Polygon

LXLY Bridge

Smart Contract Security Assessment

Version: 2.1

February, 2024

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Polygon smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Polygon smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Polygon smart contracts.

Overview

Polygon zkEVM is a Layer 2 network zero-knowledge (ZK) rollup scaling solution designed to work with the Ethereum Virtual Machine (EVM). As a zkEVM, it is intended to support the deployment of smart contracts written for the EVM while providing scaling in the ZK prover. Due to the ZK prover, there is a faster security consensus achieved than with other optimistic rollup designs.

Faster consensus enables faster bridging between Polygon zkEVM and other Layer 2s or Ethereum Mainnet, this is natively supported via the Polygon LXLY bridge. The LXLY bridge enables cross-chain communication between various Polygon chains and/or the Ethereum Mainnet.

Consensus on Polygon zkEVM is achieved by two distinct roles, the Sequencer and Aggregator. At current these roles are trusted actors but Polygon has mechanisms in place to force transaction inclusion as a method of censorship resistance. Users submit transactions to the Sequencer who then pushes these transaction batches on-chain for verification by the Aggregator. The Aggregator generates a ZK proof for a batch and publishes it on-chain, completing the process.

Security Assessment Summary

This review was conducted on the files hosted on the Polygon zkEVM repository and were assessed at commit 1403e2b.

Retesting activities were performed on commit 7677b8c.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 19 issues during this assessment. Categorised by their severity:

- Medium: 7 issues.
- Low: 4 issues.
- Informational: 8 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Polygon smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
LXLY-01	Empty Committee Is Accepted In Validium Setup	Medium	Resolved
LXLY-02	Aggregator Rewards Are Averaged Across Forced Batches & Rollups	Medium	Closed
LXLY-03	Batch Fees Multiplier Cap Bypassed With Multiple Calls	Medium	Closed
LXLY-04	Missing `ifNotEmergencyState` Modifier	Medium	Resolved
LXLY-05	Rollups Can Be Overwritten Without Checks	Medium	Resolved
LXLY-06	Empty Forced Batch Event Will Look Like It Came From An EOA	Medium	Closed
LXLY-07	Token Wrapped Creation Code Is Not Deterministic	Medium	Resolved
LXLY-08	Timeouts Can Expire During Emergency State	Low	Resolved
LXLY-09	Lack Of Slippage Protection For Sequencer	Low	Closed
LXLY-10	Sequencer Can Censor Forced Batches	Low	Closed
LXLY-11	EIP-155 Transactions Are Replayable	Low	Closed
LXLY-12	Contract Address Could Change Domain Separator	Informational	Closed
LXLY-13	Emergency Mode Activates For All Rollups	Informational	Closed
LXLY-14	Batch Fees Only Updated On Non-Trusted Aggregator Verifications	Informational	Closed
LXLY-15	Forced Batch Fee Initialized To Wrong Value	Informational	Closed
LXLY-16	Pending Admin Can Accept Repeatedly	Informational	Closed
LXLY-17	Soundness Check Functions Should Trigger Emergency State	Informational	Closed
LXLY-18	`permit()` Calls Are Front-Runnable	Informational	Closed
LXLY-19	Miscellaneous General Comments	Informational	Resolved

LXLY-01	Empty Committee Is Accepted In Validium Setup		
Asset	`CDKDataCommittee.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Setting up the Validium rollup signing committee is achieved by calling `setupCommittee()`. Due to a lack of input validation it is possible to register an empty committee.

An empty committee occurs when following parameters are submitted.

- `_requiredAmountOfSignatures = 0`
- `urls.length = 0`
- `addrsBytes.length = 0`

Any existing members will be removed and no new members are added. The result is that any sequenced batches will pass the Validium signatures checks without the need for a committee to sign the message.

The issue may be exploited since `CDKDataCommittee.verifySignatures()` is called with `requiredAmountOfSignatures = 0`. Thus, passing an empty `signaturesAndAddrs` will successfully verify the `signedHash`.

Recommendations

When setting up a signing committee `setupCommittee()` should include a check that `membersLength` is non-zero to prevent deleting the existing committee and use of an empty committee.

Resolution

The development team is aware of this use case and has chosen to leave the code as is since it is a desirable feature. To ensure the update function is called with the correct parameters, additional comments have been added to `setupCommittee()` as follows.

> *It is advised to use timelocks for the admin address in case of Validium since it can change the* `dataAvailabilityProtocol`.

By using a timelock and multisig contract as access control for `setupCommittee()` the likelihood of setting incorrect parameters is significantly reduced.

LXLY-02	Aggregator Rewards Are Averaged Across Forced Batches & Rollups		
Asset	`PolygonRollupBase.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

There is a single batch fee shared across all rollups. It is set based off throughput, increasing when there is high throughput and decreasing when there is low throughput. Since there is only a single batch fee, rollups with large throughput will increase the batch fee for rollups with low throughput.

The sequencer will only include batches if there is sufficient reward in the gas fees and MEV to cover the batch fee. Hence, when the batch fee is increasing the sequencer will increase the L2 gas price users must pay to have their transaction included in a block. Therefore, if there is a single rollup which has high throughput this will increase gas fee paid by users on all rollups.

Additionally, aggregator rewards are based off the contract's POL balance divided by the number of sequenced batches but not aggregated. The rewards distributed for aggregating a batch are not directly matched to the fee charged for the batch.

```
function calculateRewardPerBatch() public view returns (uint256) {
    uint256 currentBalance = pol.balanceOf(address(this));

    // Total Batches to be verified = total Sequenced Batches - total verified Batches
    uint256 totalBatchesToVerify = totalSequencedBatches -
        totalVerifiedBatches;

    if (totalBatchesToVerify == 0) return 0;
    return currentBalance / totalBatchesToVerify;
}
```

The function `calculateRewardPerBatch()` calculates aggregator rewards by taking the total balance of the contract and dividing it by the total number of sequenced batches. Thus, the fee paid when sequencing a batch is not directly aligned with the amount rewarded to the aggregator for verifying a batch.

Forced batches will incur a higher sequence batch fee, to which the aggregator will not receive the entirety of the forced batch fee. Furthermore, if the batch fee has increased or decreased significantly from when the batch was sequenced the aggregator will not receive exactly the original fee paid.

Recommendations

It is recommended to add further modularisation such that there is a batch fee for each rollup. Additionally, this would require to internally account for the POL fees contributed by each rollup. Thus, when an aggregator verifies a batch they will receive rewards only associated for that rollup.

Keeping track of forced batch fees awaiting sequencing would enable accurate transmission of the forced batch fee to the `PolygonRollupManager` contract. However, the testing team acknowledges that including this extra logic will complicate the sequencing fee system and increase gas costs as a result. Therefore, as forced batches are likely to be rare, it may also be adequate to document the behaviour of forced batch rewards to make aggregators aware.

Resolution

The development team has acknowledged the comments above and decided to close the issue with the following rationale:

- The fee pricing mechanism works as intended. A higher throughput would result in more machines being spun up to aggregate in parallel and would thus not result in an increased batch fee. Additionally, there are several fallback mechanisms to fix any issues in case they arise.
- The (high) force batch fee is intended as a disincentive to the user, not as an incentive to the aggregator. As such, the high fee can be used to accelerate the aggregation of the following batches.
- Having one common pool of POL is useful since it smoothes out the rewards for the aggregators. Furthermore, it makes it easy to incentise faster aggregation by sending POL to the pool.

LXLY-03	Batch Fees Multiplier Cap Bypassed With Multiple Calls		
Asset	`PolygonRollupManager.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The aggregators who verify batches by posting zero-knowledge proofs on-chain are compensated by a fee paid by the sequencer. This fee is updated when a non-trusted aggregator verifies a batch so that it remains competitive. Depending on if more batches are late or early the `_batchFee` will be adjusted down or up. There is a maximum multiplier size to prevent the batch fee moving too much in one verification call. However, it is trivial to bypass this maximum multiplier.

An attacker will check if the number of new batches to be verified exceeds `_MAX_BATCH_MULTIPLIER` in size, if so they split the total batches to verify into `_MAX_BATCH_MULTIPLIER` segments. By splitting verification into segments with `_MAX_BATCH_MULTIPLIER` batches, an aggregator can ensure they receive the maximum increase in batch fees. However, the L1 gas fees will increase for the aggregator as the FFLONK verification must occur for each new segment.

When more than `_MAX_BATCH_MULTIPLIER` are available for sequencing and are late, the aggregator has the option to pay more L1 gas fees and increase the batch fee. This would increase future rewards for all aggregators.

Likewise, when an aggregator detects that early batches have occurred which will decrease batch fee they may include the maximum possible number of batches in one transaction to limit the decrease. However, in a decentralised aggregator environment, delaying submitting batches to limit the fee decrease may result in other aggregators first calling `verifyBatches()` to claim the rewards.

This will result in less control over batch fees paid by the sequencer and may lead to the sequencer not realising a profit in extreme cases. As a knock-on effect the sequencer is then less financially motivated to batch user transactions.

Recommendations

Polygon should consider different mechanisms for how to prevent the batch fee fluctuating excessively. Some potential methods are:

- Limiting batch fee changes per block or timestamps. For example only allowing `_MAX_BATCH_MULTIPLIER` number of fee increases per block, OR
- Limiting batch fee changes per total number of batches verified. For example, only allowing `_MAX_BATCH_MULTIPLIER` number of fee increases per 100 aggregated batches.

Resolution

The development team has acknowledged these comments and decided to close the issue with the following rationale:

It is not clear this is a profitable attack, the malicious aggregator would have to pay a significantly increased gas fee for a marginally higher reward. Additionally, an aggregator is not incentivised to increase the batch fee as this would result

in higher L2 fees and thus lower throughput. Leading to less batches that need aggregation and lower income over the long term.

LXLY-04	Missing `ifNotEmergencyState` Modifier		
Asset	`PolygonZkEVMBridgeV2.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The system contains an emergency mode which pauses functionality accross multiple contracts. This emergency mode can be activated manually or through the discovery of a soundness bug to protect assets during a security incident.

Most importantly, functionality for the `claimAsset()` and `claimMessage()` functions is paused when the emergency state is active. For consistency, `bridgeAsset()`, `bridgeMessage()` and `bridgeMessageWETH()` are also meant to be paused. However, for these latter two functions the `ifNotEmergencyState` modifier is missing, meaning that they will continue to function when the emergency state is active.

Recommendations

It is recommended to add the `ifNotEmergencyState` modifier to `bridgeMessage()` and `bridgeMessageWETH()` functions in `PolygonZkEVMBridgeV2.sol`.

Resolution

This issue has been addressed in commit 0f57a109. `ifNotEmergencyState` modifiers were added to `bridgeMessage()` and `bridgeMessageWETH()` functions.

LXLY-05	Rollups Can Be Overwritten Without Checks		
Asset	`PolygonRollupManager.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

It is possible to overwrite an existing rollup by calling `addExistingRollup()` with a rollup address relating to an existing rollup. This then allows the user to arbitrarily overwrite the other parameters stored about a rollup in `PolygonRollupManager`.

If done on an existing rollup, then `rollupAddressToID` will be overwritten for this address and cannot be reset to its original value without a contract upgrade.

It is worth noting that this is an access-controlled function that is normally only accessible to the timelock contract and requires a different `chainId` to the existing value to succeed. Hence, the risk of this bug occurring is low.

Recommendations

Checks should be added to `addExistingRollup()` to ensure the rollup address does not already exist in the system. This can be done by checking the `rollupAddressToID` mapping to ensure it returns a zero `rollupID`.

Resolution

The development team has acknowledged these comments and added extra checks to `addExistingRollup()` in PR #167.

LXLY-06	Empty Forced Batch Event Will Look Like It Came From An EOA		
Asset	`PolygonRollupBase.sol`, `PolygonRollupBaseEtrog.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

If a batch is forced with `transactions.length = 0`, it will emit the same event with the same data whether `msg.sender == tx.origin` is true or false.

Since the `msg.sender` is not equal to `tx.origin` the batch data will not be available in the transaction calldata. Instead, any attack controlled data will exist in the transaction calldata.

The comments around line [**632-636**] indicate that this may cause issues with the off-chain monitoring. When attempting to fetch the batch transaction data from the transaction calldata it will return incorrect results causing the aggregators to halt.

Recommendations

To remediate this issue enforce `transactions.length > 0` in `forceBatch()`.

Resolution

The development team acknowledges this comment and decided to close the issue with the following rationale:

The node that processes this transaction will know the values of `msg.sender` and `tx.origin`. As a result, it knows if the transaction comes from a contract or an EOA and can handle the transaction accordingly. Additionally, the case of the empty batch is actually valid and may be useful in certain situations. As such, it should be supported.

LXLY-07	Token Wrapped Creation Code Is Not Deterministic		
Asset	`PolygonZkEVMBridgeV2.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

The creation code for the smart contract `TokenWrapped` will change when using a new compiler or modifying any metadata associated with the contract.

The variable `type(TokenWrapped).creationCode` is used to deterministically calculate the address of wrapped ERC20 tokens. However, when upgrading to version 2 of the bridge the value of `type(TokenWrapped).creationCode` will change. This occurs due to two reasons, first a new compiler is used which outputs different bytecode and second, the smart contract metadata such as directory structure has changed and this will be included in the bytecode.

The impact is that wrapped token addresses on other chains can no longer be pre-calculated. Furthermore, `precalculatedWrapperAddress()` will not return the correct address.

Recommendations

To handle the different values of `type(TokenWrapped).creationCode` use a constant bytes variable which contains the creation code of the version 1 contracts. This constant bytes variable will need to be used every time a new `TokenWrapped` contract is to be deployed. Solidity does not directly allow deploying from constant bytes, therefore it will need to be done in assembly using the `CREATE2` opcode.

Resolution

The development team has acknowledged the above comments and provided a fix in PR #164 by creating a constant variable that contains the creation code.

LXLY-08	Timeouts Can Expire During Emergency State		
Asset	`PolygonRollupManager.sol`, `PolygonRollupBase.sol` & `PolygonRollupBaseEtrog.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The measurement of timeouts do not pause when the emergency state is active. This can lead to timeouts being inaccurately fully expired once the emergency state is deactivated.

Specifically, `trustedAggregatorTimeout` in `PolygonRollupManager.verifyBatches()` could be passed, allowing any address to verify batches. `forceBatchTimeout` in `PolygonRollupBase.sequenceForceBatches()` could also have passed, allowing forced batches to be immediately sequenced by any address.

Note that new forced batches can be submitted during an emergency state, so it is possible for batches to be forced and verified by an external party without either the trusted sequencer or the trusted aggregator having had a chance to process them.

This issue has been anticipated and there are some mitigations present:

- `trustedAggregatorTimeout` is automatically set to `_HALT_AGGREGATION_TIMEOUT` (a high value) in `overridePendingState()`.
- `trustedAggregatorTimeout` can be increased during the emergency state in `setTrustedAggregatorTimeout()`.
- `forceBatchTimeout` can be increased during the emergency state in `setForceBatchTimeout()`.

In spite of these mitigations, the potential for an expired timeout after an emergency state is still present if admin roles do not extend the timeout periods before ending an emergency state.

Also, in the case of `forceBatchTimeout`, the maximum value is capped at `_HALT_AGGREGATION_TIMEOUT` (1 week), so an emergency state longer than that would allow forced batches to be sequenced immediately on deactivation of the emergency state.

Recommendations

Consider overriding `EmergencyManager._deactivateEmergencyState()` to check that there is still a minimum amount of time remaining on the timeouts before allowing emergency state to be deactivated. This would also require it to be possible to set `forceBatchTimeout` to a value greater than `_HALT_AGGREGATION_TIMEOUT` if there is an emergency state.

Furthermore, prevent `forceBatch()` from being called during an emergency state.

Resolution

The development team acknowledges the above comments and has mitigated the issue for `forceBatchTimeout` in PR #166. For `trustedAggregatorTimeout` the team deemed this a non-issue as it is up to the trusted aggregator to aggregate all batches before deactivating the emergency mode.

LXLY-09	Lack Of Slippage Protection For Sequencer		
Asset	`PolygonRollupBase.sol` & `PolygonRollupBaseEtrog.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

There is lack of slippage protection in the function `sequenceBatches()`.

Within the function `sequenceBathces()`, the trusted sequencer will be charged a fee of POL according to the following lines of code.

```
// Pay collateral for every non-forced batch submitted
pol.safeTransferFrom(
    msg.sender,
    address(rollupManager),
    rollupManager.getBatchFee() * nonForcedBatchesSequenced
);
```

It is possible for `rollupManager.getBatchFee()` to increase significantly when an aggregator verifies multiple batches. One such attack an aggregator can perform to rapidly increase fees is described in LXLY-03.

This would cause the sequencer to pay significantly more in fees than intended and may cause the batch to be sequenced at a loss.

Recommendations

To resolve the issue add the parameter `maxFee` to `sequenceBathces()`, which represents the maximum amount of POL tokens the sequencer is willing pay. If `rollupManager.getBatchFee() * nonForcedBatchesSequenced` is larger than `maxFee` revert the transaction.

This will provide protection to the sequencer against rapid increases in the cost of batch fees.

Resolution

The development team has acknowledged the above comment and decided to close this issue with the following response:

> *The sequencer might lose some slippage calling this function, but delaying the sequence transaction could have an even worse impact on the network. This delay means adding delay to the finalization of the transaction and the bridges, which results in a poor UX. The sequencer can compensate the slippage in future transaction fees.*

LXLY-10	Sequencer Can Censor Forced Batches		
Asset	`PolygonRollupBase.sol` & `PolygonRollupBaseEtrog.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

It is possible for a sequencer to partially censor a forced batch at the cost of one batch fee and lost revenue of the forced batch.

When there are multiple transactions in a forced batch it is possible for a sequencer to extract one of the transactions from a forced batch and include it in a separate batch. During execution the transactions will increment the nonce of the signer account, thereby invalidating in future transactions with this nonce.

After the forced batch is sequenced it will contain a transaction which signed a consumed nonce. Thus, due to the invalid nonce the batch will be reverted.

The cost to the sequencer is less than a single batch fee to sequence one of the transactions of the forced batch. Noting here that the sequencer can fill this batch with other valid transaction to gain gas fees and MEV revenue. Additionally, the sequencer will forgo and rewards received in the forced batch which are likely negligible.

The impact is that only a single transaction of a forced batch would be executed, the remaining transactions would be reverted.

Note this issue works under the assumption that an invalid nonce will cause an entire batch to be reverted.

Recommendations

Consider skipping transactions with an invalid nonce but valid signature on the zkEVM validation rather than reverting the batch. This would allow transactions to be sequenced multiple times but would prevent a malicious sequencer from censoring forced batches.

Resolution

The development team has acknowledged the above comment and closed the issue since an invalid nonce will only cause that transaction to be invalid, not the entire batch.

LXLY-11	EIP-155 Transactions Are Replayable		
Asset	/*		
Status	**Closed:** Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The Polygon zkEVM supports EIP-155 transactions.

EIP-155 transactions do not sign over a chain ID or any data restricting the transaction for use on a single chain. As a result, it is possible to take a transaction used on one chain and execute it on any other chain that supports EIP-155 transactions. For the replay to be successful it must include a nonce that matches the account nonce and have sufficient funds.

One of the most common examples is that all transactions before EIP-155 was implemented on Ethereum mainnet did not include the chain ID. These transactions can be replayed on any other chain which supports EIP-155.

Recommendations

It is recommended to avoid supporting EIP-155 to prevent transaction replay across chains.

Resolution

The development team acknowledge the above comment and decided to close this issue with the following response:

> *EIP155 is intentionally supported like in Ethereum because it provides some functionalities like keyless deployments which allow for having the same create2 factories that some known projects use like gnosis safe.*

LXLY-12	Contract Address Could Change Domain Separator
Asset	`TokenWrapped.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

In `TokenWrapped.DOMAIN_SEPARATOR()`, the chain ID is checked and, if it is different than the deployment chain ID, the domain separator is recalculated. This is to prevent signed permits with the wrong chain ID being generated, which could then be reused on the chain with that ID.

It is conceivable, however, that the address of the contract could also change. Layer 2's often adjust the addresses of a smart contract but leave the contract bytecode unchanged. In this case, the cached domain separator, which references `address(this)` on deployment, would be potentially reusable at the original address. Or permits could be generated at the original address which would be valid at the new token address.

It is worth noting that if a layer 2 protocol were to perform this change it is likely that the `chainId` would be modified and therefore this issue would not occur.

Recommendations

Consider adding a check in `TokenWrapped.DOMAIN_SEPARATOR()`. Check the current and deployment values of `address(this)` as well as those of `block.chainid` before returning the cached domain separator `_DEPLOYMENT_DOMAIN_SEPARATOR`.

See the OpenZeppelin implementation as a guide.

Resolution

The development team has acknowledged the above comment and is aware of this edge case to prevent potential issues in the future.

LXLY-13	Emergency Mode Activates For All Rollups
Asset	`PolygonRollupManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `RollupManager` contract contains an emergency mode which pauses batch sequencing when a flaw in the verifier system's soundness is detected. This emergency mode applies for all rollups the manager handles, meaning that if the emergency mode is triggered for one rollup then it will block sequencing of new transactions in all rollups.

This means that no new transactions can be sequenced or verified on any rollup managed by the `RollupManager` until the soundness violation is dealt with by the trusted aggregator and the rollup is deemed safe to resume operation. As this process can only be triggered by the admin and would take some time it would result in loss of service to all smart contracts on other rollups connected to the RollupManager despite them potentially having no soundness flaws. In addition to this, the same emergency mode is used by the LXLY bridge and would inhibit bridging or claiming tokens.

Recommendations

It is recommended that the emergency system be specific to different rollups. That way if one rollup is placed into emergency mode it will not impact processing transactions on other Polygon rollups.

If different rollups share the same verifier then it is logical to group these rollups together under the same emergency system as a soundness error in one would indicate a wider problem in all rollups using the same verifier.

Resolution

The development team acknowledges the above comment and closed the issue with the following response:

> *The emergency mode should be super unlikely and would mean that a high vulnerability has been found in the system. Probably the most cautious thing to do is to stop everything until it's fixed.*

LXLY-14	Batch Fees Only Updated On Non-Trusted Aggregator Verifications
Asset	`PolygonRollupManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The aggregators who verify batches by posting zero-knowledge proofs on-chain are compensated by a fee paid by the sequencer. This fee is only updated when a non-trusted aggregator verifies a batch. As a result the batch fee paid can remain too low or high for sustained periods.

If compensation is too low aggregators may choose to not verify batches leading to a poor user experience for the rollup effected, as transaction finalisation increases significantly. On the other hand if the batch fee is too high then the sequencer will continuously overpay for batch verification, reducing their compensation for sequencing batches.

Note that the trusted aggregator would then be incentivised to always submit late batches to increase the batch fee for future rewards.

Recommendations

Consider calling `_updateBatchFee()` in `verifyBatchesTrustedAggregator()`, this way the batch fees will always remain up-to-date. However, this will change the level of trust associated with the trust aggregator as it will now be financially motivated to delay batch verification.

Alternatively, the functionality can be achieved from the `SET_FEE_ROLE` permissioned user calling `setBatchFee()` during times when aggregation is restricted to a trusted aggregator.

Resolution

The development team has acknowledged and closed this issue with the following comment:

> *This is intended to avoid issues where the fees randomly increase/decrease without making much sense. Since we as the admins can change the batchFee of the system, updating the fee on trusted aggregation only adds gas to the verification and some randomness on the fees.*

LXLY-15	Forced Batch Fee Initialized To Wrong Value
Asset	`PolygonRollupManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The Polygon team communicated that the initial fee for forced batches should be 100 POL tokens but because `_batchFee = 0.1 ether` on line [399] and the forced batch fee is 100 times this value the initial forced batch fee will instead be 10 POL.

Recommendations

The Polygon team should determine if the 10 POL forced batch fee will satisfy their economic design and if not, they should consider changing the initial value that `_batchFee` is set as to remedy this.

Resolution

The development team has acknowledged the above comment and clarified that 10 POL is the intended forced batch fee.

LXLY-16	Pending Admin Can Accept Repeatedly
Asset	`PolygonRollupBase.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The pending admin can accept the admin transfer more than once, triggering the `AcceptAdminRole` event each time.

This is because the storage variable `pendingAdmin` remains unchanged after a successful call to `acceptAdminRole()` by a pending admin. Standard practice for functions such as this would be to set `pendingAdmin` to `address(0)` after the pending admin has been successfully confirmed as admin.

Recommendations

Consider setting `pendingAdmin` to `address(0)` after a successful call to `acceptAdminRole()` by a pending admin.

Resolution

The development team has acknowledged the above comment and closed the issue with the following response:

> *It was implemented this way in order to save gas. Since it does not have security implications we think it's fine.*

LXLY-17	Soundness Check Functions Should Trigger Emergency State
Asset	`PolygonRollupManager.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Both `proveNonDeterminisitcPendingState()` and `overridePendingState()` include calls to `_proveDistinctPendingState()` which validates that a soundness violation has occurred within the proving system.

However, only `proveNonDeterminisitcPendingState()` triggers the emergency state. Given a soundness violation likely indicates a wider issue with the prover code it is advisable to trigger the emergency state whenever a soundness violation is detected.

Recommendations

`overridePendingState()` should cause emergency state to become active. If soundness violations are intended to only be reported via `proveNonDeterminisitcPendingState()` then `overridePendingState()` should only be accessible during the emergency state to enforce the correct function call ordering.

Resolution

The development team acknowledged the above comment and closed this issue, clarifying that the trusted aggregator should have the option to not trigger the emergency system if desired.

LXLY-18	`permit()` Calls Are Front-Runnable
Asset	`PolygonZkEVMBridgeV2.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Any call to `permit()` for ERC20 tokens is front-runnable. This may cause a small griefing attack to `bridgeAsset()` if it is called with non-empty `permitData`.

The `permit()` function for ERC20 tokens is defined in EIP 2612. Any user is able to call the function with a valid signature and it will set the allowance for a `spender` and `signer`. However, there is no restriction on who may call the `permit()` function.

Signatures are only valid for a single call to `permit()` due to an incrementing nonce.

The issue present in `PolygonZkEVMBridgeV2` is that if a user calls `bridgeAsset()` with `permitData` it is possible for an attacker to extract this signature in the mempool. The attacker may then directly call the `permit()` function on the ERC20 token and consume the nonce. The user's call to `bridgeAsset()` will then revert due to invalid permit data.

Note that the attacker will have successfully called `permit()` and therefore the user will have approved the bridge for the correct amount. Therefore, if the user repeats the call to `bridgeAsset()` except changing `permitData = b""` the transaction will succeed.

Recommendations

As this attack only allows griefing a single transaction call to `bridgeAsset()` and the attacker will have to pay gas fees, there is little benefit to the attacker. Thus, this issue is raised as informational severity and the testing team does not encourage any modifications to the code.

If the development team are looking to resolve this issue it is possible to call `token.permit()` in assembly and continue executing if a revert occurs. Since the `safeTransferFrom()` will revert if there is insufficient balance it is not required to ensure `permit()` succeeds.

Resolution

The development team acknowledges the above comments and clarifies that this is a non-issue since the call to `token.permit()` is a low-level call and should thus not revert on failure.

LXLY-19	Miscellaneous General Comments
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Unclear comments**

   Numerous NatSpec comments are missing parameters, contain typos or poor grammar, some are duplicates of field descriptions that are meant for another variable. The comments of all files should be checked to ensure their meanings are clear.

2. **Proxy implementation contracts lack disablers**

   The `PolygonRollupManager` and `PolygonZkEVMBridgeV2` contracts should contain `disableInitializers()` function calls inside their constructors to adhere to best practices for OpenZeppelin proxy initializers.

3. **Set bool function**

   It's recommended to pass in a variable bool newState rather than toggle it, `PolygonDataComittee.switchSequenceWithDataAvailability()` is one such function. Passing in a variable helps prevent changing the bool state if the function is accidentally called multiple times in a row.

4. **Cache variables after updating them**

   In `PolygonZkEVMGlobalExitRootV2.sol`, there are two cached variables, `cacheLastRollupExitRoot` and `cacheLastMainnetExitRoot`. Both are defined and then immediately modified. It would save gas and be functionally identical to simply define them after the modification, around what is currently line [**57**].

```
47  bytes32 cacheLastRollupExitRoot = lastRollupExitRoot;
    bytes32 cacheLastMainnetExitRoot = lastMainnetExitRoot;
49
    if (msg.sender == bridgeAddress) {
51      lastMainnetExitRoot = newRoot;
        cacheLastMainnetExitRoot = newRoot;
53  } else if (msg.sender == rollupManager) {
        lastRollupExitRoot = newRoot;
55      cacheLastRollupExitRoot = newRoot;
```

5. **Zero POL transfer when only forced batches are sequenced**

   `PolygonRollupBase.sequenceBatches()` and `PolygonRollupBaseEtrog.sequenceBatches()` both have an unconditional call to transfer POL tokens (on line [**554**] and line [**536**] respectively). If there are only forced batches being sequenced, this transfer is for zero POL as `nonForcedBatchesSequenced` will be zero. Consider putting this call inside a block with the condition `if (nonForcedBatchesSequenced > 0)`.

6. **No custom errors**

   The comment on line [**16**] of `PolygonTransparentProxy.sol` states that custom errors have been added to the file, but no custom errors are present.

7. **Unused struct**

   Both `PolygonRollupBase` and `PolygonRollupBaseEtrog` define a struct `SequencedBatchData` which is not used and can be removed.

8. **Update comment for Etrog**
   The description of `accInputHash` on line [**51**] of `LegacyZKEVMStateVariables.sol` only covers the specification for `PolygonRollupBase`. Consider updating this to include the specification for `PolygonRollupBaseEtrog` and also `PolygonRollupBaseEtrog` forced batches.

9. **Stale interface used**
   `PolygonZkEVMGlobalExitRootV2` has no interface and in files such as `PolygonRollupManager` it is represented by `IBasePolygonZkEVMGlobalExitRoot` while this causes no issues currently it is advisable to keep interfaces matching the contract they are calling.

10. **Duplicate function body**
    `PolygonDataComittee.sequenceBatchesDataComittee()` only differs from `PolygonRollupBase.sequenceBatches()` by the call to `dataCommittee.verifySignatures()` on line [**197**] and checking of transaction data. Consider modifying the base contract such that it may handle additional logic.

11. **Event parameters have multiple purposes**
    The event `BridgeEvent` in the contract `PolygonZkEVMBridgeV2` is used in both functions `bridgeAsset()` and `_bridgeMessage()`. The parameter `originAddress` has multiple meanings depending which function calls this event. For `bridgeAsset()` it represents the address of the token being bridged, for `_bridgeMessage()` it represents the address of the sender. Likewise, the `metadata` and `originNetwork` fields have different meanings. It is advisable to create two different events so that the meaning of each field is consistent and clear.

12. **Unclear event parameter name**
    The event `BridgeEvent` contains a parameter `depositCount`. This parameter signifies the index of the deposit in the tree and is used as such off-chain. As such, it is recommended to rename this event (e.g.: to `depositIndex`) to highlight its indexing nature (starts at 0) instead of a counter (starts at 1).

13. **Inaccurate comment**
    `PolygonZkEVMGlobalExitRootV2.sol` contains a mapping `globalExitRootMap` that maps all valid global exit roots to their respective blockhash. However, in the previous version `PolygonZkEVMGlobalExitRoot.sol`, global exit roots were mapped to their timestamp. Since `PolygonZkEVMGlobalExitRootV2.sol` is upgraded from `PolygonZkEVMGlobalExitRoot.sol` the `globalExitRootMap` mapping will contain both timestamps and blockhashes. The comment above this mapping conveys that it only contains blockhashes while it should be noted that it also contains timestamps.

14. **Constant defined in multiple contracts**
    `_HALT_AGGREGATION_TIMEOUT` is defined in both `PolygonRollupBase` and `PolygonRollupManager`, it is better to only define constants in one file to prevent the values differing in the event you update one and forget to update the other.

15. **Avoid reading storage variable in a loop**
    line [**131**] of `CDKDataCommittee.sol` will cause the storage variable `requiredAmountOfSignatures` to be read on each iteration of the loop. This is unnecessarily expensive in terms of gas. Consider copying `requiredAmountOfSignatures` into memory before the loop and checking this cached copy instead.

16. **Checking of global index is not strict**
    In `PolygonZkEVMBridgeV2` for either `claimAsset()` or `claimMessage()` the parameter `globalIndex` is not strictly checked. In agreement with the following comments found on line [**419-420**] it is possible to set bits in the range `[0:190]` and they will not be validated.

```
/*
...
* @param globalIndex Global index is defined as:
* [0:190] not checked, [191] mainnet flag, rollupIndex [192, 223], localRootIndex[224, 255]
...
*/
```

While no exploits were found to re-use nonces, it is recommended to strictly check all bits of a nonce. This will increase maintainability and decrease the likelihood of a nonce re-use attack.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The comments above have been acknowledged by the development team, and relevant changes actioned in PR #167 where appropriate.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `Forge` framework was used to perform these tests and the output is given below.

```
Running 1 test for test/EmergencyManager.t.sol:EmergencyManagerTest
[PASS] test_fullEmergency() (gas: 38786)
Test result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.76ms

Running 2 tests for test/PolygonTransparentProxy.t.sol:PolygonTransparentProxyTest
[PASS] test_ptpFallback_fallback() (gas: 55780)
[PASS] test_ptpFallback_upgrade() (gas: 45868)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 2.89ms

Running 6 tests for test/PolygonAccessControlUpgradeable.t.sol:PolygonAccessControlUpgradeableTest
[PASS] test_grantRole() (gas: 57848)
[PASS] test_legacyOwner() (gas: 12672)
[PASS] test_renounceRole() (gas: 38820)
[PASS] test_revokeRole() (gas: 72436)
[PASS] test_setRoleAdmin() (gas: 113181)
[PASS] test_setupRole() (gas: 20709)
Test result: ok. 6 passed; 0 failed; 0 skipped; finished in 2.14ms

Running 2 tests for test/PolygonDataCommitteeEtrog.t.sol:PolygonDataCommitteeEtrogTest
[PASS] test_SequenceBatches() (gas: 319870)
[PASS] test_sequenceBatchesDataCommittee() (gas: 330578)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 1.68s

Running 4 tests for test/CDKDataCommittee.t.sol:CDKDataCommitteeTest
[PASS] test_emptyCommittee() (gas: 44096)
[PASS] test_overwriteCommittee() (gas: 260697)
[PASS] test_setup() (gas: 503873)
[PASS] test_verifySignatures() (gas: 273038)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.84s

Running 4 tests for test/DepositContractV2.t.sol:DepositContractV2Test
[PASS] test_addLeaf() (gas: 301225)
[PASS] test_addMultipleLeaves() (gas: 841028)
[PASS] test_revertWhenFull() (gas: 114408)
[PASS] test_setUp() (gas: 161552)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.84s

Running 4 tests for test/DepositContract.t.sol:DepositContractTest
[PASS] test_addLeaf() (gas: 301249)
[PASS] test_addMultipleLeaves() (gas: 840986)
[PASS] test_revertWhenFull() (gas: 114408)
[PASS] test_setUp() (gas: 161576)
Test result: ok. 4 passed; 0 failed; 0 skipped; finished in 1.84s

Running 10 tests for test/PolygonRollupManager.t.sol:PolygonRollupManagerTest
[PASS] test_GlobalEmergencyMode() (gas: 513490)
[PASS] test_OverridePendingState() (gas: 629079)
[PASS] test_activateEmergencyState() (gas: 534595)
[PASS] test_consolidatePendingState() (gas: 801833)
[PASS] test_deactivateEmergencyState() (gas: 80805)
[PASS] test_maxSizeVerification() (gas: 19479822)
[PASS] test_overwriteRollupId() (gas: 34055)
[PASS] test_proveNonDeterministicPendingState() (gas: 408539)
[PASS] test_verifyBatches() (gas: 446476)
[PASS] test_verifyBatchesTrustedAggregator() (gas: 680450)
Test result: ok. 10 passed; 0 failed; 0 skipped; finished in 1.86s

Running 17 tests for test/PolygonZkEVMBridgeV2Existent.t.sol:PolygonZkEVMBridgeV2TestExistent
[PASS] test_ETHNativeBridgeETH() (gas: 9663160)
[PASS] test_ETHNativeBridgeMessage() (gas: 9783190)
[PASS] test_ETHNativeBridgeToken() (gas: 11118508)
[PASS] test_L2ToL2Bridge() (gas: 7599066)
```

```
[PASS] test_bridgeAndClaimETH() (gas: 2397632)
[PASS] test_bridgeAndClaimGasToken() (gas: 2771235)
[PASS] test_bridgeAndClaimMessage() (gas: 2531568)
[PASS] test_bridgeAndClaimMessageWETH() (gas: 2909863)
[PASS] test_bridgeWithPermit() (gas: 315449)
[PASS] test_cantClaimTwice() (gas: 2421316)
[PASS] test_claimOldDeposits() (gas: 2188611)
[PASS] test_claimWithOldRoot() (gas: 2475711)
[PASS] test_emergencyState() (gas: 372844)
[PASS] test_emergencyStateMessage_Vuln() (gas: 44268)
[PASS] test_initialized() (gas: 47122)
[PASS] test_multipleBridgeAndClaim() (gas: 3259599)
[PASS] test_precalculatedWrapperAddress() (gas: 3936801)
Test result: ok. 17 passed; 0 failed; 0 skipped; finished in 1.86s

Running 5 tests for test/TokenWrapped.t.sol:TokenWrappedTest
[PASS] test_tkwr_burn() (gas: 73381)
[PASS] test_tkwr_constructor() (gas: 17102)
[PASS] test_tkwr_decimals() (gas: 5523)
[PASS] test_tkwr_mint() (gas: 67730)
[PASS] test_tkwr_permit() (gas: 148078)
Test result: ok. 5 passed; 0 failed; 0 skipped; finished in 3.09ms

Running 2 tests for test/V1ToV2Upgrade.t.sol:V1ToV2UpgradeTests
[PASS] test_pendingStateLostOnUpgradeProtection() (gas: 11885968)
[PASS] test_revertingToV1Safely() (gas: 13200441)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 5.00ms

Running 25 tests for test/PolygonRollupBaseEtrog.t.sol:PolygonRollupBaseEtrogTest
[PASS] test_etrog_activateForceBatches() (gas: 31841)
[PASS] test_etrog_constructor() (gas: 23519)
[PASS] test_etrog_forceBatch_NotEnoughPOLAmount() (gas: 42337)
[PASS] test_etrog_forceBatch_TransactionsLengthAboveMax() (gas: 55809)
[PASS] test_etrog_forceBatch_vanilla() (gas: 164826)
[PASS] test_etrog_isForceBatchActive() (gas: 30810)
[PASS] test_etrog_onVerifyBatches() (gas: 31206)
[PASS] test_etrog_sequenceBatchesForced_ForcedDataDoesNotMatch() (gas: 178381)
[PASS] test_etrog_sequenceBatchesForced_vanilla() (gas: 256778)
[PASS] test_etrog_sequenceBatches_ExceedMaxVerifyBatches() (gas: 967412)
[PASS] test_etrog_sequenceBatches_OneBatch() (gas: 231049)
[PASS] test_etrog_sequenceBatches_OnlyTrustedSequencer() (gas: 46335)
[PASS] test_etrog_sequenceBatches_ThreeBatches() (gas: 252967)
[PASS] test_etrog_sequenceBatches_TransactionsLengthAboveMax() (gas: 586616)
[PASS] test_etrog_sequenceBatches_ZeroBatches() (gas: 19667)
[PASS] test_etrog_sequenceForceBatches_ExceedMaxVerifyBatches() (gas: 1081933)
[PASS] test_etrog_sequenceForceBatches_ForceBatchTimeoutNotExpired() (gas: 143967)
[PASS] test_etrog_sequenceForceBatches_ForceBatchesOverflow() (gas: 148536)
[PASS] test_etrog_sequenceForceBatches_ForcedDataDoesNotMatch() (gas: 143320)
[PASS] test_etrog_sequenceForceBatches_SequenceZeroBatches() (gas: 133901)
[PASS] test_etrog_sequenceForceBatches_vanilla() (gas: 210569)
[PASS] test_etrog_setForceBatchTimeout() (gas: 77901)
[PASS] test_etrog_setTrustedSequencer() (gas: 38083)
[PASS] test_etrog_setTrustedSequencerURL() (gas: 42396)
[PASS] test_etrog_transferAcceptAdminRole() (gas: 76725)
Test result: ok. 25 passed; 0 failed; 0 skipped; finished in 11.11ms

Running 25 tests for test/PolygonRollupBaseEtrogUpgraded.t.sol:PolygonRollupBaseEtrogUpgradedTest
[PASS] test_etrogUpgraded_activateForceBatches() (gas: 31807)
[PASS] test_etrogUpgraded_constructor() (gas: 23476)
[PASS] test_etrogUpgraded_forceBatch_NotEnoughPOLAmount() (gas: 42338)
[PASS] test_etrogUpgraded_forceBatch_TransactionsLengthAboveMax() (gas: 55810)
[PASS] test_etrogUpgraded_forceBatch_vanilla() (gas: 164826)
[PASS] test_etrogUpgraded_isForceBatchActive() (gas: 30767)
[PASS] test_etrogUpgraded_onVerifyBatches() (gas: 31143)
[PASS] test_etrogUpgraded_sequenceBatchesForced_ForcedDataDoesNotMatch() (gas: 178357)
[PASS] test_etrogUpgraded_sequenceBatchesForced_vanilla() (gas: 256779)
[PASS] test_etrogUpgraded_sequenceBatches_ExceedMaxVerifyBatches() (gas: 967412)
[PASS] test_etrogUpgraded_sequenceBatches_OneBatch() (gas: 231092)
[PASS] test_etrogUpgraded_sequenceBatches_OnlyTrustedSequencer() (gas: 46292)
[PASS] test_etrogUpgraded_sequenceBatches_ThreeBatches() (gas: 252944)
```

```
[PASS] test_etrogUpgraded_sequenceBatches_TransactionsLengthAboveMax() (gas: 586639)
[PASS] test_etrogUpgraded_sequenceBatches_ZeroBatches() (gas: 19647)
[PASS] test_etrogUpgraded_sequenceForceBatches_ExceedMaxVerifyBatches() (gas: 1081890)
[PASS] test_etrogUpgraded_sequenceForceBatches_ForceBatchTimeoutNotExpired() (gas: 143945)
[PASS] test_etrogUpgraded_sequenceForceBatches_ForceBatchesOverflow() (gas: 148582)
[PASS] test_etrogUpgraded_sequenceForceBatches_ForcedDataDoesNotMatch() (gas: 143341)
[PASS] test_etrogUpgraded_sequenceForceBatches_SequenceZeroBatches() (gas: 133921)
[PASS] test_etrogUpgraded_sequenceForceBatches_vanilla() (gas: 210548)
[PASS] test_etrogUpgraded_setForceBatchTimeout() (gas: 77900)
[PASS] test_etrogUpgraded_setTrustedSequencer() (gas: 38127)
[PASS] test_etrogUpgraded_setTrustedSequencerURL() (gas: 42351)
[PASS] test_etrogUpgraded_transferAcceptAdminRole() (gas: 59604)
Test result: ok. 25 passed; 0 failed; 0 skipped; finished in 1.93s

Running 2 tests for test/PolygonZkEVMGlobalExitRootV2.t.sol:PolygonZkEVMGlobalExitRootV2Test
[PASS] test_updateExitRoot() (gas: 1495516)
[PASS] test_upgrade() (gas: 2348281)
Test result: ok. 2 passed; 0 failed; 0 skipped; finished in 7.17ms

Running 16 tests for test/PolygonZkEVMBridgeV2.t.sol:PolygonZkEVMBridgeV2Test
[PASS] test_ETHNativeBridgeETH() (gas: 9622898)
[PASS] test_ETHNativeBridgeMessage() (gas: 9742993)
[PASS] test_ETHNativeBridgeToken() (gas: 11078246)
[PASS] test_L2ToL2Bridge() (gas: 7567096)
[PASS] test_bridgeAndClaimETH() (gas: 2354911)
[PASS] test_bridgeAndClaimGasToken() (gas: 2736532)
[PASS] test_bridgeAndClaimMessage() (gas: 2497398)
[PASS] test_bridgeAndClaimMessageWETH() (gas: 2875411)
[PASS] test_bridgeWithPermit() (gas: 352685)
[PASS] test_cantClaimTwice() (gas: 2381294)
[PASS] test_claimWithOldRoot() (gas: 2451875)
[PASS] test_emergencyState() (gas: 404433)
[PASS] test_emergencyStateMessage_Vuln() (gas: 39119)
[PASS] test_initialized() (gas: 43367)
[PASS] test_multipleBridgeAndClaim() (gas: 3242443)
[PASS] test_precalculatedWrapperAddress() (gas: 3899791)
Test result: ok. 16 passed; 0 failed; 0 skipped; finished in 1.85s

Ran 15 test suites: 125 tests passed, 0 failed, 0 skipped (125 total tests)
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

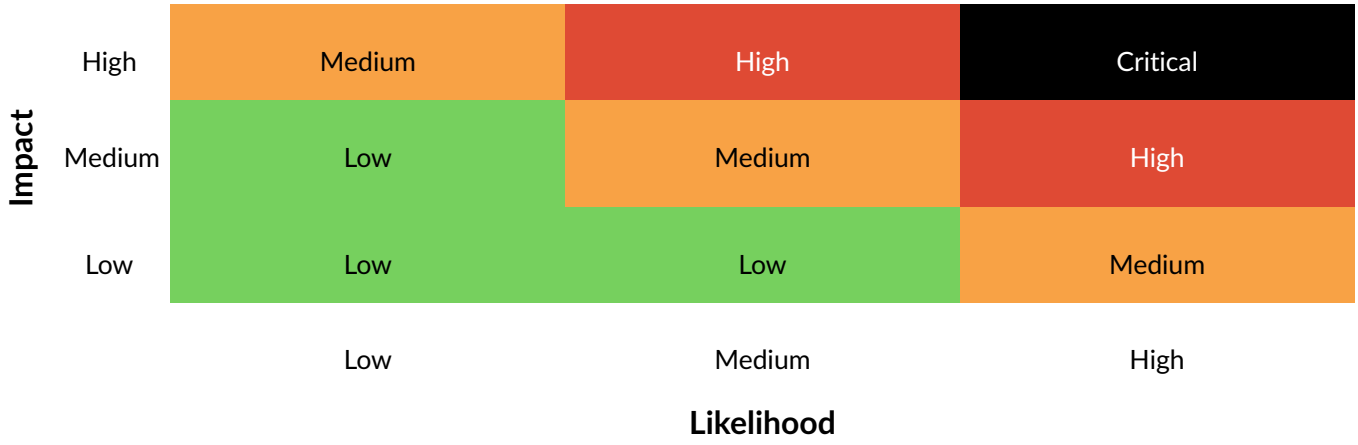

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].