 | 19 | 11 |
| **Total** | **68** | **50** | **18** |

# 5 Findings

## 5.1 Medium Risk

### 5.1.1 Funds can be sent to a non existing destination

**Severity:** Medium Risk

**Context:** PolygonZkEVMBridge.sol#L129-L257

**Description:** The function **`bridgeAsset()`** and **`bridgeMessage()`** do check that the destination network is different than the current network. However, they don't check if the destination network exists. If accidentally the wrong **`networkId`** is given as a parameter, then the function is sent to a nonexisting network. If the network would be deployed in the future the funds would be recovered. However, in the meantime they are inaccessible and thus lost for the sender and recipient. Note: other bridges usually have validity checks on the destination.

```
function bridgeAsset(...) ... {
   require(destinationNetwork != networkID, ... );
   ...
}
function bridgeMessage(...) ... {
   require(destinationNetwork != networkID, ... );
   ...
}
```

**Recommendation:** Check the destination **`networkID`** is a valid destination.

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

### 5.1.2 Fee on transfer tokens

**Severity:** Medium Risk

**Context:** PolygonZkEVMBridge.sol#L171

**Description:** The bridge contract will not work properly with a fee on transfer tokens

1. User A bridges a fee on transfer Token A from Mainnet to Rollover R1 for amount X.
2. In that case **`X-fees`** will be received by bridge contract on Mainnet but the deposit receipt of the full amount **`X`** will be stored in Merkle.
3. The amount is claimed in R1 and a new TokenPair for Token A is generated and the full amount **`X`** is minted to User A
4. Now the full amount is bridged back again to Mainnet
5. When a claim is made on Mainnet then the contract tries to transfer amount **`X`** but since it received the amount **`X-fees`** it will use the amount from other users, which eventually causes DOS for other users using the same token

**Recommendation:** Use the exact amount which is transferred to the contract which can be obtained using below sample code

```
uint256 balanceBefore = IERC20Upgradeable(token).balanceOf(address(this));
IERC20Upgradeable(token).safeTransferFrom(address(msg.sender), address(this), amount);
uint256 balanceAfter = IERC20Upgradeable(token).balanceOf(address(this));
uint256 transferedAmount = balanceAfter - balanceBefore;

// if you dont want to support fee on transfer token use below:
require (transferedAmount == amount, ...);

// use transferedAmount if you want to support fee on transfer token
```

**Polygon-Hermez:** Solved in PR 87. To protect against reentrancy with erc777 tokens, a check for reentrancy MUST be added.

Solved in PR 91.

**Spearbit:** Verified.

#### 5.1.3 Function `consolidatePendingState()` can be executed during emergency state

**Severity:** Medium Risk

**Context:** PolygonZkEVM.sol#L783-L793

**Description:** The function **`consolidatePendingState()`** can be executed by everyone even when the contract is in an emergency state. This might interfere with cleaning up the emergency.

Most other functions are disallowed during an emergency state.

```
function consolidatePendingState(uint64 pendingStateNum) public {
    if (msg.sender != trustedAggregator) {
        require(isPendingStateConsolidable(pendingStateNum),...);
    }
    _consolidatePendingState(pendingStateNum);
}
```

**Recommendation:** Consider adding the following, which also improves consistency

```
function consolidatePendingState(uint64 pendingStateNum) public {
    if (msg.sender != trustedAggregator) {
+       require(!isEmergencyState,...);
        require(isPendingStateConsolidable(pendingStateNum),...);
    }
    _consolidatePendingState(pendingStateNum);
}
```

**Polygon-Hermez:** Solved in PR 87.

**Spearbit:** Verified.

## 5.2 Low Risk

### 5.2.1 Sequencers can re-order forced and non-forced batches

**Severity:** Low Risk

**Context:** PolygonZkEVM.sol#L409-L533

**Description:** Sequencers have a certain degree of control over how non-forced and forced batches are ordered.

Consider the case where we have two sets of batches; non-forced (NF) and forced (F). A sequencer can order the following sets of batches (F1, F2) and (NF1, NF2) in any order so as long as the order of the forced batch and non-forced batch sets are kept in order.

i.e. A sequencer can sequence batches as **`F1 -> NF1 -> NF2 -> F2`** but they can also equivalently sequence these same batches as **`NF1 -> F1 -> F2 -> NF2`**.

**Recommendation:** Ensure this behavior is understood and consider what impact decentralizing the sequencer role would have on bridge users.

**Polygon-Hermez:** Added comments to function **`forceBatch()`** in PR 85.

**Spearbit:** Acknowledged.

### 5.2.2 Check length of `smtProof`

**Severity:** Low Risk

**Context:** DepositContract.sol#L90-L112

**Description:** An obscure Solidity bug could be triggered via a call in solidity 0.4.x. Current solidity versions revert with panic **`0x41`**. The problem could occur if unbounded memory arrays were used. This situation happens to be the case as **`verifyMerkleProof()`** (and all the functions that call it) don't check the length of the array (or loop over the entire array). It also depends on memory variables (for example structs) being used in the functions, that doesn't seem to be the case.

Here is a POC of the issue which can be run in remix

```
// SPDX-License-Identifier: MIT
// based on
↪  https://github.com/paradigm-operations/paradigm-ctf-2021/blob/master/swap/private/Exploit.sol
pragma solidity ^0.4.24; // only works with low solidity version
import "hardhat/console.sol";

contract test{
    struct Overlap {
        uint field0;
    }

  function mint(uint[] memory amounts) public {
      Overlap memory v;
      console.log("before: ",amounts[0]);
      v.field0 = 567;
      console.log("after: ",amounts[0]); // would expect to be 0 however is 567
    }

  function go() public { // this part requires the low solidity version
      bytes memory payload = abi.encodeWithSelector(this.mint.selector, 0x20, 2**251);
      bool success = address(this).call(payload);
      console.log(success);
  }
}
```

**Recommendation:** Although it currently isn't exploitable, to be sure consider adding a check on the length of **`smtProof`**;

```
function verifyMerkleProof..., bytes32[] memory smtProof, ...) public pure returns (bool) {
+   require (smtProof.length == _DEPOSIT_CONTRACT_TREE_DEPTH);
    ...
}
```

Or use a fixed-size array.

```
function verifyMerkleProof(..., bytes32[_DEPOSIT_CONTRACT_TREE_DEPTH] memory smtProof,... ) ... {
}
```

**Polygon-Hermez:** Solved in PR 85.

**Spearbit:** Verified.

#### 5.2.3 Transaction delay due to free `claimAsset()` transactions

**Severity:** Low Risk

**Context:** batchbuilder.go#L29-L122

**Description:** The sequencer first processes the free **`claimAsset()`** transaction and then the rest. This might delay other transactions if there are many free **`claimAsset()`** transactions. As these transactions would have to be initiated on the mainnet, the gas costs there will reduce this problem. However, once multiple rollups are supported in the future the transactions could originate from another rollup with low gas costs.

```
func (s *Sequencer) tryToProcessTx(ctx context.Context, ticker *time.Ticker) {
    ...
    appendedClaimsTxsAmount := s.appendPendingTxs(ctx, true, 0, getTxsLimit, ticker)  // `claimAsset()`
    ↪  transactions
    appendedTxsAmount := s.appendPendingTxs(ctx, false, minGasPrice.Uint64(),
    ↪  getTxsLimit-appendedClaimsTxsAmount, ticker) + appendedClaimsTxsAmount
    ...
}
```

**Recommendation:** Consider adding a limited amount of free **`claimAsset()`** transactions to a batch.

**Polygon-Hermez:** This is important to consider once multiple rollups are implemented. For now, we'll let it be like this.

**Spearbit:** Acknowledged.

#### 5.2.4 Misleading token addresses

**Severity:** Low Risk

**Context:** PolygonZkEVMBridge.sol#L272-L373

**Description:** The function **`claimAsset()`** deploys **`TokenWrapped`** contracts via **`create2`** and a **`salt`**. This **`salt`** is based on the **`originTokenAddress`**. By crafting specific **`originTokenAddress`**es, it's possible to create vanity addresses on the other chain. These addresses could be similar to legitimate tokens and might mislead users. Note: it is also possible to directly deploy tokens on the **`other`** chain with vanity addresses (e.g. without using the bridge)

```
function claimAsset(...) ... {
    ...
    bytes32 tokenInfoHash = keccak256(abi.encodePacked(originNetwork, originTokenAddress));
    ...
    TokenWrapped newWrappedToken = (new TokenWrapped){ salt: tokenInfoHash }(name, symbol, decimals);
    ...
}
```

**Recommendation:** Have a way for users to determine legitimate bridged tokens, for example via tokenlists.

**Polygon-Hermez:** Our front end will show a list of supported tokens, and the users will be able also to import custom tokens too ( the same way it happens in a lot of DEXs).

But we cannot control of course the use of all the dapps deployed. Dapps that integrate with our system should be able to import our token list or calculate them. There are view functions to help calculate an L2 address **`precalculatedWrapperAddress`** given a mainnet token address and metadata. Also easily a user/dapp can check the corresponding L1 token address of every token of L2 calling **`wrappedTokenToTokenInfo`** and checking **`originTokenAddress`** is in their Mainnet token list for example. Our front end will show a list of supported tokens, the users will be able also to import custom tokens too ( the same way it happens in a lot of DEXs). But we cannot control of course the use of all the dapps deployed. Dapps that integrate with our system should be able to import our token list or calculate them. There's view functions to help calculate a L2 address **`precalculatedWrapperAddress`** given a mainnet token address and metadata. Also easily a user/dapp can check the corresponding L1 token address of every token of L2 calling **`wrappedTokenToTokenInfo`** and checking **`originTokenAddress`** is in their Mainnet token list for example.

**Spearbit:** Acknowledged.

#### 5.2.5 Limit amount of gas for free `claimAsset()` transactions

**Severity:** Low Risk

**Context:** PolygonZkEVMBridge.sol#L272-L436

**Description:** Function **`claimAsset()`** is subsidized (e.g. gasprice is 0) on L2 and allows calling a custom contract. This could be misused to execute elaborate transactions for free. Note: **`safeTransfer`** could also call a custom contract that has been crafted before and bridged to L1. Note: this is implemented in the **`Go`** code, which detects transactions to the bridge with function **`bridgeClaimMethodSignature == "0x7b6323c1"`**, which is the selector of **`claimAsset()`**. See function **`IsClaimTx()`** in transaction.go.

```
function claimAsset(...) ... {
    ...
    (bool success, ) = destinationAddress.call{value: amount}(new bytes(0));
    ...
    IERC20Upgradeable(originTokenAddress).safeTransfer(destinationAddress,amount);
    ...
}
```

**Recommendation:** Limit the amount of gas supplied to **`claimAsset()`** while doing free transactions.

Note: A PR is being made to limit the available gas: PR 1551.

**Polygon-Hermez:** We will take this into consideration in the current and future implementation of the sequencer.

**Spearbit:** Acknowledged.

#### 5.2.6 What to do with funds that can't be delivered

**Severity:** Low Risk

**Context:** PolygonZkEVMBridge.sol#L272-L436

**Description:** Both **`claimAsset()`** and **`claimMessage()`** might **`revert`** on different locations (even after retrying). Although the funds stay in the bridge, they are not accessible by the originator or recipient of the bridge action. So they are essentially lost for the originator and recipient. Some other bridges have recovery addresses where the funds can be delivered instead.

Here are several potential **`revert`** situations:

```
function claimAsset(...) ... {
    ...
    (bool success, ) = destinationAddress.call{value: amount}(new bytes(0));
    require(success, ... );
    ...
    IERC20Upgradeable(originTokenAddress).safeTransfer(destinationAddress,amount);
    ...
    TokenWrapped newWrappedToken = (new TokenWrapped){ salt: tokenInfoHash }(name, symbol, decimals);
    ...
}
function claimMessage(...) ... {
     ...
     (bool success, ) = destinationAddress.call{value: amount}(
          abi.encodeCall(
               IBridgeMessageReceiver.onMessageReceived,
               (originAddress, originNetwork, metadata)
          )
     );
     require(success, "PolygonZkEVMBridge::claimMessage: Message failed");
     ...
}
```

**Recommendation:** Consider having a recovery mechanism for funds that can't be delivered. For example, the funds could be delivered to an EOA recovery address on a destination chain. This requires adding more info in the merkle root and also needs to check those senders add a recovery address, otherwise, there isn't always a recovery address. It could be implemented via a function (e.g. **`recoverAsset()`**) where the recovery address initiates a transaction (similar to **`claimAsset()`** ) and the funds are delivered to the recovery address.

**Polygon-Hermez:** We decided that such a mechanism is not worth implementing since the user can also put an invalid recoverAddress and it adds too much overhead. If the user put an invalid address as a destination address, either if it was mistaken or puts a smart contract that cannot receive funds, the funds will be lost.

**Spearbit:** Acknowledged.

#### 5.2.7 Inheritance structure does not openly support contract upgrades

**Severity:** Low Risk

**Context:** PolygonZkEVMBridge.sol#L18-L21

**Description:** The solidity compiler uses C3 linearisation to determine the order of contract inheritance. This is performed as left to right of all child contracts before considering the parent contract.

Storage slot assignment is as follows: **`Initializable`** -> **`DepositContract`** -> **`EmergencyManager`** -> **`PolygonZkEVMBridge`**

The **`Initializable.sol`** already reserves storage slots for future upgrades and because **`PolygonZkEVMBridge.sol`** is inherited last, storage slots can be safely appended. However, the two intermediate contracts, **`DepositContract.sol`** and **`EmergencyManager.sol`**, cannot handle storage upgrades.

**Recommendation:** Consider introducing a storage gap to the two intermediate contracts.

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

### 5.2.8 Function `calculateRewardPerBatch()` could divide by 0

**Severity:** Low Risk

**Context:** PolygonZkEVM.sol#L1488-L1498

**Description:** The function **`calculateRewardPerBatch()`** does a division by **`totalBatchesToVerify`**. If there are currently no batches to verify, then **`totalBatchesToVerify`** would be **`0`** and the transaction would revert.

When **`calculateRewardPerBatch()`** is called from **`_verifyBatches()`** this doesn't happen as it will revert earlier. However when the function is called externally this situation could occur.

```
function calculateRewardPerBatch() public view returns (uint256) {
    ...
    uint256 totalBatchesToVerify = ((lastForceBatch - lastForceBatchSequenced) + lastBatchSequenced) -
↪   getLastVerifiedBatch();
    return currentBalance / totalBatchesToVerify;
}
```

**Recommendation:** Consider changing the code to something like

```
function calculateRewardPerBatch() public view returns (uint256) {
    ...
    uint256 totalBatchesToVerify = ((lastForceBatch - lastForceBatchSequenced) + lastBatchSequenced) -
    ↪  getLastVerifiedBatch();
+   if (totalBatchesToVerify == 0) return 0;
    return currentBalance / totalBatchesToVerify;
}
```

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

### 5.2.9 Limit gas usage of `_updateBatchFee()`

**Severity:** Low Risk

**Context:** PolygonZkEVM.sol#L839-L908

**Description:** The function **`_updateBatchFee()`** loops through all unverified batches. Normally this would be 30 min/5 min ~ 6 batches.

Assume the aggregator malfunctions and after one week, **`verifyBatches()`** is called, which calls **`_updateBatchFee()`**. Then there could be 7 * 24 * 60 min/ 5 min ~ 2352 batches. The function **`verifyBatches()`** limits this to **`MAX_VERIFY_BATCHES == 1000`**. This might result in an out-of-gas error. This would possibly require multiple **`verifyBatches()`** tries with a smaller number of batches, which would increase network outage.

```
function _updateBatchFee(uint64 newLastVerifiedBatch) internal {
    ...
    while (currentBatch != currentLastVerifiedBatch) {
        ...
        if (block.timestamp - currentSequencedBatchData.sequencedTimestamp >veryBatchTimeTarget) {
            ...
        }
        ...
    }
    ...
}
```

**Recommendation:** Optimize the function **`_updateBatchFee()`** and/or check the available amount of gas and simplify the algorithm to update the **`batchFee`**.

As suggested by the project, the following optimization below could be made. This will alleviate the problem because only a limited amount of batches will be below target. The bulk will be above target, but they won't be looped over.

```
// Check if timestamp is above or below the VERIFY_BATCH_TIME_TARGET
if (
    block.timestamp - currentSequencedBatchData.sequencedTimestamp < // Notice the changed <
    veryBatchTimeTarget
) {
    totalBatchesBelowTarget += // Notice now it's Below
        currentBatch -
        currentSequencedBatchData.previousLastBatchSequenced;
}
else {
    break; // Since the rest of batches will be above! ^^
}
```

**Polygon-Hermez:** Solved in PR 90.

**Spearbit:** Verified.

#### 5.2.10 Keep precision in `_updateBatchFee()`

**Severity:** Low Risk

**Context:** PolygonZkEVM.sol#L839-L908

**Description:** Function **`_updateBatchFee()`** uses a trick to prevent losing precision in the calculation of **`accDivisor`**. The value **`accDivisor`** includes an extra multiplication with **`batchFee`**, which is undone when doing **`batchFee = (batchFee * batchFee) / accDivisor`** because this also contains an extra multiplication by **`batchFee`**.

However, if **`batchFee`** happens to reach a small value (also see issue **`Minimum and maximum value for batchFee`**) then the trick doesn't work that well. In the extreme case of **`batchFee ==0`** then a division by **`0`** will take place, resulting in a **`revert`**. Luckily this doesn't happen in practice.

```
function _updateBatchFee(uint64 newLastVerifiedBatch) internal {
    ...
    uint256 accDivisor = (batchFee * (uint256(multiplierBatchFee) ** diffBatches)) / (10 **
↪   (diffBatches * 3));
    batchFee = (batchFee * batchFee) / accDivisor;
    ...
}
```

**Recommendation:** Replace the multiplication factor with a fixed and sufficiently large value. For example in the following way:

```
function _updateBatchFee(uint64 newLastVerifiedBatch) internal {
    ...
-    uint256 accDivisor = (batchFee * (uint256(multiplierBatchFee) ** diffBatches)) / (10 **
↪   (diffBatches * 3));
-    batchFee = (batchFee * batchFee) / accDivisor;
+    uint256 accDivisor = (1E18 * (uint256(multiplierBatchFee) ** diffBatches)) / (10 ** (diffBatches *
↪   3));
+    batchFee = (1E18 * batchFee) / accDivisor;
    ...
}
```

**Polygon-Hermez:** Solved in PR 90.

**Spearbit:** Verified.

### 5.2.11 Minimum and maximum value for `batchFee`

**Severity:** Low Risk

**Context:** PolygonZkEVM.sol#L839-L908

**Description:** Function **`_updateBatchFee()`** updates the **`batchFee`** depending on the batch time target. If the batch times are repeatedly below or above the target, the **`batchFee`** could shrink or grow unlimited.

If the **`batchFee`** would get too low, problems with the economic incentives might arise.

If the **`batchFee`** would get too high, overflows might occur. Also, the fee might too be high to be practically payable. Although not very likely to occur in practice, it is probably worth the trouble to implement limits.

```
function _updateBatchFee(uint64 newLastVerifiedBatch) internal {
    ...
    if (totalBatchesBelowTarget < totalBatchesAboveTarget) {
        ...
        batchFee = (batchFee * (uint256(multiplierBatchFee) ** diffBatches)) / (10 ** (diffBatches *
↪  3));
    } else {
        ...
        uint256 accDivisor = (batchFee * (uint256(multiplierBatchFee) ** diffBatches)) / (10 **
↪  (diffBatches * 3));
        batchFee = (batchFee * batchFee) / accDivisor;
    }
}
```

**Recommendation:** Consider implementing a minimum and maximum value for **`batchFee`**.

**Polygon-Hermez:** Solved in PR 90.

**Spearbit:** Verified.

### 5.2.12 Bridge deployment will fail if `initialize()` is front-run

**Severity:** Low Risk

**Context:** deployContracts.js

**Description:** By default, Hardhat will deploy transparent upgradeable proxies when calling **`upgrades.deployProxy()`** with no type specified. This function accepts data which is used to initialize the state of the contract being deployed. However, because the zkEVM bridge script utilizes the output of each contract address on deployment, it is not trivial to atomically deploy and initialize contracts. As a result, there is a small time window available for attackers to front-run calls to initialize the necessary bridge contracts, allowing them to temporarily DoS during the deployment process.

**Recommendation:** Consider pre-calculating all contract addresses prior to proxy deployment so each contract can be initialized atomically.

**Polygon-Hermez:** We are currently deciding the best way to make a deterministic deployment for the bridge but will take this in account for sure.

**Spearbit:** Acknowledged.

### 5.2.13 Add input validation for the `setVeryBatchTimeTarget` method

**Severity:** Low Risk

**Context:** PolygonZkEVM.sol#L1171-L1176

**Description:** The **`setVeryBatchTimeTarget`** method in **`PolygonZkEVM`** accepts a **`uint64 newVeryBatchTimeTarget`** argument to set the **`veryBatchTimeTarget`**. This variable has a value of **`30 minutes`** in the **`initialize`** method, so it is expected that it shouldn't hold a very big value as it is compared to timestamps difference in **`_updateBatchFee`**. Since there is no upper bound for the value of the **`newVeryBatchTimeTarget`** argument, it is possible (for example due to fat-fingering the call) that an admin passes a big value (up to **`type(uint64).max`**) which will result in wrong calculation in **`_updateBatchFee`**.

**Recommendation:** Add a sensible upper bound for the value of **`newVeryBatchTimeTarget`**, for example, **`1 day`**.

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

### 5.2.14 Single-step process for critical ownership transfer

**Severity:** Low Risk

**Context:** PolygonZkEVM.sol#L1182, OwnableUpgradeable.sol#L74

**Description:** If the nominated **`newAdmin`** or **`newOwner`** account is not a valid account, the owner or admin risks locking themselves out.

```
function setAdmin(address newAdmin) public onlyAdmin {
    admin = newAdmin;

    emit SetAdmin(newAdmin);
}
```

```
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _transferOwnership(newOwner);
}
```

**Recommendation:** Consider implementing a two-step process where the owner or admin nominates an account, and the nominated account calls an **`acceptTransfer()`** function for the transfer to succeed.

**Polygon-Hermez:** Solved for Admin in PR 87. The owner is intended to be a **`multisig`** in a bootstrap period, and afterward will be renounced. We consider **`acceptTransfer()`** to be an overkill for this address.

**Spearbit:** Verified and Acknowledged.

### 5.2.15 Ensure no native asset value is sent in `payable` method that can handle ERC20 transfers as well

**Severity:** Low Risk

**Context:** PolygonZkEVMBridge.sol#L154-L187

**Description:** The **`bridgeAsset`** method of **`PolygonZkEVMBridge`** is marked **`payable`** as it can work both with the native asset as well as with ERC20 tokens. In the codepath where it is checked that the token is not the native asset but an ERC20 token, it is not validated that the user did not actually provide value to the transaction. The likelihood of this happening is pretty low since it requires a user error but if it does happen then the native asset value will be stuck in the **`PolygonZkEVMBridge`** contract.

**Recommendation:** Ensure that no native asset value is sent when the bridged asset is an ERC20 token by adding the following code to the codepath of the ERC20 bridging:

```
require(msg.value == 0, "PolygonZkEVMBridge::bridgeAsset: Expected zero native asset value when
↪  bridging ERC20 tokens");
```

You can also use an **`if`** statement and a custom error instead of a **`require`** statement.

**Polygon-Hermez:** Solved in PR 87.

**Spearbit:** Verified.

#### 5.2.16 Calls to the `name`, `symbol` and `decimals` functions will be unsafe for non-standard ERC20 tokens

**Severity:** Low Risk

**Context:** PolygonZkEVMBridge.sol#L181-185

**Description:** The **`bridgeAsset`** method of **`PolygonZkEVMBridge`** accepts an **`address token`** argument and later calls the **`name`**, **`symbol`** and **`decimals`** methods of it. There are two potential problems with this:

1. Those methods are not mandatory in the ERC20 standard, so there can be ERC20-compliant tokens that do not have either or all of the **`name`**, **`symbol`** or **`decimals`** methods, so they will not be usable with the protocol, because the calls will revert
2. There are tokens that use **`bytes32`** instead of **`string`** as the value type of their **`name`** and **`symbol`** storage variables and their getter functions (example is MKR). This can cause reverts when trying to consume metadata from those tokens.

Also, see weird-erc20 for nonstandard tokens.

**Recommendation:** For the first problem, a simple solution is to use a low-level **`staticcall`** for those method calls and if they are unsuccessful to use some default values. For the second problem it would be best to again do a low-level **`staticcall`** and then use an external library that checks if the returned data is of type **`string`** and if not to cast it to such.

Also, see MasterChefJoeV3 as an example for a possible solution.

**Polygon-Hermez:** Solved in PR 90 and PR 91.

**Spearbit:** Verified.

### 5.3 Gas Optimization

#### 5.3.1 Use `calldata` instead of `memory` for array parameters

**Severity:** Gas Optimization

**Context:** PolygonZkEVMBridge.sol#L272-L373, PolygonZkEVMBridge.sol#L520-L581, DepositContract.sol#L90-L112

**Description:** The code frequently uses **`memory`** arrays for externally called functions. Some gas could be saved by making these **`calldata`**. The **`calldata`** can also be cascaded to **`internal`** functions that are called from the **`external`** functions.

```
function claimAsset(bytes32[] memory smtProof) public {
    ...
    _verifyLeaf(smtProof);
    ...
}
function _verifyLeaf(bytes32[] memory smtProof) internal {
    ...
    verifyMerkleProof(smtProof);
    ...
}
function verifyMerkleProof(..., bytes32[] memory smtProof, ...) internal {
    ...
}
```

**Recommendation:** Consider changing the code to

```
-function claimAsset(bytes32[] memory   smtProof) public {
+function claimAsset(bytes32[] calldata smtProof) public {
    ...
    _verifyLeaf(smtProof);
    ...
}
-function _verifyLeaf(bytes32[] memory   smtProof) internal {
+function _verifyLeaf(bytes32[] calldata smtProof) internal {
    ...
    verifyMerkleProof(smtProof);
    ...
}
-function verifyMerkleProof(..., bytes32[] memory   smtProof, ...) internal {
+function verifyMerkleProof(..., bytes32[] calldata smtProof, ...) internal {
    ...
}
```

**Polygon-Hermez:** Solved in PR 85.

**Spearbit:** Verified.

#### 5.3.2 Optimize `networkID == MAINNET_NETWORK_ID`

**Severity:** Gas Optimization

**Context:** PolygonZkEVMBridge.sol#L520-L581, PolygonZkEVMBridge.sol#L69-L77

**Description:** The value for **`networkID`** is defined in **`initialize()`** and **`MAINNET_NETWORK_ID`** is constant. So **`networkID == MAINNET_NETWORK_ID`** can be calculated in **`initialize()`** and stored to save some gas. It is even cheaper if **`networkID`** is immutable, which would require adding a **`constructor`**.

```
uint32 public constant MAINNET_NETWORK_ID = 0;
uint32 public networkID;
function initialize(uint32 _networkID, ...) public virtual initializer {
    networkID = _networkID;
    ...
}
function _verifyLeaf(...) ... {
    ...
    if (networkID == MAINNET_NETWORK_ID) {
        ...
    } else {
        ...
    }
}
```

**Recommendation:** Consider calculating **`networkID == MAINNET_NETWORK_ID`** in **`initialize()`** or preferable a **`constructor`**.

**Polygon-Hermez:** We decided to go for a deterministic deployment for the **`PolygonZkEVMBridge`** contract, therefore we prefer to not use **`immutable`**s in this contract. When implemented without **`immutable`**, the resulting code is less optimal.

**Spearbit:** Acknowledged.

### 5.3.3 Optimize `updateExitRoot()`

**Severity:** Gas Optimization

**Context:** PolygonZkEVMGlobalExitRoot.sol#L54-L75

**Description:** The function **`updateExitRoot()`** accesses the global variables **`lastMainnetExitRoot`** and **`lastRollupExitRoot`** multiple times. This can be optimized using temporary variables.

```
function updateExitRoot(bytes32 newRoot) external {
    ...
    if (msg.sender == rollupAddress) { lastRollupExitRoot = newRoot; }
    if (msg.sender == bridgeAddress) { lastMainnetExitRoot = newRoot; }
    bytes32 newGlobalExitRoot = keccak256( abi.encodePacked(lastMainnetExitRoot, lastRollupExitRoot) );
    if ( ... ) {
        ...
        emit UpdateGlobalExitRoot(lastMainnetExitRoot, lastRollupExitRoot);
    }
}
```

**Recommendation:** Consider changing the code to something like

```
function updateExitRoot(bytes32 newRoot) external {
    ...
    bytes32 cm = lastMainnetExitRoot;
    bytes32 cr = lastRollupExitRoot;
    if (msg.sender == rollupAddress) { lastRollupExitRoot = cr = newRoot; }
    if (msg.sender == bridgeAddress) { lastMainnetExitRoot = cm = newRoot; }
    bytes32 newGlobalExitRoot = keccak256( abi.encodePacked(cm, cr) );
    if ( ... ) {
        ...
        emit UpdateGlobalExitRoot(cm, cr);
    }
}
```

**Polygon-Hermez:** Solved in PR 82.

**Spearbit:** Verified.

### 5.3.4 Optimize `_setClaimed()`

**Severity:** Gas Optimization

**Context:** PolygonZkEVMBridge.sol#L272-L436, PolygonZkEVMBridge.sol#L520-L581, PolygonZkEVMBridge.sol#L587-L605

**Description:** The function **`claimAsset()`** and **`claimMessage()`** first verify **`!isClaimed()`** (via the function **`_verifyLeaf()`**) and then do **`_setClaimed()`**. These two functions can be combined in a more efficient version.

```
function claimAsset(...) ... {
    _verifyLeaf(...);
    _setClaimed(index);
    ...
}
function claimMessage(...) ... {
    _verifyLeaf(...);
    _setClaimed(index);
    ...
}
function _verifyLeaf(...) ... {
    require( !isClaimed(index), ...);
    ...
}
function isClaimed(uint256 index) public view returns (bool) {
    uint256 claimedWordIndex = index / 256;
    uint256 claimedBitIndex = index % 256;
    uint256 claimedWord = claimedBitMap[claimedWordIndex];
    uint256 mask = (1 << claimedBitIndex);
    return (claimedWord & mask) == mask;
}
function _setClaimed(uint256 index) private {
    uint256 claimedWordIndex = index / 256;
    uint256 claimedBitIndex = index % 256;
    claimedBitMap[claimedWordIndex] = claimedBitMap[claimedWordIndex] | (1 << claimedBitIndex);
}
```

**Recommendation:** The following suggestion is based on Uniswap permit2 bitmap:

Replace the duo of **`isClaimed()`** and **`_setClaimed()`** with the following function **`_setAndCheckClaimed()`**. This could be either inside or outside of _verifyLeaf().

```
function _setAndCheckClaimed(uint256 index) private {
    (uint256 wordPos, uint256 bitPos) = _bitmapPositions(index);
    uint256 mask = 1 << bitPos;
    uint256 flipped = claimedBitMap[wordPos] ^= mask;
    require (flipped & mask != 0,"PolygonZkEVMBridge::_verifyLeaf: Already claimed");
}
function _bitmapPositions(uint256 index) private pure returns (uint256 wordPos, uint256 bitPos) {
    wordPos = uint248(index >> 8);
    bitPos = uint8(index);
}
```

Note: update the error message, depending on whether it will be called inside or outside of _verifyLeaf()

If the status of the bits has to be retrieved from the outside, add the following function:

```
function isClaimed(uint256 index) external view returns (bool) {
    (uint256 wordPos, uint256 bitPos) = _bitmapPositions(index);
    uint256 mask = (1 << bitPos);
    return ( claimedBitMap[wordPos] & mask) == mask;
}
```

**Polygon-Hermez:** Solved in PR 82.

**Spearbit:** Verified.

### 5.3.5 SMT branch comparisons can be optimised

**Severity:** Gas Optimization

**Context:** DepositContract.sol#L99-L109, DepositContract.sol#L65-L77, DepositContract.sol#L30-L46

**Description:** When verifying a merkle proof, the search does not terminate until we have iterated through the tree depth to calculate the merkle root. The path is represented by the lower 32 bits of the **`index`** variable where each bit represents the direction of the path taken.

Two changes can be made to the following snippet of code:

- Bit shift **`currentIndex`** to the right instead of dividing by 2.
- Avoid overwriting the **`currentIndex`** variable and perform the bitwise comparison in-line.

```
function verifyMerkleProof(
...
  uint256 currrentIndex = index;
  for (
      uint256 height = 0;
      height < _DEPOSIT_CONTRACT_TREE_DEPTH;
      height++
  ) {
      if ((currrentIndex & 1) == 1)
          node = keccak256(abi.encodePacked(smtProof[height], node));
      else node = keccak256(abi.encodePacked(node, smtProof[height]));
      currrentIndex /= 2;
  }
```

**Recommendation:** Consider changing the code to

```
function verifyMerkleProof(
...
  uint256 currrentIndex = index;
  for (
      uint256 height = 0;
      height < _DEPOSIT_CONTRACT_TREE_DEPTH;
      height++
  ) {
-     if ((currrentIndex & 1) == 1)
+     if (((index >> height) & 1) == 1)
          node = keccak256(abi.encodePacked(smtProof[height], node));
      else node = keccak256(abi.encodePacked(node, smtProof[height]));
-     currrentIndex /= 2;
  }
```

This same optimization can also be applied to similar instances found in **`_deposit()`** and **`getDepositRoot()`**.

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

### 5.3.6 Increments can be optimised by pre-fixing variable with ++

**Severity:** Gas Optimization

**Context:** DepositContract.sol#L64

**Description:** There are small gas savings in performing when pre-fixing increments with **++**. Sometimes this can be used to combine multiple statements, like in function **`_deposit()`**.

```
function _deposit(bytes32 leafHash) internal {
    ...
    depositCount += 1;
    uint256 size = depositCount;
    ...
}
```

Other occurrences of **++**:

```
PolygonZkEVM.sol:        for (uint256 i = 0; i < batchesNum; i++) {
PolygonZkEVM.sol:                currentLastForceBatchSequenced++;
PolygonZkEVM.sol:            currentBatchSequenced++;
PolygonZkEVM.sol:            lastPendingState++;
PolygonZkEVM.sol:        lastForceBatch++;
PolygonZkEVM.sol:        for (uint256 i = 0; i < batchesNum; i++) {
PolygonZkEVM.sol:            currentLastForceBatchSequenced++;
PolygonZkEVM.sol:            currentBatchSequenced++;
lib/DepositContract.sol:            height++
lib/DepositContract.sol:        for (uint256 height = 0;height < _DEPOSIT_CONTRACT_TREE_DEPTH;height++)
↪  {
lib/DepositContract.sol:        for (uint256 height = 0;height < _DEPOSIT_CONTRACT_TREE_DEPTH;height++)
↪  {
lib/TokenWrapped.sol:                nonces[owner]++,
verifiers/Verifier.sol:        for (uint i = 0; i < elements; i++) {
verifiers/Verifier.sol:        for (uint i = 0; i < input.length; i++) {
verifiers/Verifier.sol:        for (uint i = 0; i < input.length; i++) {
```

**Recommendation:** Consider pre-fixing all variables where **++** is used. Combine statement where possible and it doesn't reduce readability.

```
function _deposit(bytes32 leafHash) internal {
    ...
    uint256 size = ++depositCount;
    ...
}
```

**Polygon-Hermez:** Solved partially in PR 82. Some exceptions where it would make the code less readable or it wouldn't save gas.

**Spearbit:** Verified.

### 5.3.7 Move initialization values from `initialize()` to `immutable` via `constructor`

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L337-L370, PolygonZkEVMBridge.sol#L69-L77

**Description:** The contracts **`PolygonZkEVM`** and **`PolygonZkEVMBridge`** initialize variables via **`initialize()`**. If these variables are never updated they could also be made **`immutable`**, which would save some gas. In order to achieve that, a **`constructor`** has to be added to set the **`immutable`** variables. This could be applicable for **`chainID`** in contract **`PolygonZkEVM`** and **`networkID`** in contract **`PolygonZkEVMBridge`**

```
contract PolygonZkEVM is ... {
    ...
    uint64 public chainID;
    ...
    function initialize(...) ... {
        ...
        chainID = initializePackedParameters.chainID;
        ...
    }
contract PolygonZkEVMBridge is ... {
    ...
    uint32 public networkID;
    ...
    function initialize(uint32 _networkID,...) ... {
        networkID = _networkID;
        ...
    }
```

**Recommendation:** Consider making variables **`immutable`** and add a **`constructor`** to initialize these variables.

**Polygon-Hermez:** Implemented for **`PolygonZkEVM`** in PR 88.

Not implemented for **`PolygonZkEVMBridge`** because it will be deployed with the same address on different chains via CREATE2. In that case it is easier if the **`initBytecode`** is the same and thus its easier to stay with **`initialize()`**.

**Spearbit:** Verified and acknowledged.

### 5.3.8 Optimize `isForceBatchAllowed()`

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L393-L399

**Description:** The modifier **`isForceBatchAllowed()`** includes a redundant check **`== true`**. This can be optimized to save some gas.

```
modifier isForceBatchAllowed() {
    require(forceBatchAllowed == true, ... );
    _;
}
```

**Recommendation:** Consider changing the code to

```
modifier isForceBatchAllowed() {
-   require(forceBatchAllowed == true, ... );
+   require(forceBatchAllowed, ... );
    _;
}
```

**Polygon-Hermez:** We decide to erase this modifier, this was intended to be present only in testnet, when the forced Batches were not supported yet by the node and prover, this won't be necessary anymore. It's a requirement that even the admin won't be able to censor the system.

**Spearbit:** Verified.

#### 5.3.9 Optimize loop in `_updateBatchFee()`

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L839-L908

**Description:** The function **`_updateBatchFee()`** uses the following check in a loop: **`block.timestamp - currentSequencedBatchData.sequencedTimestamp > veryBatchTimeTarget`**. The is the same as: **`block.timestamp - veryBatchTimeTarget > currentSequencedBatchData.sequencedTimestamp`**

As **`block.timestamp - veryBatchTimeTarget`** is constant during the execution of this function, it can be taken outside the loop to save some gas.

```
function _updateBatchFee(uint64 newLastVerifiedBatch) internal {
    ...
    while (currentBatch != currentLastVerifiedBatch) {
        ...
        if ( block.timestamp - currentSequencedBatchData.sequencedTimestamp >  veryBatchTimeTarget ) {
            ...
        }
    }
}
```

**Recommendation:** Consider optimizing the loop by storing the value of **`block.timestamp - veryBatchTimeTarget`** in a temporary variable and use that in the comparison.

**Polygon-Hermez:** Solved in PR 90.

**Spearbit:** Verified.

#### 5.3.10 Optimize multiplication

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L888

**Description:** The multiplication in function **`_updateBatchFee`** can be optimized to save some gas.

**Recommendation:** Consider changing the code to:

```
+unchecked {
-    (10 ** (diffBatches * 3))
+    (1000 ** diffBatches)
+}
```

Note: the gas saving is only received when using **`unchecked`** and is very minimal, so double-check it is worth the trouble.

**Polygon-Hermez:** Solved in PR 90.

**Spearbit:** Verified.

### 5.3.11 Changing `constant` storage variables from `public` to `private` will save gas

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L114-L127, PolygonZkEVMBridge.sol#L38-L44, TokenWrapped.sol#L9-L20

**Description:** Usually **`constant`** variables are not expected to be read on-chain and their value can easily be seen by looking at the source code. For this reason, there is no point in using **`public`** for a **`constant`** variable since it auto-generates a getter function which increases deployment cost and sometimes function call cost.

**Recommendation:** Replace all **`public constant`** occurrences referenced with **`private constant`** to save gas.

**Polygon-Hermez:** The public constants of the **`TokenWrapped`** can be useful on both Front ends implementations or another contract, as can be seen in the implementation of permit2 of uniswap.

The rest of the constants are changed in PR 85.

**Spearbit:** Verified.

### 5.3.12 Storage variables not changeable after deployment can be `immutable`

**Severity:** Gas Optimization

**Context:** Mentioned in **Recommendation**

**Description:** If a storage variable is not changeable after deployment (set in the constructor) it can be turned into an **`immutable`** variable to save gas.

**Recommendation:**

- PolygonZkEVMGlobalExitRoot.sol#L6-L48

```
- import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
- address public bridgeAddress;
- address public rollupAddress;
+ address public immutable bridgeAddress;
+ address public immutable rollupAddress;

- function initialize(address _rollupAddress,address _bridgeAddress) public initializer {
+ constructor(address _rollupAddress,address _bridgeAddress) {
        rollupAddress = _rollupAddress;
        bridgeAddress = _bridgeAddress;
  }
```

- PolygonZkEVMGlobalExitRootL2.sol#L27

```
- address public bridgeAddress;
+ address public immutable bridgeAddress;
```

- PolygonZkEVMTimelock.sol#L14

```
- PolygonZkEVM public polygonZkEVM;
+ PolygonZkEVM public immutable polygonZkEVM;
```

- TokenWrapped.sol#L29-L32

```
- address public bridgeAddress;
+ address public immutable bridgeAddress;
...
- uint8 private _decimals;
+ uint8 private immutable _decimals;
```

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

### 5.3.13 Optimize check in `_consolidatePendingState()`

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L799-L832

**Description:** The check in function **`_consolidatePendingState()`** can be optimized to save some gas. As **`lastPendingStateConsolidated`** is of type **`uint64`** and thus is at least **`0`**, the check **`pendingStateNum > lastPendingStateConsolidated`** makes sure **`pendingStateNum > 0`**.

So the explicit check for **`pendingStateNum != 0`** isn't necessary.

```
uint64 public lastPendingStateConsolidated;
function _consolidatePendingState(uint64 pendingStateNum) internal {
    require(
        pendingStateNum != 0 &&
        pendingStateNum > lastPendingStateConsolidated &&
        pendingStateNum <= lastPendingState,
        "PolygonZkEVM::_consolidatePendingState: pendingStateNum invalid"
    );
    ...
}
```

**Recommendation:** Consider changing the code to

```
function _consolidatePendingState(uint64 pendingStateNum) internal {
    require(
-       pendingStateNum != 0 &&
        pendingStateNum > lastPendingStateConsolidated && // pendingStateNum can't be 0
        pendingStateNum <= lastPendingState,
        "PolygonZkEVM::_consolidatePendingState: pendingStateNum invalid"
    );
    ...
}
```

**Polygon-Hermez:** Solved in PR 87.

**Spearbit:** Verified.

### 5.3.14 Custom errors not used

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L373

**Description:** Custom errors lead to cheaper deployment and run-time costs.

**Recommendation:** For a cheaper gas cost, consider using custom errors throughout the whole project.

**Polygon-Hermez:** Solved in PR 90.

**Spearbit:** Verified.

### 5.3.15 Variable can be updated only once instead of on each iteration of a loop

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L495, PolygonZkEVM.sol#L1039

**Description:** In functions **`sequenceBatches()`** and **`sequenceForceBatches()`**, the **`currentBatchSequenced`** variable is increased by 1 on each iteration of the loop but is not used inside of it. This means that instead of doing **`batchesNum`** addition operations, you can do it only once, after the loop.

**Recommendation:** Delete the **`currentBatchSequenced++;`** line and just do **`currentBatchSequenced += batchesNum;`** right after the loop.

**Polygon-Hermez:** Solved in PR 87.

**Spearbit:** Verified.

### 5.3.16 Optimize emits in `sequenceBatches()` and `sequenceForceBatches()`

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L409-L533, PolygonZkEVM.sol#L972-L1054

**Description:** The **`emit`**s in functions **`sequenceBatches()`** and **`sequenceForceBatches()`** could be gas optimized by using the tmp variables which have been just been stored in the **`emit`**ed global variables.

```
function sequenceBatches(...) ... {
    ...
    lastBatchSequenced = currentBatchSequenced;
    ...
    emit SequenceBatches(lastBatchSequenced);
}
function sequenceForceBatches(...) ... {
    ...
    lastBatchSequenced = currentBatchSequenced;
    ...
    emit SequenceForceBatches(lastBatchSequenced);
}
```

**Recommendation:** Consider changing the code to:

```
function sequenceBatches(...) ... {
    ...
    lastBatchSequenced = currentBatchSequenced;
    ...
-   emit SequenceBatches(lastBatchSequenced);
+   emit SequenceBatches(currentBatchSequenced);
}
function sequenceForceBatches(...) ... {
    ...
    lastBatchSequenced = currentBatchSequenced;
    ...
-   emit SequenceForceBatches(lastBatchSequenced);
+   emit SequenceForceBatches(currentBatchSequenced);
}
```

**Polygon-Hermez:** Solved in PR 85.

**Spearbit:** Verified.

### 5.3.17 Only update `lastForceBatchSequenced` if nessary in function `sequenceBatches()`

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L409-L533

**Description:** The function **`sequenceBatches()`** writes back to **`lastForceBatchSequenced`**, however this is only necessary if there are forced batches. This could be optimized to save some gas and at the same time the calculation of **`nonForcedBatchesSequenced`** could also be optimized.

```
function sequenceBatches(...) ... {
    ...
    uint64 currentLastForceBatchSequenced = lastForceBatchSequenced;
    ...
    if (currentBatch.minForcedTimestamp > 0) {
       currentLastForceBatchSequenced++;
    ...
    uint256 nonForcedBatchesSequenced = batchesNum - (currentLastForceBatchSequenced -
↪   lastForceBatchSequenced);
    ...
    lastForceBatchSequenced = currentLastForceBatchSequenced;
    ...
}
```

**Recommendation:** Consider changing the code to something like the following. Do check if the gas savings are worth the trouble.

```
function sequenceBatches(...) ... {
    ...
    uint64 currentLastForceBatchSequenced = lastForceBatchSequenced;
+   uint64 orgLastForceBatchSequenced = currentLastForceBatchSequenced;
    ...
    if (currentBatch.minForcedTimestamp > 0) {
       currentLastForceBatchSequenced++;
    ...
-   uint256 nonForcedBatchesSequenced = batchesNum - (currentLastForceBatchSequenced -
↪   lastForceBatchSequenced);
+   uint256 nonForcedBatchesSequenced = batchesNum - (currentLastForceBatchSequenced -
↪   orgLastForceBatchSequenced);
    ...
+   if (currentLastForceBatchSequenced != orgLastForceBatchSequenced)
        lastForceBatchSequenced = currentLastForceBatchSequenced;
    ...
}
```

**Polygon-Hermez:** Solved in PR 85.

**Spearbit:** Verified.

### 5.3.18 Delete `forcedBatches[currentLastForceBatchSequenced]` after use

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L409-L533, PolygonZkEVM.sol#L972-L1054

**Description:** The functions **`sequenceBatches()`** and **`sequenceForceBatches()`** use up the **`forcedBatches[]`** and then afterward they are no longer used. Deleting these values might give a gas refund and lower the L1 gas costs.

```
function sequenceBatches(...) ... {
    ...
    currentLastForceBatchSequenced++;
    ...
    require(hashedForcedBatchData ==  forcedBatches[currentLastForceBatchSequenced],...);
    ...
}
function sequenceForceBatches(...) ... {
    ...
    currentLastForceBatchSequenced++;
    ...
    require(hashedForcedBatchData == forcedBatches[currentLastForceBatchSequenced],...);
    ...
}
```

**Recommendation:** Consider deleting the values of **`forcedBatches[currentLastForceBatchSequenced]`**. Verify if this indeed saves gas.

**Polygon-Hermez:** Solved in PR 85.

**Spearbit:** Verified.

### 5.3.19 Calculate `keccak256(currentBatch.transactions)` once

**Severity:** Gas Optimization

**Context:** PolygonZkEVM.sol#L409-L533, PolygonZkEVM.sol#L972-L1054

**Description:** Both functions **`sequenceBatches()`** and **`sequenceForceBatches()`** calculate **`keccak256(currentBatch.transactions)`** twice. As the **`currentBatch.transactions`** could be rather large, calculating the **`keccak256()`** of it could be relatively expensive.

```
function sequenceBatches(BatchData[] memory batches) ... {
    ...
    if (currentBatch.minForcedTimestamp > 0) {
        ...
        bytes32 hashedForcedBatchData = ... keccak256(currentBatch.transactions) ...
        ...
    }
    ...
    currentAccInputHash = ... keccak256(currentBatch.transactions) ...
    ...
}

function sequenceForceBatches(ForcedBatchData[] memory batches) ... {
    ...
    bytes32 hashedForcedBatchData = ... keccak256(currentBatch.transactions) ...
    ...
    currentAccInputHash = ... keccak256(currentBatch.transactions) ...
    ...
}
```

**Recommendation:** Consider storing the result of **`keccak256(currentBatch.transactions)`** in a temporary variable and reuse it later on.

**Polygon-Hermez:** Solved in PR 85.

**Spearbit:** Verified.

## 5.4 Informational

### 5.4.1 Function definition of `onMessageReceived()`

**Severity:** Informational

**Context:** IBridgeMessageReceiver.sol#L9-L13, PolygonZkEVMBridge.sol#L388-L436

**Description:** As discovered by the project: the function definition of **`onMessageReceived()`** is **`view`** and returns a **`bool`**ean. Also, it is not **`payable`**. The function is meant to receive ETH so it should be **`payable`**. Also, it is meant to take action so is shouldn't be **`view`**. The **`bool`** return value isn't used in **`PolygonZkEVMBridge`** so isn't necessary. Because the function is called via a low-level call this doesn't pose a problem in practice. The current definition is confusing though.

```
interface IBridgeMessageReceiver {
    function onMessageReceived(...) external view returns (bool);
}
contract PolygonZkEVMBridge is ... {
    function claimMessage( ... ) ... {
        ...
        (bool success, ) = destinationAddress.call{value: amount}(
            abi.encodeCall(
                IBridgeMessageReceiver.onMessageReceived,
                (originAddress, originNetwork, metadata)
            )
        );
        require(success, "PolygonZkEVMBridge::claimMessage: Message failed");
        ...
    }
}
```

**Recommendation:** Change the function definition to the following:

```
- function onMessageReceived(...) external view returns (bool);
+ function onMessageReceived(...) external payable;
```

**Polygon-Hermez:** Solved in PR 89.

**Spearbit:** Verified.

### 5.4.2 `batchesNum` can be explicitly casted in `sequenceForceBatches()`

**Severity:** Informational

**Context:** PolygonZkEVM.sol#L987-L990

**Description:** The **`sequenceForceBatches()`** function performs a check to ensure that the sequencer does not sequence forced batches that do not exist. The **`require`** statement compares two different types; **`uint256`** and **`uint64`**. For consistency, the **`uint256`** can be safely cast down to **`uint64`** as solidity **`^0.8.0`** checks for overflow/underflow.

**Recommendation:** Consider explicitly casting down **`batchesNum`** to **`uint64`**.

**Polygon-Hermez:** Solved by casting to **`uint256`** in PR 85.

**Spearbit:** Verified.

### 5.4.3 Metadata are not migrated on changes in l1 contract

**Severity:** Informational

**Context:** PolygonZkEVMBridge.sol#L338-L340

**Description:** If metadata changes on mainnet (say decimal change) after wrapped token creation then also wrapped token's metadata will not change and would point to the older decimal

1. Token T1 was on mainnet with decimals 18.
2. This was bridged to rollup R1.
3. A wrapped token is created with decimal 18.
4. On mainnet T1 decimal is changed to 6.
5. Wrapped token on R1 still uses 18 decimals.

**Recommendation:** This behavior needs to be documented so that users are careful while interacting with their L2 counterpart if any such scenario occurs

**Polygon-Hermez:** We consider that this is a very unusual behavior to the point that we are not aware of any token that has changed his metadata ( name/symbol/decimals), even if we could consider that as a malicious token since easily can break multiple dapps or trick users changing any of the metadata fields. We will add a comment that we cannot support an update of the metadata if the original token updates its own metadata.

**Spearbit:** Verified.

### 5.4.4 Remove unused import in `PolygonZkEVMGlobalExitRootL2`

**Severity:** Informational

**Context:** PolygonZkEVMGlobalExitRootL2.sol#L5-L11

**Description:** The contract **`PolygonZkEVMGlobalExitRootL2`** imports **`SafeERC20.sol`**, however, this isn't used in the contract.

```
import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
contract PolygonZkEVMGlobalExitRootL2 {
}
```

**Recommendation:** Remove the unused import in **`PolygonZkEVMGlobalExitRootL2`**.

```
- import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";
```

**Polygon-Hermez:** Solved in PR 82.

**Spearbit:** Verified.

### 5.4.5 Switch from `public` to `external` for all non-internally called methods

**Severity:** Informational

**Context:** DepositContract.sol#L90, DepositContract.sol#L124, Verifier.sol#L320, PolygonZkEVM-GlobalExitRoot.sol#L42, PolygonZkEVMGlobalExitRootL2.sol#L40, PolygonZkEVMBridge.sol#L69, PolygonZkEVMBridge.sol#L129, PolygonZkEVMBridge.sol#L222, PolygonZkEVMBridge.sol#L272, PolygonZkEVMBridge.sol#L388, PolygonZkEVMBridge.sol#L446, PolygonZkEVMBridge.sol#L480, PolygonZkEVM.sol#L337, PolygonZkEVM.sol#L409, PolygonZkEVM.sol#L545, PolygonZkEVM.sol#L624, PolygonZkEVM.sol#L783, PolygonZkEVM.sol#L920, PolygonZkEVM.sol#L972, PolygonZkEVM.sol#L1064, PolygonZkEVM.sol#L1074, PolygonZkEVM.sol#L1084, PolygonZkEVM.sol#L1097, PolygonZkEVM.sol#L1110, PolygonZkEVM.sol#L1133, PolygonZkEVM.sol#L1155, PolygonZkEVM.sol#L1171, PolygonZkEVM.sol#L1182, PolygonZkEVM.sol#L1204, PolygonZkEVM.sol#L1258

**Description:** Functions that are not called from inside of the contract should be **external** instead of **public**, which prevents accidentally using a function internally that is meant to be used externally.

See also issue "Use **calldata** instead of **memory** for function parameters".

**Recommendation:** Change the function visibility from **public** to **external** for the linked methods.

**Polygon-Hermez:** Solved in PR 85. Note: **bridgeAsset()** is still **public** since it's used in one of the mocks and changing to external has no effect on bytecode length or gas cost.

**Spearbit:** Verified.

#### 5.4.6 Common interface for PolygonZkEVMGlobalExitRoot and PolygonZkEVMGlobalExitRootL2

**Severity:** Informational

**Context:** PolygonZkEVMGlobalExitRoot.sol#L5, PolygonZkEVMGlobalExitRootL2.sol#L11

**Description:** The contract **PolygonZkEVMGlobalExitRoot** inherits from **IPolygonZkEVMGlobalExitRoot**, while **PolygonZkEVMGlobalExitRootL2** doesn't, although they both implement a similar interface.

Note: **PolygonZkEVMGlobalExitRoot** implements an extra function **getLastGlobalExitRoot()**. Inheriting from the same interface file would improve the checks by the compiler.

```
import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
contract PolygonZkEVMGlobalExitRoot is IPolygonZkEVMGlobalExitRoot, ... {
    ...
}
contract PolygonZkEVMGlobalExitRootL2 {
}
```

**Recommendation:** Consider having a common interface file and an additional interface for the extra function in **PolygonZkEVMGlobalExitRoot**.

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

#### 5.4.7 Abstract the way to calculate GlobalExitRoot

**Severity:** Informational

**Context:** PolygonZkEVMGlobalExitRoot.sol#L54-L85, PolygonZkEVMBridge.sol#L520-L581

**Description:** The **algorithm** to combine the **mainnetExitRoot** and **rollupExitRoot** is implemented in several locations in the code. This could be abstracted in contract **PolygonZkEVMBridge**, especially because this will be enhanced when more L2s are added.

```
contract PolygonZkEVMGlobalExitRoot is ... {
    function updateExitRoot(bytes32 newRoot) external {
        ...
        bytes32 newGlobalExitRoot = keccak256(abi.encodePacked(lastMainnetExitRoot, lastRollupExitRoot)
↪   ); // first
        ...
    }
    function getLastGlobalExitRoot() public view returns (bytes32) {
        return keccak256(abi.encodePacked(lastMainnetExitRoot, lastRollupExitRoot) ); // second
    }
}
contract PolygonZkEVMBridge is ... {
    function _verifyLeaf(..., bytes32 mainnetExitRoot, bytes32 rollupExitRoot, ...) ... {
        ...
        uint256 timestampGlobalExitRoot = globalExitRootManager
            .globalExitRootMap(  keccak256(abi.encodePacked(mainnetExitRoot, rollupExitRoot))  ); //
↪   third
        ...
    }
}
```

**Recommendation:** Consider using functions like the functions below:

```
function calculateGlobalExitRoot(bytes32 mainnetExitRoot, bytes32 rollupExitRoot) public view returns
↪   (bytes32) {
        return keccak256(abi.encodePacked(mainnetExitRoot, rollupExitRoot) );
}
```

The following function is useful to prevent having to call **`globalExitRootManager`** twice from **`PolygonZkEVMBridge`**.

```
function globalExitRootMapCalculated(bytes32 mainnetExitRoot, bytes32 rollupExitRoot) public view
↪   returns (bytes32) {
    return globalExitRootMap[calculateGlobalExitRoot(mainnetExitRoot, rollupExitRoot)];
}
```

**Polygon-Hermez:** We created a library to abstract this calculation: **`GlobalExitRootLib.sol`**. We didn't put it in the **`PolygonZkEVMGlobalExitRoot`**, since then it should also be put in **`PolygonZkEVMGlobalExitRootL2`** and that contract should be as simple as possible and shouldn't have to be updated when adding new networks. Solved in PR 88.

**Spearbit:** Verified.

### 5.4.8 ETH honeypot on L2

**Severity:** Informational

**Context:** genesis-gen.json#L9

**Description:** The initial ETH allocation to the Bridge contract on L2 is rather large: 2E8 ETH on the test network and 1E11 ETH on the production network according to the documentation. This would make the bridge a large honey pot, even more than other bridges. If someone would be able to retrieve the ETH they could exchange it with all available other coins on the L2, bridge them back to mainnet, and thus steal about all TVL on the L2.

**Recommendation:** A possible solution could be

- Have a **`cold`** wallet contract on L2 that contains the bulk of the ETH.
- Have a function that can only transfer ETH to the bridge contract (the bridge contract must be able to receive this).
- A government action could trigger the ETH transfer to the bridge; making sure there is a sufficiently large amount for any imaginable bridge action.

- Alternatively this action could be permissionless, but that would require implementing time and amount limits, which would complicate the contract.

**Polygon-Hermez:** An alternative to pre-minting could be to have a function to mint ETH, which we decided not to implement because it is too risky. If the bridge would be taken over then there are several other ways to steal the TVL. Any solutions would complicate the logic and introduce more risk, especially if human interaction is involved.

**Spearbit:** Acknowledged.

#### 5.4.9 Allowance is not required to burn wrapped tokens

**Severity:** Informational

**Context:** TokenWrapped.sol#L62-L64

**Description:** The **`burn`** of tokens of the deployed **`TokenWrapped`** doesn't **`use up`** any allowance, because the Bridge has the right to burn the wrapped token. Normally a user would approve a certain amount of tokens and then do an action (e.g. **`bridgeAsset()`**). This could be seen as an extra safety precaution. So you lose the extra safety this way and it might be unexpected from the user's point of view. However, it's also very convenient to do a one-step bridge (comparable to using the **`permit`**). Note: most other bridges do it also this way.

```
function burn(address account, uint256 value) external onlyBridge {
    _burn(account, value);
}
```

**Recommendation:** Document the behavior so users are aware an allowance is not required.

**Polygon-Hermez:** Will be added to the documentation and a comment is added in PR 88.

**Spearbit:** Verified.

#### 5.4.10 Messages are lost when delivered to EOA by `claimMessage()`

**Severity:** Informational

**Context:** PolygonZkEVMBridge.sol#L388-L436

**Description:** The function **`claimMessage()`** calls the function **`onMessageReceived()`** via a low-level call. When the receiving address doesn't contain a contract the low-level call still succeeds and delivers the ETH. The documentation says: "... IBridgeMessageReceiver interface and such interface must be fulfilled by the receiver contract, it will ensure that the receiver contract has implemented the logic to handle the message."

As we understood from the project this behavior is intentional. It can be useful to deliver ETH to Externally owned accounts (EOAs), however, the message (which is the main goal of the function) isn't interpreted and thus lost, without any notification.

The loss of the delivery of the message to EOAs (e.g. non contracts) might not be obvious to the casual readers of the code/documentation.

```
function claimMessage(...) ... {
    ...
    (bool success, ) = destinationAddress.call{value: amount}(
        abi.encodeCall(
            IBridgeMessageReceiver.onMessageReceived,
            (originAddress, originNetwork, metadata)
        )
    );
   require(success, "PolygonZkEVMBridge::claimMessage: Message failed");
    ...
}
```

**Recommendation:** Doublecheck this behavior is true as intended. Add comments about EOAs in the code and clarify them in the documentation.

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

#### 5.4.11 Replace assembly of `_getSelector()` with Solidity

**Severity:** Informational

**Context:** PolygonZkEVMBridge.sol#L611-L736

**Description:** The function **`_getSelector()`** gets the first four bytes of a series of bytes and used assembly. This can also be implemented in Solidity, which is easier to read.

```
function _getSelector(bytes memory _data) private pure returns (bytes4 sig) {
    assembly { sig := mload(add(_data, 32))  }
}
function _permit(..., bytes calldata permitData) ... {
    bytes4 sig = _getSelector(permitData);
    ...
}
```

**Recommendation:** Consider using the following way to retrieve the first four bytes and remove function **`_getSelector()`**

```
function _permit(..., bytes calldata permitData) ... {
-   bytes4 sig = _getSelector(permitData);
+   bytes4 sig = bytes4(permitData[:4]);
```

**Polygon-Hermez:** Solved in PR 82.

**Spearbit:** Verified.

#### 5.4.12 Improvement suggestions for `Verifier.sol`

**Severity:** Informational

**Context:** Verifier.sol#L14, IVerifierRollup.sol

**Description:** **`Verifier.sol`** is a contract automatically generated by snarkjs and is based on the template verifier_groth16.sol.ejs. There are some details that can be improved on this contract. However, changing it will require doing PRs for the Snarkjs project.

**Recommendation:** We have the following improvement suggestions:

- Change to Solidity version 0.8.x because version 0.6.11 is older and misses overflow/underflow checks. Also, rest of the PolygonZkEVM uses version 0.8.x. This will also allow using custom errors.
- Use **`uint256`** instead of **`uint`**, because then its immediately clear how large the **`uint`**s are.
- Doublecheck **`sub(gas(), 2000`** in the **`staticcall()`**s, as it might not be necessary anymore after the Tangerine Whistle fork because only 63/64 of the gas is sent.
- Generate an explicit interface file like **`IVerifierRollup.sol`** and let **`Verifier.sol`** inherit this to make sure they are compatible.

We also have these suggestions for gas optimizations:

- **`uint q`** in function **`negate()`** could be a constant;
- **`uint256 snark_scalar_field`** in function **`verify()`** could be a constant;
- remove **`switch success case 0 { invalid() } }`** as it is redundant with the **`require()`** after it;
- in function **`pairing()`** store the **`i*6`** in a tmp variable;
- in function **`verifyProof()`** use **`return (verify(inputValues, proof) == 0);`**.

**Polygon-Hermez:** These are great suggestions, but it requires some time to analyze if these optimizations are completely safe so we will postpone it to add them in future versions.

**Spearbit:** Acknowledged.

#### 5.4.13 Variable named incorrectly

**Severity:** Informational

**Context:** PolygonZkEVM.sol#L854

**Description:** Seems like the variable **veryBatchTimeTarget** was meant to be named **verifyBatchTimeTarget** as evidenced from the comment below:

```
// Check if timestamp is above or below the VERIFY_BATCH_TIME_TARGET
```

**Recommendation:** Rename the variable **veryBatchTimeTarget** to **verifyBatchTimeTarget**. Also, rename the event **SetVeryBatchTimeTarget** to **SetVerifyBatchTimeTarget** In case if the variable is named correctly then update the comment below:

```
// Check if timestamp is above or below the veryBatchTimeTarget
```

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

#### 5.4.14 Add additional comments to function `forceBatch()`

**Severity:** Informational

**Context:** PolygonZkEVM.sol#L920-L966

**Description:** The function **forceBatch()** contains a comment about **synch attacks**. It's not immediately clear what is meant by that. The team explained the following:

- Getting the call data from an EOA is easy/cheap so no need to put the transactions in the event (which is expensive).
- Getting the internal call data from internal transactions (which is done via a smart contract) is complicated (because it requires an archival node) and then it's worth it to put the transactions in the **event**, which is easy to query.

```
function forceBatch(...) ... {
    ...
    // In order to avoid synch attacks, if the msg.sender is not the origin
    // Add the transaction bytes in the event
    if (msg.sender == tx.origin) {
        emit ForceBatch(lastForceBatch, lastGlobalExitRoot, msg.sender, "");
    } else {
        emit ForceBatch(lastForceBatch,lastGlobalExitRoot,msg.sender,transactions);
   }
}
```

**Recommendation:** Add additional comments to the function **forceBatch()** to explain the approach taken.

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

### 5.4.15 Check against `MAX_VERIFY_BATCHES`

**Severity:** Informational

**Context:** PolygonZkEVM.sol#L409-L533, PolygonZkEVM.sol#L545-L611, PolygonZkEVM.sol#L972-L1054

**Description:**In several functions a comparison is made with **`< MAX_VERIFY_BATCHES`**. This should probably be **`<= MAX_VERIFY_BATCHES`**, otherwise, the MAX will never be reached.

```
uint64 public constant MAX_VERIFY_BATCHES = 1000;
function sequenceForceBatches(ForcedBatchData[] memory batches) ... {
    uint256 batchesNum = batches.length;
    ...
    require(batchesNum < MAX_VERIFY_BATCHES, ... );
    ...
}
function sequenceBatches(BatchData[] memory batches) ... {
    uint256 batchesNum = batches.length;
    ...
    require(batchesNum < MAX_VERIFY_BATCHES, ...);
    ...
}
function verifyBatches(...) ... {
    ...
   require(finalNewBatch - initNumBatch < MAX_VERIFY_BATCHES, ... );
   ...
}
```

**Recommendation:** Double-check the conclusion and consider applying these changes

```
-require( ... < MAX_VERIFY_BATCHES, ... );
+require( ... <= MAX_VERIFY_BATCHES, ... );
```

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

### 5.4.16 Prepare for multiple aggregators/sequencers to improve availability

**Severity:** Informational

**Context:** PolygonZkEVM.sol#L377-L391

**Description:** As long are there is one (trusted)sequencer and one (trusted)aggregator the availability risks are relatively high. However, the current code isn't optimized to support multiple trusted sequencers and multiple trusted aggregators.

```
modifier onlyTrustedSequencer() {
    require(trustedSequencer == msg.sender, ... );
    _;
}
modifier onlyTrustedAggregator() {
    require(trustedAggregator == msg.sender, ... );
    _;
}
```

**Recommendation:** It might be useful to make small changes in the code to support multiple trusted sequencers and multiple trusted aggregators to improve availability.

**Polygon-Hermez:** Since the **`trustedSequencer`**/**`trustedAggregator`** address can be changed, if we want to support multiple trusted actors or/and add a consensus layer to coordinate them this will be delegated in the future by another smart contract.

**Spearbit:** Acknowledged.

#### 5.4.17 Temporary Fund freeze on using Multiple Rollups

**Severity:** Informational

**Context:** PolygonZkEVMBridge.sol#L157-L165

**Description:** Claiming of Assets will freeze temporarily if multiple rollups are involved as shown below.

The asset will be lost if the transfer is done between: a. Mainnet -> R1 -> R2 b. R1 -> R2 -> Mainnet

1. USDC is bridged from Mainnet to Rollover R1 with its metadata.
2. User claims this and a new Wrapped token is prepared using USDC token and its metadata.

```
bytes32 tokenInfoHash = keccak256(abi.encodePacked(originNetwork, originTokenAddress));
TokenWrapped newWrappedToken = (new TokenWrapped){salt: tokenInfoHash}(name, symbol, decimals);
```

3. Let's say the User bridge this token to Rollup R2. This will burn the wrapped token on R1

```
 if (tokenInfo.originTokenAddress != address(0)) {
    // The token is a wrapped token from another network
    // Burn tokens
    TokenWrapped(token).burn(msg.sender, amount);
    originTokenAddress = tokenInfo.originTokenAddress;
    originNetwork = tokenInfo.originNetwork;
}
```

4. The problem here is now while bridging the metadata was not set.
5. So once the user claims this on R2, wrapped token creation will fail since abi.decode on empty metadata will fail to retrieve name, symbol,...

The asset will be temporarily lost since it was bridged properly but cannot be claimed

Showing the transaction chain

Mainnet bridgeAsset(usdc,R1,0xUser1, 100, “)

- Transfer 100 USDC to Mainnet M1
- originTokenAddress=USDC
- originNetwork = Mainnet
- metadata = (USDC,USDC,6)
- Deposit node created

R1 claimAsset(...,Mainnet,USDC,R1,0xUser1,100, metadata = (USDC,USDC,6))

- Claim verified
- Marked claimed
- tokenInfoHash derived from originNetwork, originTokenAddress which is Mainnet, USDC
- tokenInfoToWrappedToken[Mainnet,USDC] created using metadata = (USDC,USDC,6)
- User minted 100 amount of tokenInfoToWrappedToken[Mainnet, USDC]

bridgeAsset(tokenInfoToWrappedToken[Mainnet,USDC],R2,0xUser2, 100, “)

- Burn 100 tokenInfoToWrappedToken[Mainnet,USDC]
- originTokenAddress=USDC
- originNetwork = Mainnet

- metadata = ""
- Deposit node created with empty metadata

R2 claimAsset(...,Mainnet,USDC,R2,0xUser2,100, metadata = "")

- Claim verified
- Marked claimed
- tokenInfoHash derived from originNetwork, originTokenAddress which is Mainnet, USDC
- Since metadata = "" , abi decode fails

**Recommendation:** Since the current system does not have multiple rollups currently so marking this issue informational. But in the future when multiple rollups are in place, this code needs to be upgraded to retrieve metadata during burn while bridging which will ensure recipient rollup to correctly decode metadata

**Polygon-Hermez:** We will take this into account if we upgrade the system to support multiple rollups.

**Spearbit:** Acknowledged.

#### 5.4.18 Off by one error when comparing with `MAX_TRANSACTIONS_BYTE_LENGTH` constant

**Severity:** Informational

**Context:** PolygonZkEVM.sol#L933, PolygonZkEVM.sol#L471

**Description:** When comparing against **`MAX_TRANSACTIONS_BYTE_LENGTH`**, the valid range should be **<=** instead of **<**.

```
require(
    transactions.length < MAX_TRANSACTIONS_BYTE_LENGTH,
    "PolygonZkEVM::forceBatch: Transactions bytes overflow"
);
```

```
require(
    currentBatch.transactions.length <
    MAX_TRANSACTIONS_BYTE_LENGTH,
    "PolygonZkEVM::sequenceBatches: Transactions bytes overflow"
);
```

**Recommendation:** Consider using **<=** instead of **<**.

**Polygon-Hermez:** Solved in PR 88.

**Spearbit:** Verified.

#### 5.4.19 trustedAggregatorTimeout value may impact batchFees

**Severity:** Informational

**Context:** PolygonZkEVM.sol#L855-L858 , PolygonZkEVM.sol#L557-L562

**Description:** If **`trustedAggregatorTimeout`** and **`veryBatchTimeTarget`** are valued nearby then all batches verified by 3rd party will be above target (totalBatchesAboveTarget) and this would impact batch fees.

1. Let's say veryBatchTimeTarget is 30 min and trustedAggregatorTimeout is 31 min.
2. Now anyone can call verifyBatches only after 31 min due to the below condition.

```
require(
            sequencedBatches[finalNewBatch].sequencedTimestamp +
                trustedAggregatorTimeout <=
                block.timestamp,
            "PolygonZkEVM::verifyBatches: Trusted aggregator timeout not expired"
        );
```

3. This means _updateBatchFee can at minimum be called after 31 min of sequencing by a nontrusted aggregator.
4. The below condition then always returns true.

```
if (
                block.timestamp - currentSequencedBatchData.sequencedTimestamp >veryBatchTimeTarget
// 31>30
            ) {
```

**Recommendation:** As mentioned by the product team, the **`trustedAggregatorTimeout`** value is only meant to be high at initialization and emergency time, during which 3rd party aggregation should not happen, thus preventing the issue For time part from initialization and emergency, **`trustedAggregatorTimeout`** is meant to move towards 0 value. Hence it could be documented as a reference to Admin that Admin should always take care to set the value lower than **`veryBatchTimeTarget`** during normal scenarios.

**Polygon-Hermez:** Add a comment in PR 88.

**Spearbit:** Verified.

#### 5.4.20 Largest allowed batch fee multiplier is `1023` instead of `1024`

**Severity:** Informational

**Context:** PolygonZkEVM.sol#L1155, PolygonZkEVM.sol#L133

**Description:** Per the **`setMultiplierBatchFee`** function, the largest allowed batch fee multiplier is **`1023`**.

```
/**
* @notice Allow the admin to set a new multiplier batch fee
* @param newMultiplierBatchFee multiplier bathc fee
*/
function setMultiplierBatchFee(
    uint16 newMultiplierBatchFee
) public onlyAdmin {
    require(
    newMultiplierBatchFee >= 1000 && newMultiplierBatchFee < 1024,
    "PolygonZkEVM::setMultiplierBatchFee: newMultiplierBatchFee incorrect range"
    );

    multiplierBatchFee = newMultiplierBatchFee;
    emit SetMultiplierBatchFee(newMultiplierBatchFee);
}
```

However, the comment mentioned that the largest allowed is **`1024`**.

```
// Batch fee multiplier with 3 decimals that goes from 1000 - 1024
uint16 public multiplierBatchFee;
```

**Recommendation:** The implementation should be consistent with the comment and vice versa. Therefore, consider updating the implementation to allow the largest batch fee multiplier to be **`1024`** or update the comment accordingly.

**Polygon-Hermez:** The comment is updated in PR 87.

**Spearbit:** Verified.

#### 5.4.21 Deposit token associated Risk Awareness

**Severity:** Informational

**Context:** PolygonZkEVMBridge.sol#L129

**Description:** The deposited tokens locked in L1 could be at risk due to external conditions like the one shown below:

1. Assume there is a huge amount of token X being bridged to roll over.
2. Now mainnet will have a huge balance of token X.
3. Unfortunately due to a hack or LUNA like condition, the project owner takes a snapshot of the current token X balance for each user address and later all these addresses will be airdropped with a new token based on screenshot value.
4. In this case, token X in mainnet will be screenshot but at disbursal time the newly updated token will be airdropped to mainnet and not the user.
5. Now there is no emergencywithdraw method to get these airdropped funds out.
6. For the users, if they claim funds they still get token X which is worthless.

**Recommendation:** Update the documentation to make users aware of such edge cases and possible actions to be taken

**Polygon-Hermez:** This is not an scenario that affects only our zkevm, but a scenario that affects all contracts (defi, bridges, etc...). User usually notice this kind of situations, that's why usually will move their funds back to their mainnet address. For those who don't know about this situation, usually projects that make this kind of airdrops warns users about this conditions. Nevertheless the project could even airdrop the corresponding funds in L2, but this will always be responsibility of the project.

**Spearbit:** Acknowledged.

#### 5.4.22 Fees might get stuck when Aggregator is unable to verify

**Severity:** Informational

**Context:** PolygonZkEVM.sol#L523

**Description:** The collected fees from Sequencer will be stuck in the contract if Aggregator is unable to verify the batch. In this case, Aggregator will not be paid and the batch transaction fee will get stuck in the contract

**Recommendation:** In the future if these collected fees become significant then a temporary contract upgrade may be required to allow transferring these funds.

**Polygon-Hermez:** This is intended, the fees must be stuck in the contract until an aggregator verifies batches. In another way, if the fees could be retrieved from the contract, the aggregator reward could be stolen. Makes sense that aggregators do not receive rewards if the network is stopped. In case an aggregator is unable to verify a batch, then the whole system will be blocked. The aggregator fees might be a problem, but a very small one compared with a whole network that is stopped ( or at least the withdrawals are stopped), so technically all the bridge funds will be blocked, which is a bigger problem, and for sure we will need an upgrade if this happens In summary, the point is that we are making everything to avoid this situation, but if this happens, then anyway an upgrade must be necessary, we don't think it's worth putting any additional mechanism in this regard.

**Spearbit:** Acknowledged.

#### 5.4.23 Consider using OpenZeppelin's `ECDSA` library over `ecrecover`

**Severity:** Informational

**Context:** TokenWrapped.sol#L100

**Description:** As stated here, **`ecrecover`** is vulnerable to a signature malleability attack. While the code in **`permit`** is not vulnerable since a nonce is used in the signed data, I'd still recommend using OpenZeppelin's ECDSA library, as it does the malleability safety check for you as well as the **`signer != address(0)`** check done on the next line.

**Recommendation:** Replace the **`ecrecover`** call with an **`ECDSA.recover()`** call.

**Polygon-Hermez:** We prefer to use this approach since does not perform additional checks, is more gas efficient and simpler and it's used by very common projects like UNI.

**Spearbit:** Acknowledged.

#### 5.4.24 Risk of transactions not yet in `Consolidated state` on L2

**Severity:** Informational

**Context:** PolygonZkEVMBridge.sol#L540-L548

**Description:** There is are relatively long period for batches and thus transactions are to be between **`Trusted state`** and **`Consolidated state`**. Normally around 30 minutes but in exceptional situations up to 2 weeks. On the L2, users normally interact with the **`Trusted state`**. However, they should be aware of the risk for high-value transactions (especially for transactions that can't be undone, like transactions that have an effect outside of the L2, like off ramps, OTC transactions, alternative bridges, etc). There will be custom RPC endpoints that can be used to retrieve status information, see zkevm.go.

**Recommendation:** Make sure to document the risk of transactions that are not consolidated yet on L2.

**Polygon-Hermez:** We will add a comment regarding this in the documentation. We also want to add this information into the block explorer, similar on how optimism is doing it: optimistic.etherscan. In the field **`blockNumber`** put extra information like: confirmed by sequencer, virtual state, consolidate state.

**Spearbit:** Acknowledged.

#### 5.4.25 Delay of bridging from L2 to L1

**Severity:** Informational

**Context:** PolygonZkEVMBridge.sol#L540-L548

**Description:** The bridge uses the **`Consolidated state`** while bridging from L2 to L1 and the user interface public.zkevm-test.net, shows "Waiting for validity proof. It can take between 15 min and 1 hour.". Other (optimistic) bridges use liquidity providers who take the risk and allow users to retrieve funds in a shorter amount of time (for a fee).

**Recommendation:** Consider implementing a mechanism to reduce the time to bridge from L2 to L1.

**Polygon-Hermez:** We do not plan to implement the such mechanism. We consider that the 15min-1hour is an acceptable delay since it's a typical timing is most CEX, so users are already used to this timing for being able to withdraw funds. In case it becomes a must for the users, we expect that other projects can deploy on top of our system and provide such solutions with some incentives mechanism using the messages between layers.

**Spearbit:** Acknowledged.

### 5.4.26 Missing Natspec documentation

**Severity:** Informational

**Context:** PolygonZkEVM.sol#L672, PolygonZkEVM.sol#L98

**Description:** Some NatSpec comments are either missing or are incomplete.

- Missing NatSpec comment for **`pendingStateNum`**:

```
/**
* @notice Verify batches internal function
* @param initNumBatch Batch which the aggregator starts the verification
* @param finalNewBatch Last batch aggregator intends to verify
* @param newLocalExitRoot  New local exit root once the batch is processed
* @param newStateRoot New State root once the batch is processed
* @param proofA zk-snark input
* @param proofB zk-snark input
* @param proofC zk-snark input
*/
function _verifyBatches(
    uint64 pendingStateNum,
    uint64 initNumBatch,
    uint64 finalNewBatch,
    bytes32 newLocalExitRoot,
    bytes32 newStateRoot,
    uint256[2] calldata proofA,
    uint256[2][2] calldata proofB,
    uint256[2] calldata proofC
) internal {
```

- Missing NatSpec comment for **`pendingStateTimeout`**:

```
  /**
  * @notice Struct to call initialize, this basically saves gas becasue pack the parameters that
↪   can be packed
  * and avoid stack too deep errors.
  * @param admin Admin address
  * @param chainID L2 chainID
  * @param trustedSequencer Trusted sequencer address
  * @param forceBatchAllowed Indicates wheather the force batch functionality is available
  * @param trustedAggregator Trusted aggregator
  * @param trustedAggregatorTimeout Trusted aggregator timeout
  */
  struct InitializePackedParameters {
      address admin;
      uint64 chainID;
      address trustedSequencer;
      uint64 pendingStateTimeout;
      bool forceBatchAllowed;
      address trustedAggregator;
      uint64 trustedAggregatorTimeout;
  }
```

**Recommendation:** Add or complete missing NatSpec comments.

**Polygon-Hermez:** Solved in PR 87.

**Spearbit:** Verified.

#### 5.4.27 _minDelay could be 0 without emergency

**Severity:** Informational

**Context:** PolygonZkEVMTimelock.sol#L40-L46

**Description:** Normally min delay is only supposed to be 0 when in an emergency state. But this could be made to 0 even in nonemergency mode as shown below:

1. Proposer can propose an operation for changing _minDelay to 0 via updateDelay function.
2. Now, if this operation is executed by the executor then _minDelay will be 0 even without an emergency state.

**Recommendation:**Override **`updateDelay`** function and impose a minimum timelock delay preventing 0 delay OR Update docs to make users aware of such scenario(s) so that they can take a decision if such a proposal ever arrives

**Polygon-Hermez:** Being able to update the delay to **`0`** is the standard in most **`Timelock`** implementations, including the one followed by Openzepplin, we don't see a reason to change its default behavior. Since this is the default behavior, users are already aware of this scenario.

**Spearbit:** Acknowledged.

#### 5.4.28 Incorrect/incomplete comment

**Severity:** Informational

**Context:** Mentioned in Recommendation

**Description:** There are a few mistakes in the comments that can be corrected in the codebase.

**Recommendation:**

- PolygonZkEVM.sol#L1106: This function is used for configuring the pending state timeout.

```
- * @notice Allow the admin to set a new trusted aggregator timeout
+ * @notice Allow the admin to set a new pending state timeout
```

- PolygonZkEVMBridge.sol#L217: Function also sends ETH, would be good to add that to the NatSpec.

```
- * @notice Bridge message
+ * @notice Bridge message and send ETH value
```

- PolygonZkEVMBridge.sol#L69: A developer assumption is not documented.

```
+ * @notice The value of `_polygonZkEVMaddress` on the L2 deployment of the contract will be
↪  `address(0)`, so
+ * emergency state is not possible for the L2 deployment of the bridge, intentionally
```

- PolygonZkEVM.sol#L141: The comment is not synchronized with the **`ForcedBatchData`** struct. It should be **`minForcedTimestamp`** instead of **`minTimestamp`** for the last parameter.

```
// hashedForcedBatchData: hash containing the necessary information to force a batch:
- // keccak256(keccak256(bytes transactions), bytes32 globalExitRoot, unint64 minTimestamp)
+ // keccak256(keccak256(bytes transactions), bytes32 globalExitRoot, unint64 minForcedTimestamp)
```

- PolygonZkEVM.sol#L522: Comment is incorrect, the sequencer actually pays collateral for every non-forced batch submitted, not for every batch submitted

```
- // Pay collateral for every batch submitted
+ // Pay collateral for every non-forced batch submitted
```

- PolygonZkEVM.sol#L864: Comment is incorrect, the actual variable updated is **`currentBatch`** not **`currentLastVerifiedBatch`**

```
- // update currentLastVerifiedBatch
+ // update currentBatch
```

- PolygonZkEVM.sol#L1193: Comment seems to be copy-pasted from another method **proveNonDeterministicPendingState**, remove it or write a comment for the current function

```
- * @notice Allows to halt the PolygonZkEVM if its possible to prove a different state root
↪  given the same batches
```

**Polygon-Hermez:** Solved in PR 87.

**Spearbit:** Verified.

#### 5.4.29 Typos, grammatical and styling errors

**Severity:** Informational

**Context:** Mentioned in Recommendation

**Description:** There are a few typos and grammatical mistakes that can be corrected in the codebase. Some functions could also be renamed to better reflect their purposes.

**Recommendation:**

- PolygonZkEVM.sol#L671: The function **_verifyBatches()** does more than verifying batches. It also transfers matic. Therefore, it is recommended to change the function name and comment.

```
- function _verifyBatches(
+ function _verifyAndRewardBatches(
    uint64 pendingStateNum,
    uint64 initNumBatch,
```

- PolygonZkEVMBridge.sol#L37: Typo. Should be **identifier** instead of **indentifier**

```
- // Mainnet indentifier
+ // Mainnet identifier
```

- TokenWrapped.sol#L53: Typo. Should be **immutable** instead of **inmutable**

```
-        // initialize inmutable variables
+        // initialize immutable variables
```

- PolygonZkEVM.sol#L407, PolygonZkEVM.sol#L970: Typo and grammatical error. Should be **batches to** instead of **batces ot** and should be **Struct array which holds the necessary data** instead of **Struct array which the necessary data**

```
- * @param batches Struct array which the necessary data to append new batces ot the sequence
+ * @param batches Struct array which holds the necessary data to append new batches to the
↪  sequence
```

- PolygonZkEVM.sol#L1502: Typo. Should be **the** instead of **teh**

```
- * @param initNumBatch Batch which the aggregator starts teh verification
+ * @param initNumBatch Batch which the aggregator starts the verification
```

- PolygonZkEVM.sol#L1396: Typo. Should be **contracts** instead of **contrats**

```
-     * @notice Function to activate emergency state, which also enable the emergency mode on
↪  both PolygonZkEVM and PolygonZkEVMBridge contrats
+     * @notice Function to activate emergency state, which also enable the emergency mode on
↪  both PolygonZkEVM and PolygonZkEVMBridge contracts
```

- PolygonZkEVM.sol#L1430: Typo. Should be **contracts** instead of **contrats**

```
-     * @notice Function to deactivate emergency state on both PolygonZkEVM and
↪  PolygonZkEVMBridge contrats
+     * @notice Function to deactivate emergency state on both PolygonZkEVM and
↪  PolygonZkEVMBridge contracts
```

- PolygonZkEVM.sol#L144: Typo. Should be **contracts** instead of **contrats**

```
-     * @notice Internal function to activate emergency state on both PolygonZkEVM and
↪  PolygonZkEVMBridge contrats
+     * @notice Internal function to activate emergency state on both PolygonZkEVM and
↪  PolygonZkEVMBridge contracts
```

- PolygonZkEVM.sol#L1108: Typo. Should be **aggregator** instead of **aggreagator**

```
-    * @param newTrustedAggregatorTimeout Trusted aggreagator timeout
+    * @param newTrustedAggregatorTimeout Trusted aggregator timeout
```

- PolygonZkEVM.sol#L969: Grammatical error. Should be **has not done so in the timeout period** instead of **do not have done it in the timeout period**

```
-  * @notice Allows anyone to sequence forced Batches if the trusted sequencer do not have done
↪  it in the timeout period
+  * @notice Allows anyone to sequence forced Batches if the trusted sequencer has not done so
↪  in the timeout period
```

- PolygonZkEVM.sol#L1291: Typo. Should be **function** instead of **functoin**

```
-    * @notice Internal functoin that prove a different state root given the same batches to
↪  verify
+    * @notice Internal function that prove a different state root given the same batches to
↪  verify
```

- PolygonZkEVM.sol#L1062: Typo. Should be **sequencer** instead of **sequuencer**

```
-    * @param newTrustedSequencer Address of the new trusted sequuencer
+    * @param newTrustedSequencer Address of the new trusted sequencer
```

- PolygonZkEVM.sol#L1094: Legacy comment. Does not make sense with current code

```
-    * If address 0 is set, everyone is free to aggregate
```

- PolygonZkEVM.sol#L1131: Typo. Should be **aggregator** instead of **aggreagator**

```
-    * @param newPendingStateTimeout Trusted aggreagator timeout
+    * @param newPendingStateTimeout Trusted aggregator timeout
```

- PolygonZkEVM.sol#L1153: Typo. Should be **batch** instead of **bathc**

```
-    * @param newMultiplierBatchFee multiplier bathc fee
+    * @param newMultiplierBatchFee multiplier batch fee
```

- PolygonZkEVM.sol#L1367: Grammatical error. Should be **must be equal to** instead of **must be equal than**

```
-    "PolygonZkEVM::_proveDistinctPendingState: finalNewBatch must be equal than
↪  currentLastVerifiedBatch"
+    "PolygonZkEVM::_proveDistinctPendingState: finalNewBatch must be equal to
↪  currentLastVerifiedBatch"
```

- PolygonZkEVM.sol#L332: Typo. Should be **deep** instead of **depp**

```
-   * @param initializePackedParameters Struct to save gas and avoid stack too depp errors
+  * @param initializePackedParameters Struct to save gas and avoid stack too deep errors
```

- PolygonZkEVM.sol#L85: Typo. Should be **because** instead of **becasue**

```
- * @notice Struct to call initialize, this basically saves gas becasue pack the parameters that
↪  can be packed
+ * @notice Struct to call initialize, this basically saves gas because pack the parameters that
↪  can be packed
```

- PolygonZkEVM.sol#L59: Typo. Should be **calculate** instead of **calcualte**

```
- * @param previousLastBatchSequenced Previous last batch sequenced before the current one, this
↪  is used to properly calcualte the fees
+ * @param previousLastBatchSequenced Previous last batch sequenced before the current one, this
↪  is used to properly calculate the fees
```

- PolygonZkEVM.sol#L276: Typo. Should be **sequencer** instead of **seequencer**

```
- * @dev Emitted when the admin update the seequencer URL
+ * @dev Emitted when the admin update the sequencer URL
```

- PolygonZkEVM.sol#L459: Grammatical error. Should be **Check global exit root exists with proper batch length. These checks are already done in the forceBatches call.** instead of **Check global exit root exist, and proper batch length, this checks are already done in the forceBatches call**

```
-  // Check global exit root exist, and proper batch length, this checks are already done in the
↪  forceBatches call
+  // Check global exit root exists with proper batch length. These checks are already done in
↪  the forceBatches call.
```

- PolygonZkEVM.sol#L123, PolygonZkEVM.sol#L755: Typo. Should be **tries** instead of **trys**

```
-  // This should be a protection against someone that trys to generate huge chunk of invalid
↪  batches, and we can't prove otherwise before the pending timeout expires
+  // This should be a protection against someone that tries to generate huge chunk of invalid
↪  batches, and we can't prove otherwise before the pending timeout expires
```

```
-  * It trys to consolidate the first and the middle pending state
+  * It tries to consolidate the first and the middle pending state
```

- PolygonZkEVM.sol#L55: Grammatical error. Should be **Struct which will be stored** instead of **Struct which will stored**

```
-  * @notice Struct which will stored for every batch sequence
+  * @notice Struct which will be stored for every batch sequence
```

- PolygonZkEVM.sol#L71: Grammatical error. should be **will be turned off** instead of **will be turn off** and should be **in the future** instead of **in a future**

```
-  * This is a protection mechanism against soundness attacks, that will be turn off in a future
+  * This is a protection mechanism against soundness attacks, that will be turned off in the
↪  future
```

- PolygonZkEVM.sol#L90: Typo. Should be **whether** instead of **wheather**

```
-  * @param forceBatchAllowed Indicates wheather the force batch functionality is available
+  * @param forceBatchAllowed Indicates whether the force batch functionality is available
```

**Polygon-Hermez:** Solved in PR 87.

**Spearbit:** Verified.

#### 5.4.30 Enforce parameters limits in `initialize()` of `PolygonZkEVM`

**Severity:** Informational

**Context:** PolygonZkEVM.sol#L337-L370, PolygonZkEVM.sol#L1110-L1149

**Description:** The function **`initialize()`** of **`PolygonZkEVM`** doesn't enforce limits on **`trustedAggregatorTimeout`** and **`pendingStateTimeout`**, whereas the update functions **`setTrustedAggregatorTimeout()`** and **`setPendingStateTimeout()`**.

As the project has indicated it might be useful to set larger values in **`initialize()`**.

```
function initialize(..., InitializePackedParameters calldata initializePackedParameters,...) ... {
    trustedAggregatorTimeout = initializePackedParameters.trustedAggregatorTimeout;
    ...
    pendingStateTimeout = initializePackedParameters.pendingStateTimeout;
    ...
}
function setTrustedAggregatorTimeout(uint64 newTrustedAggregatorTimeout) public onlyAdmin {
    require(newTrustedAggregatorTimeout <= HALT_AGGREGATION_TIMEOUT,....);
    ...
    trustedAggregatorTimeout = newTrustedAggregatorTimeout;
    ...
}
function setPendingStateTimeout(uint64 newPendingStateTimeout) public onlyAdmin {
    require(newPendingStateTimeout <= HALT_AGGREGATION_TIMEOUT, ... );
    ...
    pendingStateTimeout = newPendingStateTimeout;
    ...
}
```

**Recommendation:** Consider enforcing the same limits in **`initialize()`** as in the update functions, this will make the code easier to reason about. If different limits are allowed on purpose then add some comments.

**Polygon-Hermez:** Solved in PR 85.

**Spearbit:** Verified.

# 6 Appendix

## 6.1 Introduction

On March 20th of 2023, Spearbit conducted a deployment review for Polygon zkEVM.

The target of the review was zkevm-contracts on commit cddde2.

*Disclaimer*: This security review does not guarantee against a hack. It is a snapshot in time of the deployment review for Polygon zkEVM according to the specific commit. Any modifications to the code will require a new security review.

## 6.2 Changes after first review

Several changes have been made after the first Spearbit review conducted on January 2023 (found in the main body of this document), which include:

- A change in the verifier to FflonkVerifier;
- A way to delay setting the global exit root as well as **`updateGlobalExitRoot()`** in **`PolygonZkEVMBridge`**;
- **`setForceBatchTimeout()`** and **`activateForceBatches()`** in**`PolygonZkEVM`** to switch on the ForceBatches mechanism on a later moment;
- Several fixes and optimizations.

## 6.3 Results

No security issues have been found in the above mentioned changes.

### 6.3.1 Mainnet deployment

The following deployment addresses on ethereum Mainnet where checked:

- PolygonZkEVM proxy
- PolygonZkEVM implementation
- FflonkVerifier
- PolygonZkEVMBridge proxy
- PolygonZkEVMBridge implementation
- PolygonZkEVMGlobalExitRoot proxy
- PolygonZkEVMGlobalExitRoot implementation
- PolygonZkEVMDeployer
- PolygonZkEVMTimelock
- ProxyAdmin

### 6.3.2 Findings

Some of the contracts have extra junk code in the etherscan verification. This is not a security risk, but could be a reputation risk. This is present in the following contracts:

- PolygonZkEVMBridge proxy
- ProxyAdmin

Different proxy contracts are used, two based on Solidity 0.8.9 and one based on Solidity 0.8.17. This is not a security risk, but it is less consistent.

- PolygonZkEVM proxy uses Solidity v0.8.9
- PolygonZkEVMGlobalExitRoot Proxy uses Solidity v0.8.9
- PolygonZkEVMBridge proxy uses Solidity v0.8.17

**Polygon:** Etherscan does not currently let you verify your code if a verification matching that same bytecode already exists. We will contact Etherscan to allow verification without junk code.

The PolygonZkEVMBridge is deployed via PolygonZkEVMDeployer to allow deployment with the same address on all chains. This is why this proxy uses Solidity 0.8.17. Otherwise the proxies are the standard version.

#### 6.3.3 L2 deployment

On the L2 (e.g. the zkEVM itself), the following contracts are deployed.

- L2 PolygonZkEVMBridge proxy
- L2 PolygonZkEVMBridge implementation
- L2 PolygonZkEVMGlobalExitRootL2 proxy
- L2 PolygonZkEVMGlobalExitRootL2 implementation
- L2 PolygonZkEVMDeployer
- L2 PolygonZkEVMTimelock
- L2 ProxyAdmin

#### 6.3.4 Findings

The deployments on L2 where not checked because ZKEVM Polygonscan isn't available yet and the contracts are not yet verified.