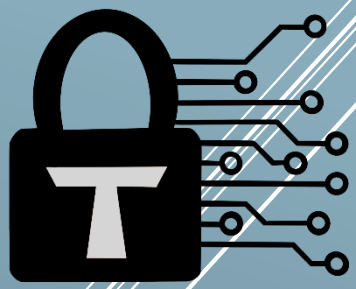
Trust Security



Smart Contract Audit

ZKsync Era Contracts

13/10/2024

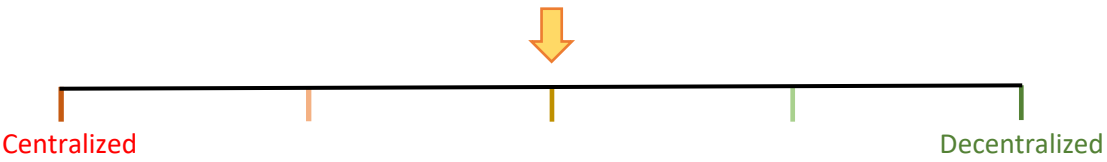
Executive summary

Category	ZK Rollup
Audited file count	117
Lines of Code	11843
Auditor	Trust, HollaDieWaldfee, Bernd Artmüller
Time period	17/06/2024 – 13/10/2024

Findings

Severity	Total	Fixed	Acknowledged
High	0	-	-
Medium	3	-	-
Low	23	-	-
Recommendation	47	-	-

Centralization score



Signature

Digitally signed by Trust
Date: 2024.10.13 21:34:27 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	13/10/2024	Client report

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security has conducted an audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Some additional recommendations have also been given when appropriate.

Scope

- l1-contracts/contracts/bridge/L1ERC20Bridge.sol
- l1-contracts/contracts/bridge/L1SharedBridge.sol
- l1-contracts/contracts/bridgehub/Bridgehub.sol
- l2-contracts/contracts/bridge/L2SharedBridge.sol
- l2-contracts/contracts/bridge/L2StandardERC20.sol
- l2-contracts/contracts/bridge/L2WrappedBaseToken.sol
- l1-contracts/contracts/state-transition/chain-deps/facets/Admin.sol
- l1-contracts/contracts/state-transition/chain-deps/facets/Executor.sol
- l1-contracts/contracts/state-transition/chain-deps/facets/Getters.sol
- l1-contracts/contracts/state-transition/chain-deps/facets/Mailbox.sol
- l1-contracts/contracts/state-transition/chain-deps/facets/ZkSyncHyperchainBase.sol
- l1-contracts/contracts/state-transition/chain-deps/ZkSyncHyperchainStorage.sol
- l1-contracts/contracts/state-transition/chain-deps/DiamondProxy.sol
- l1-contracts/contracts/state-transition/chain-deps/DiamondInit.sol
- l1-contracts/contracts/state-transition/StateTransitionManager.sol
- l1-contracts/contracts/state-transition/ValidatorTimelock.sol
- l1-contracts/contracts/state-transition/chain-interfaces/IZkSyncHyperchainBase.sol
- l1-contracts/contracts/state-transition/chain-interfaces/IZkSyncHyperchain.sol
- l1-contracts/contracts/state-transition/chain-interfaces/ITransactionFilterer.sol
- l1-contracts/contracts/state-transition/chain-interfaces/IMailbox.sol
- l1-contracts/contracts/state-transition/chain-interfaces/ILegacyGetters.sol
- l1-contracts/contracts/state-transition/chain-interfaces/IGetters.sol
- l1-contracts/contracts/state-transition/chain-interfaces/IExecutor.sol
- l1-contracts/contracts/state-transition/chain-interfaces/IAdmin.sol
- l1-contracts/contracts/state-transition/chain-interfaces/IDiamondInit.sol
- l1-contracts/contracts/state-transition/l2-deps/ISystemContext.sol
- l1-contracts/contracts/state-transition/IStateTransitionManager.sol
- l1-contracts/contracts/bridge/interfaces/IL1ERC20Bridge.sol
- l1-contracts/contracts/bridge/interfaces/IL1SharedBridge.sol
- l1-contracts/contracts/bridge/interfaces/IL2Bridge.sol
- l1-contracts/contracts/bridge/interfaces/IWETH9.sol
- l1-contracts/contracts/bridgehub/IBridgehub.sol
- l1-contracts/contracts/upgrades/IDefaultUpgrade.sol
- l2-contracts/contracts/bridge/interfaces/IL1ERC20Bridge.sol
- l2-contracts/contracts/bridge/interfaces/IL1SharedBridge.sol

- l2-contracts/contracts/bridge/interfaces/IL2SharedBridge.sol
- l2-contracts/contracts/bridge/interfaces/IL2StandardToken.sol
- l2-contracts/contracts/bridge/interfaces/IL2WrappedBaseToken.sol
- l2-contracts/contracts/interfaces/IPaymaster.sol
- l2-contracts/contracts/interfaces/IPaymasterFlow.sol
- l1-contracts/contracts/common/libraries/UnsafeBytes.sol
- l1-contracts/contracts/common/libraries/UncheckedMath.sol
- l1-contracts/contracts/common/libraries/SemVer.sol
- l1-contracts/contracts/common/libraries/L2ContractHelper.sol
- l1-contracts/contracts/state-transition/libraries/Diamond.sol
- l1-contracts/contracts/state-transition/libraries/LibMap.sol
- l1-contracts/contracts/state-transition/libraries/Merkle.sol
- l1-contracts/contracts/state-transition/libraries/PriorityQueue.sol
- l1-contracts/contracts/state-transition/libraries/TransactionValidator.sol
- l2-contracts/contracts/vendor/AddressAliasHelper.sol
- l1-contracts/contracts/common/Messaging.sol
- l1-contracts/contracts/common/ReentrancyGuard.sol
- l1-contracts/contracts/common/L2ContractAddresses.sol
- l1-contracts/contracts/common/Dependencies.sol
- l1-contracts/contracts/common/Config.sol
- l2-contracts/contracts/TestnetPaymaster.sol
- l2-contracts/contracts/SystemContractsCaller.sol
- l2-contracts/contracts/L2ContractHelper.sol
- l2-contracts/contracts/ForceDeployUpgrader.sol
- l2-contracts/contracts/Dependencies.sol
- l1-contracts/contracts/upgrades/BaseZkSyncUpgrade.sol
- l1-contracts/contracts/upgrades/BaseZkSyncUpgradeGenesis.sol
- l1-contracts/contracts/upgrades/DefaultUpgrade.sol
- l1-contracts/contracts/upgrades/GenesisUpgrade.sol
- system-contracts/bootloader/bootloader.yul
- system-contracts/contracts/AccountCodeStorage.sol
- system-contracts/contracts/BootloaderUtilities.sol
- system-contracts/contracts/ComplexUpgrader.sol
- system-contracts/contracts/Compressor.sol
- system-contracts/contracts/Constants.sol
- system-contracts/contracts/ContractDeployer.sol
- system-contracts/contracts/Create2Factory.sol
- system-contracts/contracts/DefaultAccount.sol
- system-contracts/contracts/EmptyContract.sol
- system-contracts/contracts/EventWriter.yul
- system-contracts/contracts/ImmutableSimulator.sol
- system-contracts/contracts/KnownCodesStorage.sol
- system-contracts/contracts/L1Messenger.sol
- system-contracts/contracts/L2BaseToken.sol
- system-contracts/contracts/MsgValueSimulator.sol
- system-contracts/contracts/NonceHolder.sol

- system-contracts/contracts/PubdataChunkPublisher.sol
- system-contracts/contracts/SystemContext.sol
- system-contracts/contracts/precompiles/CodeOracle.yul
- system-contracts/contracts/precompiles/EcAdd.yul
- system-contracts/contracts/precompiles/EcMul.yul
- system-contracts/contracts/precompiles/Ecrecover.yul
- system-contracts/contracts/precompiles/Keccak256.yul
- system-contracts/contracts/precompiles/P256Verify.yul
- system-contracts/contracts/precompiles/SHA256.yul
- system-contracts/contracts/interfaces/IAccount.sol
- system-contracts/contracts/interfaces/IAccountCodeStorage.sol
- system-contracts/contracts/interfaces/IBaseToken.sol
- system-contracts/contracts/interfaces/IBootloaderUtilities.sol
- system-contracts/contracts/interfaces/IComplexUpgrader.sol
- system-contracts/contracts/interfaces/IContractDeployer.sol
- system-contracts/contracts/interfaces/ICompressor.sol
- system-contracts/contracts/interfaces/IImmutableSimulator.sol
- system-contracts/contracts/interfaces/IKnownCodesStorage.sol
- system-contracts/contracts/interfaces/IL1Messenger.sol
- system-contracts/contracts/interfaces/IL2StandardToken.sol
- system-contracts/contracts/interfaces/IMailbox.sol
- system-contracts/contracts/interfaces/INonceHolder.sol
- system-contracts/contracts/interfaces/IPaymaster.sol
- system-contracts/contracts/interfaces/IPaymasterFlow.sol
- system-contracts/contracts/interfaces/IPubdataChunkPublisher.sol
- system-contracts/contracts/interfaces/ISystemContext.sol
- system-contracts/contracts/interfaces/ISystemContextDeprecated.sol
- system-contracts/contracts/interfaces/ISystemContract.sol
- system-contracts/contracts/libraries/EfficientCall.sol
- system-contracts/contracts/libraries/RLPEncoder.sol
- system-contracts/contracts/libraries/SystemContractHelper.sol
- system-contracts/contracts/libraries/SystemContractsCaller.sol
- system-contracts/contracts/libraries/UnsafeBytesCalldata.sol
- system-contracts/contracts/libraries/TransactionHelper.sol
- system-contracts/contracts/libraries/Utils.sol

Repository details

- **Repository URL:** https://github.com/matter-labs/era-contracts
- **Commit hash:** 29f9ff4bbe12dc133c852f81acd70e2b4139d6b2

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

Trust has established a dominating presence in the smart contract security ecosystem since 2022. He is a resident on the Immunefi, Sherlock and C4 leaderboards and is now focused in auditing and managing audit teams under Trust Security. When taking time off auditing & bug hunting, he enjoys assessing bounty contests in C4 as a Supreme Court judge.

HollaDieWaldfee is a renowned security expert with a track record of multiple first places in competitive audits. He is a Lead Senior Watson at Sherlock and Lead Auditor for Trust Security and Renascence Labs.

Bernd is a blockchain and smart contract security researcher that has made the transition from a successful full-stack web developer career. His ability to quickly grasp new concepts and technologies and his attention to detail have helped him become a top auditor in the blockchain space. Having conducted 50+ audits, Bernd has identified numerous vulnerabilities across a wide range of DeFi protocols, wallets, bridges, and VMs. He currently splits his time between audit competitions, private audits and bug bounty hunting.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

A broad set of manual review methodologies has been utilized in order to assess the security of ZKsync Era.

- Line-by-line review of the contracts, external documentation and integrations
- Threat modelling, described in more detail in the next section
- Differential fuzzing of certain contracts
- Customized testing
- Collective brainstorming sessions

Threat Modelling & Analysis

One of the methodologies applied during the security assessment was defining key security characteristics of each module and contract, and then challenging any conditions or assumptions that support those properties. For transparency and for future maintenance of ZKsync Era, we provide part of our exploration kit below.

L1 Bridging logic

- Can a user finalize an invalid withdrawal?
- Can a failed deposit be repeatedly claimed on L1?
- Can a deposit be treated as failed on L1 but succeed?
- Can a withdrawal be processed on both shared and legacy bridges?
- Can funds be locked in the bridges?
- Can all funds be transferred from legacy to shared bridge during upgrading?
- Can a user withdraw another user's valid withdrawal?
- Can a failed deposit always be claimed?
- Could an edge case during the upgrade from legacy bridge allow double spending, or prevent claiming or withdrawing?
- Does the legacy bridge integrate correctly with the shared bridge after the upgrade?
- Is a valid withdrawal always provable?
- Is there incorrect coupling between base token and Ether?
- Are the correct tokens deposited or withdrawn?
- Can a deposit from a second bridge be forged?

Diamond Admin

- Can any desired configuration changes be made?
- Are all sensitive functions permissioned?
- Can users other than the STM owner cause irrevocable damage?
- Is there any combination of fields that should be set atomically but aren't?
- Are there race conditions between different roles, e.g., where the chain admin can take advantage of an inconsistent state during upgrades?

Executor

- Can an unverified batch be executed?
- Can an uncommitted batch be verified?
- Can an L1 transaction be censored while maintaining liveness?
- Can the block & batching timing restrictions be bypassed?
- Can an executed batch ever be reverted?
- Can a batch with invalid logs pass sanitization?
- Can the priority TX queue become out of sync?
- Are L2 system logs correctly processed?

- Is the batch commitment satisfactory in preventing any post-commit changes?
- Can the verification procedure succeed despite the ZK verifier check failing?
- Are all values in the batch properly validated?
- Are there trust assumptions with regards to the data that validators provide?
- Can the requirements for data availability be bypassed?
- Can wrong upgrade transactions be executed?
- Can upgrade transactions be bypassed and not executed?

Mailbox

- Can multiple transactions requests result in the same transaction hash?
- Can a user provide less value than minted on L2?
- Are transactions correctly validated or is it possible to pass unsafe parameters?
- Can a forged L2 log or message be successfully proven?
- Can a wrong L2 transaction status be proven?
- Is the L2 gas price correctly derived?
- Can there be situations in which the operator incurs a loss by executing L1 -> L2 transactions?
- Can the transaction sender be set to an address in the L2 kernel space?
- Can the chain admin cause irrevocable damage by setting parameters?
- Does the Mailbox work with different base tokens (different decimals and/or valuation than ETH)?

State Transition Manager

- Can an empty entry in the hyperchain mapping be consumed unsafely?
- Can the chain admin abuse version upgrades?
- Can the action of sunsetting a protocol version have unintended consequences?
- Can a user other than the STM owner cause unrecoverable damage?
- Is the creation of a new chain missing any initialization?

Validator Timelock

- Can the validator execution delay be bypassed?
- Could administrators or validators of one chain affect another chain?

Upgrading Logic

- Could an upgrade or batch commitment to an upgrade be executed twice?
- Does the usage of protocol minor version as the upgrade nonce have unintended consequences?
- Can the standard upgrade contract block a benign upgrade from executing?
- Is there a scenario where a necessary upgrade could not be performed?
- Could an upgrade be skipped?
- Can L1 and L2 side get out of sync when an upgrade makes changes to both sides?

L2 Bridging Logic

- Can a withdrawal be initiated without burning the same amount of tokens?
- Do finalized deposits map correctly to the corresponding L1 deposit transactions?

Precompiles

- Do the implementations adhere to specifications when those are available?
- Are there any calculations that can result in overflows or underflows which are not taken into account?
- Are invalid operations guaranteed to revert?
- Are enough bytes read from the underlying precompile verbatim output?
- Are the low-level verbatim instructions accessed correctly?
- Can there be edge cases in the EC precompiles or certain values that are not handled correctly?

Bootloader

- Can a Layer 1 TX be submitted which would revert the bootloader and block the execution queue?
- Could transactions be crafted to be processed differently in a block and in a batch, causing a re-org?
- Could the operator be undercompensated for gas used in execution of TXs, either by underestimating costs, or through bypassing of user payment?
- Is non-scratch memory ever overwritten?
- Can the operator overprice a transaction in a way that is not considered by the user-facing documentation?
- Are all operator-provided byte segments non-overlapping?
- Can the result of a TX execution (success / failure) be incorrect?
- Could a system context upgrade have unintended side effects?
- Could a transaction refund ever exceed the transaction fee?
- Are all segments of the operator-supplied memory validated or explicitly trusted?
- Could transactions be executed with lower gas limit than specified?
- Can an attacker trigger unaccounted pubdata emission?
- Can a transaction determine the environment (simulation or execution) in which it is executed to change its behavior?
- Is it possible to forge the sender of a transaction?
- Can virtualization of msg.value of a TX leak forward to another?
- Can the operator consider only a substring of the transaction bytes?
- Can the operator execute a transaction that a user has not signed or otherwise execute a transaction with tampered data?
- Could the bootloader confuse a zero return-value from a near-call as a valid value instead of a panic, or vice-versa?
- Can a bytecode be marked as known without it being published?

- Does the bootloader provide sufficient safety checks to protect against malicious paymaster and account implementations?
- Can transactions make the operator perform computations that are not paid for?
- Is unused gas always refunded and is the refunded amount as expected?
- Can the separation of execution gas and pubdata cost break the gas accounting?
- Are there low-level operations like an overflowing or underflowing addition, subtraction or memory access that can it be abused by the operator or users?

L2 Base Token

- Can a token transfer revert in an unexpected way?
- Can a withdrawal to L1 be treated as invalid?
- Could the token contract be insolvent?

System Context

- Can block or batch creation contradict the documented timing requirements?
- Does block hash derivation sufficiently protect against re-org attacks?
- Are global transaction parameters (baseFee, origin, gasPrice) always consistent?
- Is it possible that the virtual blocks upgrade, after which virtual blocks need to catch up to L2 blocks, ends up in an inconsistent state?

Account Abstraction

- Can a transaction be verified by the default account without the owner's intention?
- Can transactions be executed without sufficient account balance?
- Could the paymaster not be compensated for a TX?
- Can the paymaster receive payment for a transaction without fairly attempting to execute it?
- Can the variable length transaction encoding cause a TX hash to be ambiguous?

L2 Deployment

- Is it possible for contract bytecode data to not be available on L1?
- Could a contract deployment override an existing contract?
- Could an in-construction state of a deployed contract be abused?
- Could the ZKsync address derivation scheme have collisions with EVM address derivation?
- Can the ability to update account versions and nonce ordering be abused deliberately?

Compression and publishing of L2 to L1 data

- Can incorrectly compressed bytecodes / state diffs be accepted by the Compressor?
- Could an inefficient bytecode be accepted?

- In which ways is the operator able to make changes to compressed bytecode and compressed state diffs without causing a revert?

L1Messenger

- Are all pubdata costs and gas costs of messages and logs accounted and charged for?
- Can the Merkle tree capacity be exceeded?
- Can the operator publish invalid pubdata without detection either on L2 or on L1?

NonceHolder

- Can the flexibility of nonces be abused (e.g., creating transactions that revert the bootloader)?
- Can the lack of nonces for L1 -> L2 transaction senders cause problems with regards to transaction hash uniqueness or consistency with EVM?

System Contracts (high-level)

- Are all external functions access-protected and don't cross security boundaries?
- Are the system contract addresses consistent in the bootloader and the Solidity contracts?
- Do the system contracts expose an interface to users which can be easily misused (footguns)?
- Can the libraries be used in an unsafe way (e.g. *UnsafeBytesCalldata* library) and how is this protected against upstream?
- Do the system contracts match the behavior of Ethereum? If not, are differences already known?

Cross-Module

- Are transactions sufficiently sandboxed within the execution pipeline?
- Can an L1 to L2 transaction be initiated where the gas spending will be higher than the provided gas?
- Do gas costs accounted for on L1 match the costs on L2?
- Can data be serialized from L1 Mailbox so that the bootloader would revert in deserialization, sanitization, or execution?
- Can all actions on L2 be performed via L1 -> L2 transactions?
- Could a valid batch be submitted with payloads (e.g. log hashes) that would not be accepted by the L1 Executor?
- Are there any unexpected interactions between the nonce system and priority TXs?
- Are there any differences between the simulations done before publishing a block, and the sealing of a batch, that would make it possible for a batch to not be sealed, causing a re-org?

Standardization Adherence

- Does an EVM instruction or Solidity contract developed for EVM behave differently in the zkEVM without it being documented?
- Is an established EIP stated and used but a SHOULD or MUST clause is not respected?
- Does the default AA implementation behave the same as if a contract is an EOA?

zkEVM ABI

- Are fields packed correctly for each ABI call?
- Are all values masked properly?
- Is there an instance of a call using the wrong forwarding mode?
- Does the code page logic interact safely with all possible upgrading flows?
- Are the special CALL instruction semantics that are unique to zkEVM well behaved for different combinations of flags and does it match the documentation?
- Are dangerous low-level instructions like mimic calls properly restricted to kernel space addresses?

Configuration and constants

- Are the variables and constants correct?
- Are there outdated constants or configuration variables for which the impact can be escalated beyond minor gas differences?

Qualitative analysis

Metric	Rating	Comments
Code complexity	**Good**	A significant attempt was made to reduce complexity in most areas of the code, reducing attack risks. Complexity mainly arises from the differences of the EraVM compared to the EVM.
Documentation	**Good**	Code is well documented with inline and external documentation, but the documentation is partially outdated.
Best practices	**Good**	Project adheres to industry standards.
Centralization risks	**Moderate**	The protocol owner is fully trusted. However, it is a decentralized Governance with further plans for decentralization. Governance contracts have been out of scope for this audit.

Findings

Medium severity findings

TRST-M-1 Missing pubdata accounting in L1Messenger.sendL2ToL1Log() allows to force re-org of L2 blocks

- **Category:** Re-org attacks
- **Source:** L1Messenger.sol
- **Status:** Open

Description

After processing each L1 -> L2 transaction, the *Bootloader* calls *L1Messenger.sendL2ToL1Log()* such that the transaction's success or failure can be proved on L1.

Sending one log to L1 costs 88 pubdata bytes, and there is limited space of pubdata per batch. For example, when constructing a batch for which the pubdata should be published in L1 calldata, there can be no more than **126,976 bytes** of pubdata, which is the amount of pubdata that fits into one blob.

Hence, it is necessary that the batch is closed before the available pubdata is consumed. If the batch is not closed, and the overconsumption of pubdata is only recognized at the time when the operator tries to finalize the batch, the operator must roll back transactions to decrease the pubdata consumption of the batch. This is a problem because the transactions have been published by the operator and making changes results in a re-org of L2 blocks.

Now, to understand the attack, it is necessary to understand that the operator keeps track of seal criteria that are checked after the execution of each transaction. The data about how much pubdata has been consumed in the seal criteria is connected to the pubdata that is accounted for in *L1Messenger.sendL2ToL1Log()*. The code with the seal criteria has been out of scope of the audit, but the mechanism has been discussed and verified with the client.

```
        // We need to charge cost of hashing, as it will be used in
`publishPubdataAndClearState`:
        // - keccakGasCost(L2_TO_L1_LOG_SERIALIZE_SIZE) and keccakGasCost(64) when
reconstructing L2ToL1Log
        // - at most 1 time keccakGasCost(64) when building the Merkle tree (as merkle
tree can contain
        // ~2*N nodes, where the first N nodes are leaves the hash of which is
calculated on the previous step).
        uint256 gasToPay = keccakGasCost(L2_TO_L1_LOG_SERIALIZE_SIZE) + 2 *
keccakGasCost(64);
        SystemContractHelper.burnGas(Utils.safeCastToU32(gasToPay), 0);
```

The second parameter in the call to *SystemContractHelper.burnGas()* is zero, and so no pubdata is accounted for. That does not cause misaccounting of user's pubdata spending, as it is only used for reporting of L1->L2 transactions, which already prepaid for this pubdata. However, it does consume 88 pubdata without it being accounted for in the seal criteria.

By purposefully creating L1 -> L2 transactions (they can have empty transaction data, and thus be minimally expensive), the operator can be tricked into giving out pubdata without sealing the batch. For example, the operator may seal the batch if there are more than **100kb** of

pubdata consumed with the maximum amount being **126,976 bytes** as mentioned before. By submitting **26,976 bytes / 88 bytes per transactions + 1 transaction = 307 transactions**, the pubdata can be increased beyond the limit of **126,976 bytes** as the operator attempts to fill the available pubdata with additional transactions which themselves consume another **100kb** of pubdata (assuming the internal pubdata counter was at zero). Now, since the pubdata limit has been exceeded, the operator must go back and remove transactions from the batch that it has already committed to, causing a re-org of L2 blocks.

To cause a more severe re-org, consider submitting 1000 L1 -> L2 transactions that the operator includes in a batch. The difference between the true and perceived pubdata that the batch has consumed is at this point **1,000 transactions * 88 bytes per transaction = 88,000 bytes**. The operator now continues to build the batch and adds as many transactions as needed to consume another **100kb** of pubdata. The re-org is now more severe since at least transactions worth **188,000 bytes – 126,976 bytes = 61,024 bytes** must be discarded, reaching further into the past.

Recommended mitigation

Solving the issue is straightforward, by accounting for the pubdata cost. The recommendation also includes the updated **gasToPay** variable, although this is not strictly necessary since only *Bootloader* can call *L1Messenger.sendL2ToL1Log()* and the gas cost of publishing the log must be included in the **L1_TX_INTRINSIC_L2_GAS** constant.

```
--- a/system-contracts/contracts/L1Messenger.sol
+++ b/system-contracts/contracts/L1Messenger.sol
@@ -87,7 +87,7 @@ contract L1Messenger is IL1Messenger, ISystemContract {
         // - at most 1 time keccakGasCost(64) when building the Merkle tree (as
merkle tree can contain
         // ~2*N nodes, where the first N nodes are leaves the hash of which is
calculated on the previous step).
-        uint256 gasToPay = keccakGasCost(L2_TO_L1_LOG_SERIALIZE_SIZE) + 2 *
keccakGasCost(64);
+        uint256 gasToPay = keccakGasCost(L2_TO_L1_LOG_SERIALIZE_SIZE) + 2 *
keccakGasCost(64) + COMPUTATIONAL_PRICE_FOR_PUBDATA * L2_TO_L1_LOG_SERIALIZE_SIZE;
-        SystemContractHelper.burnGas(Utils.safeCastToU32(gasToPay), 0);
+        SystemContractHelper.burnGas(Utils.safeCastToU32(gasToPay),
L2_TO_L1_LOG_SERIALIZE_SIZE);
     }

     /// @notice Internal function to send L2ToL1Log.
```

Team response

TBD

TRST-M-2 Gas price mechanism is unfair for users and provides undesired incentives

- **Category:** Economic issues
- **Source:** Mailbox.sol, bootloader.yul
- **Status:** Open

Description

The finding aims to describe two categories of problems with how L2 gas prices are calculated. The first one is about how L2 gas prices are calculated and committed to at one time, but the

cost for batch commitment and DA is paid at a later time. Secondly, L2 gas pricing is not manipulation resistant.

From the root causes, it is then possible to lay out the incentives that govern the behavior of different actors (sophisticated and less sophisticated) and to reason about the resulting behavior. This goes beyond straightforward manipulation, which is also possible, and shows how the gas pricing mechanism does not achieve the desired outcome. There are straightforward measures that can be taken to provide a fairer and more manipulation resistant gas pricing.

For L1 -> L2 transactions, the L2 gas price is determined in *Mailbox._deriveL2GasPrice()* at the time the transaction is submitted on L1. The formula takes **tx.gasprice** and additional chain parameters, most notably, since they change over time, **baseTokenGasPriceMultiplierNominator** and **baseTokenGasPriceMultiplierDenominator**, as inputs. If the base token on L2 is not ETH, **baseTokenGasPriceMultiplierNominator** and **baseTokenGasPriceMultiplierDenominator** must be continuously updated by the chain admin to track changes in the ETH to L2 base token exchange rate.

For L2 transactions, the operator sets the L2 gas price at the beginning of a batch. At this point, the L2 gas price is committed to and stays valid for the duration of the batch (around 30 minutes to 1 hour).

Manipulation of the L2 gas price is only possible for L1 -> L2 transactions, by manipulating the inputs to the L2 gas price formula. The base fee on L1 can be increased on purpose by filling up blocks. For an estimation of the cost, this article has been used. Accordingly, a 2x manipulation costs around 0.32 ETH for an initial gas price of 2 gwei.

The second source of problems is that the L2 gas price does not match the cost of committing the batch and making the pubdata available, since L2 gas price is determined earlier. Sophisticated users can thus determine if the current L2 gas price at which they can submit transactions is below the fair price or above the fair price. For example, for an L2 transaction, the operator may have committed to a base fee of 1 gwei almost an hour ago. Sophisticated users can see that L1 gas prices have increased, and the base fee in the next L2 batch will likely increase to 5 gwei. It is incentivized to quickly submit transactions in the current batch such as not to pay the higher base fee in the next batch. The operator will accept these transactions, because even though the transactions don't pay enough gas, it is likely still enough to generate a marginal profit. Also consider that by not accepting transactions into the batch, the operating costs per batch are not socialized across as many transactions, therefore the gas assumptions would be wrong and not enough revenue per batch would be collected.

For L1 -> L2 transactions, users are encouraged to submit transactions when L1 fees are unusually low, at which point they can be expected to return to more normal levels soon. It is fair to assume that the updates to **baseTokenGasPriceMultiplierNominator** and **baseTokenGasPriceMultiplierDenominator** by the chain admin can be delayed, either due to network congestion, or a slow update interval. This is another source for arbitrage opportunities. Furthermore, similar to the block stuffing attack that could manipulate the L2 gas price, an attacker could stuff blocks to front-run multiplier updates, thus increasing the period where it's inaccurate and possibly make it deviate substantially from the non-manipulated value.

It can thus be observed that L2 gas prices encourage strategic behavior where sophisticated actors are able to underpay for their transactions. As a consequence, L2 gas prices must be raised to keep the same average, which must be paid for by regular users. Overall, the L2 gas prices present an arbitrage opportunity where sophisticated actors can get part of their transaction fees paid for by regular users.

The ability of the operator to simply raise L2 gas prices for all users, and by the chain admin to adjust L1 gas parameters for all users, is precisely what exacerbates the problem. There is no risk that the chain generates a deficit, and so the arbitrage can continue in the long term.

Recommended mitigation

Different measures can be taken to prevent manipulation of gas prices for L1 -> L2 transactions and to make L2 gas prices more resistant to arbitrage for both L1 -> L2 transactions and L2 transactions.

On L1, **tx.gasprice** can be made resistant to manipulation by using a gas price oracle. Similarly, the exchange rate of the ETH and the L2 base token can be retrieved from a price oracle, instead of relying on constant updates from the chain admin.

Another idea for L1 -> L2 transactions is to add a buffer to the paid gas which accounts for the uncertainty until the batch is committed. The refund of the buffer can then be claimed on L1 depending on the L1 gas price and the DA cost at the time of batch commitment.

For L2 transactions, it is possible to update the base fee for each block, as opposed to once per batch. The more frequent updates can counter arbitrage opportunities.

Team response

TBD

TRST-M-3 Gas spent on pubdata is not subtracted in between near calls, leading to uncompensated operator calculations and potential DOS

- **Category:** Gas issues
- **Source:** bootloader.yul
- **Status:** Open

Description

The *bootloader* executes transactions with the amount of gas that has been requested and paid for by the transactions. Pubdata cost is charged separately from execution cost and must be taken into account by calculating the difference of the pubdata counter after and before the transaction execution and multiplying the result by **gasPerPubdata**. A transaction is executed over multiple near calls, and so each near call checks if the remaining gas is sufficient to pay for the overall pubdata that the transaction has consumed up to this point.

An example will make it clear. In the example, the next near call to execute is *ZKSYNC_NEAR_CALL_callPostOp()*. Assume that at this point, **gasLeft = 40 million** and **50,000 bytes** of pubdata have been published. Accordingly, with **gasPerPubdata = 800**, the available execution gas should be **40 million – 800 * 50,000 = 0**. But **gasLeft** is equal to **40 million** because the pubdata expenditure is only subtracted after all near calls have been performed.

Therefore, *ZKSYNC_NEAR_CALL_callPostOp()* is allowed to spend the 40 million gas. This will eventually cause a panic within *ZKSYNC_NEAR_CALL_callPostOp()* when *isNotEnoughGasForPubdata()* is called and it is discovered that there doesn't remain enough gas to pay for **50,000 bytes** of pubdata.

Even though by reverting the near call it is ensured that the overspent gas does not cause any state changes, the operator is forced to perform computations without receiving compensation for it. This is made worse for two reasons. Firstly, users can force these additional computations without any cost on their end. Their transactions get executed with all the gas that they have paid for and a panic in *ZKSYNC_NEAR_CALL_callPostOp()* does not affect them, the overall transaction still gets executed. And secondly, transactions that force the operator to perform additional computations in this way, can't get caught and sorted out in the load balancing that might be in front of the main operator node as the transactions as a whole are successfully executed. This can make a DOS attack more likely.

The same root cause, which is that two near calls are made after one another where the first one doesn't subtract the pubdata cost before passing the remaining gas to the next near call, exists throughout the bootloader and the pattern is applied consistently. As such, it is sufficient to provide one of them as an example.

Recommended mitigation

Since the root cause is the pattern in which near calls are performed, it is not possible to provide a straightforward and narrow mitigation, instead a refactor of the near calls must be considered.

The following pattern is the most straightforward to implement instead:

1. After a near call, the gas for the pubdata that the near call has consumed is subtracted from the available gas.
2. The **basePubdataSpent** counter needs to be updated after each near call to not charge for the same pubdata more than once.
3. Any gas that is spent on pubdata must be subtracted from **reservedGas** first. In other words, when charging for pubdata, **reservedGas** is used before execution gas is used. Only when **reservedGas** is zero should the pubdata cost be subtracted from the execution gas.

A downside of this approach is that pubdata costs in one near call can't be offset by pubdata refunds in a later near call. The available gas might only turn out to be sufficient after the pubdata costs have been refunded again. For users it is unclear that there is such isolation between near calls.

Team response

TBD

Low severity findings

TRST-L-1 L1SharedBridge._getERC20Getters() incorrectly encodes ETH name and ETH symbol

- **Category:** Encoding issues
- **Source:** L1SharedBridge.sol
- **Status:** Open

Description

L1SharedBridge._getERC20Getters() ABI encodes **name**, **symbol**, and **decimals** for each token bridged to L2. The function separates ETH and ERC20 tokens. For ETH, hardcoded values are used.

```
if (_token == ETH_TOKEN_ADDRESS) {
    bytes memory name = bytes("Ether");
    bytes memory symbol = bytes("ETH");
    bytes memory decimals = abi.encode(uint8(18));
    return abi.encode(name, symbol, decimals); // when depositing eth to a non-eth
based chain it is an ERC20
}
```

name and **symbol** are not ABI encoded which reverts execution when trying to ABI decode them in *L2StandardERC20.bridgeInitialize()*. The ABI encoded bytes are not the same as the bytes themselves.

L2StandardERC20 accounts for the case when decoding reverts and ignores the values. As a result, the *name()* and *symbol()* functions for ETH revert on L2 chains that don't use ETH as their native token.

Recommended mitigation

In *L1SharedBridge._getERC20Getters()*, it is necessary that **name** and **symbol** for ETH are ABI encoded.

```
        if (_token == ETH_TOKEN_ADDRESS) {
-           bytes memory name = bytes("Ether");
-           bytes memory symbol = bytes("ETH");
+           bytes memory name = abi.encode(bytes("Ether"));
+           bytes memory symbol = abi.encode(bytes("ETH"));
            bytes memory decimals = abi.encode(uint8(18));
            return abi.encode(name, symbol, decimals); // when depositing eth to a
non-eth based chain it is an ERC20
        }
```

Team response

TBD

TRST-L-2 TransactionFilterer.isTransactionAllowed() is called with inconsistent sender and refundRecipient parameters

- **Category:** Logical issues
- **Source:** Mailbox.sol

- **Status:** Open

Description

The purpose of *TransactionFilterer* is to filter L1 -> L2 transactions that are requested in *Mailbox*.

Mailbox._requestL2TransactionSender() calls *TransactionFilterer.isTransactionAllowed()* with different parameters than the requested transaction contains. The two affected parameters are **_request.sender** and **_request.refundRecipient**.

Both parameters are not necessarily the values with which the transaction is added to the queue.

After *isTransactionAllowed()* has been called, it is possible that the two parameters change.

```
request.refundRecipient =
AddressAliasHelper.actualRefundRecipient(request.refundRecipient, request.sender);
// Change the sender address if it is a smart contract to prevent address collision
between L1 and L2.
// Please note, currently zkSync address derivation is different from Ethereum one,
but it may be changed in the future.
// slither-disable-next-line tx-origin
if (request.sender != tx.origin) {
    request.sender = AddressAliasHelper.applyL1ToL2Alias(request.sender);
}
```

If *TransactionFilterer* does not apply the same aliasing conditioned on **request.sender**, **request.refundRecipient**, **tx.origin** and code size, transactions can be executed that should be restricted.

Recommended mitigation

Call *TransactionFilterer.isTransactionAllowed()* after all aliases have been applied. Also, since address aliasing is an error-prone behavior, make sure to document this at the *TransactionFilterer* level.

```
@@ -237,21 +237,6 @@ contract MailboxFacet is ZkSyncHyperchainBase, IMailbox {
     function _requestL2TransactionSender(
         BridgehubL2TransactionRequest memory _request
     ) internal nonReentrant returns (bytes32 canonicalTxHash) {
-        // Check that the transaction is allowed by the filterer (if the filterer is
set).
-        if (s.transactionFilterer != address(0)) {
-            require(
-                ITransactionFilterer(s.transactionFilterer).isTransactionAllowed({
-                    sender: _request.sender,
-                    contractL2: _request.contractL2,
-                    mintValue: _request.mintValue,
-                    l2Value: _request.l2Value,
-                    l2Calldata: _request.l2Calldata,
-                    refundRecipient: _request.refundRecipient
-                }),
-                "tf"
-            );
-        }
-
         // Enforcing that `_request.l2GasPerPubdataByteLimit` equals to a certain
constant number. This is needed
         // to ensure that users do not get used to using "exotic" numbers for
_request.l2GasPerPubdataByteLimit, e.g. 1-2, etc.
```

```
        // VERY IMPORTANT: nobody should rely on this constant to be fixed and every
contract should give their users the ability to provide the
@@ -284,6 +269,21 @@ contract MailboxFacet is ZkSyncHyperchainBase, IMailbox {
             request.sender = AddressAliasHelper.applyL1ToL2Alias(request.sender);
         }

+        // Check that the transaction is allowed by the filterer (if the filterer is
set).
+        if (s.transactionFilterer != address(0)) {
+            require(
+                ITransactionFilterer(s.transactionFilterer).isTransactionAllowed({
+                    sender: request.sender,
+                    contractL2: request.contractL2,
+                    mintValue: request.mintValue,
+                    l2Value: request.l2Value,
+                    l2Calldata: request.l2Calldata,
+                    refundRecipient: request.refundRecipient
+                }),
+                "tf"
+            );
+        }
+
         // populate missing fields
         _params.expirationTimestamp = uint64(block.timestamp + PRIORITY_EXPIRATION);
// Safe to cast
```

Team response

TBD

TRST-L-3 The isWithdrawalFinalized and depositAmount state variables in L1ERC20Bridge are inaccurate for legacy deposits after the v24 upgrade

- **Category:** Logical issues
- **Source:** L1ERC20Bridge.sol
- **Status:** Open

Description

The *L1ERC20Bridge* legacy bridge contract keeps track of finalized withdrawals in the **isWithdrawalFinalized** storage mapping, indicating that an L2 -> L1 withdrawal message was already processed on L1. Since the v24 upgrade on June 7 2024, *L1ERC20Bridge* proxies its requests to *L1SharedBridge.*

However, the legacy *L1ERC20Bridge.finalizeWithdrawal()* function does not set **isWithdrawalFinalized** to **true**. Instead, the withdrawal is processed by calling *L1SharedBridge.finalizeWithdrawalLegacyErc20Bridge(),* which prevents repeated withdrawal finalizations. Consequently, contracts that still integrate with the legacy *L1ERC20Bridge* cannot use **isWithdrawalFinalized** to reliably determine whether a withdrawal has been finalized, thus breaking the backward compatibility.

Similarly, the public **depositAmount** storage mapping in *L1ERC20Bridge*, which stores the initial deposit amount and which is cleared when using the legacy claim mechanism for a failed L2 deposit to prevent repeated claims, is not cleared by *L1SharedBridge.claimFailedDeposit().* However, since the visibility of **depositAmount** has been set to **internal** until the v24 upgrade,

integrating contracts have not been able to use this state variable before, and backward compatibility is maintained.

Recommended mitigation

To ensure third party contracts that integrate with the legacy *L1ERC20Bridge* contract can accurately determine if a withdrawal has been finalized, visibility of the **isWithdrawalFinalized** mapping should be changed from **public** to **internal** and a new getter function *isWithdrawalFinalized(uint256 _l2BatchNumber, uint256 _l2ToL1MessageNumber)* should be added to *L1ERC20Bridge* which checks both the **internal isWithdrawalFinalized** mapping and **isWithdrawalFinalized** in *L1SharedBridge*.

Additionally, it is recommended to change the visibility of the **depositAmount** storage mapping in *L1ERC20Bridge* to **internal** again to prevent integrating contracts from accessing potentially outdated values. But it should be noted that this will break contracts that rely on this variable since the v24 upgrade.

Team response

TBD

TRST-L-4 bootloader.upgradeSystemContextIfNeeded() overwrites future SystemContext upgrades

- **Category:** Logical issues
- **Source:** bootloader.yul
- **Status:** Open

Description

At the start of every new batch, the bootloader calls *upgradeSystemContextIfNeeded()*, which is only temporarily needed for a one-time upgrade of the *SystemContext* system contract. Since the upgrade has already been performed, this function is no longer needed and can be safely removed. However it is important to state that it *needs* to be removed.

In a scenario where *SystemContext* is upgraded another time, the next batch will reset the *SystemContext* code to the old implementation (it is a hardcoded bytecode in the bootloader), leading to inconsistent, and potentially dangerous, behavior.

Recommended mitigation

Remove the *upgradeSystemContextIfNeeded()* function.

Team response

TBD

TRST-L-5 registerAlreadyDeployedHyperchain() may overwrite an existing hyperchain due to not checking if the chain is already registered

- **Category:** Front-running issues

- **Source:** StateTransitionManager.sol
- **Status:** Open

Description

The *StateTransitionManager* owner can register an already deployed hyperchain with *registerAlreadyDeployedHyperchain()* by providing the **_chainId** and the address of the chain's state transition contract.

However, it is not checked whether the chain with the given **_chainId** is already registered. This opens up the possibility of a race condition where another transaction that creates a hyperchain with *createNewChain()* is executed first, followed by a transaction that executes *registerAlreadyDeployedHyperchain()*, resulting in the chain getting overwritten. This is especially relevant when creating new hyperchains is made permissionless in future versions.

Recommended mitigation

In *registerAlreadyDeployedHyperchain*(), it is recommended to check whether the **_chainId** is already registered with a hyperchain.

Team response

TBD

TRST-L-6 L1 -> L2 priority transactions cannot invalidate L2 account nonces

- **Category:** Access control issues
- **Source:** NonceHolder.sol
- **Status:** Open

Description

NonceHolder.increaseMinNonce() increases the minimum nonce for **msg.sender**, preventing transactions with a nonce lower than this minimum value from being executed on L2. However, this function has the **onlySystemCall** modifier, enforcing that it is only called via an explicit system call or that it is called from a system contract in the kernel space. This restriction prevents *increaseMinNonce()* from being called by an L1 -> L2 priority transaction via the L1 Mailbox. As a result, users cannot cancel already signed but not yet executed L2 transactions with a priority transaction and thus must use an L2 transaction instead, which might stay pending in the mempool without invalidating the nonces in time.

Notably, in the current version, the operator can censor L1 -> L2 priority transactions, which means that the priority transaction may not be included and executed, preventing users from invalidating nonces on L2. However, all transactions must be censored; it is not possible to do it selectively.

Once L1 -> L2 transactions are made fully censorship resistant, being unable to use L1 -> L2 transactions to invalidate L2 transactions introduces a crucial limitation for L1 -> L2 transactions.

Recommended mitigation

To enable L1 -> L2 priority transactions to overwrite and invalidate account nonces on L2, it is recommended to allow system calls to call *NonceHolder.increaseMinNonce()*. This could be implemented similarly to how calls to the *ContractDeployer* are selectively allowed in *shouldMsgValueMimicCallBeSystem()*.

Team response

TBD

TRST-L-7 Bootloader gas price constants might not be compatible with custom L2 base tokens

- **Category:** Configuration issues
- **Source:** bootloader.yul
- **Status:** Open

Description

The values returned by *MAX_ALLOWED_FAIR_PUBDATA_PRICE()* and *MAX_ALLOWED_FAIR_L2_GAS_PRICE()* in the bootloader are hardcoded to **1000000000000000** and **10000000000000**, respectively. However, given that L2s can configure custom base tokens for gas payments, these hardcoded values may not be universally appropriate. For example, if a base token has a high monetary value, it will lead to disproportionately high gas prices, unlike with a base token of low value.

Recommended mitigation

It is recommended to parameterize the values returned by *MAX_ALLOWED_FAIR_PUBDATA_PRICE()* and *MAX_ALLOWED_FAIR_L2_GAS_PRICE()* as part of the bootloader preprocessing stage. The fact that chains with different base tokens need different *bootloader* implementations, must also be taken into account in the L1 contracts that create and manage the chains.

Team response

TBD

TRST-L-8 Inconsistent AccountCodeStorage.getCodeHash() behavior for aliased accounts

- **Category:** Specification issues
- **Source:** AccountCodeStorage.sol
- **Status:** Open

Description

AccountCodeStorage.getCodeHash() returns **EMPTY_STRING_KECCAK** (i.e., the keccak hash of the empty string - **keccak("")**) when the account's code hash is zero, and the raw nonce is greater than zero, suggesting that the account on L2 is an EOA with at least one transaction.

However, this behavior is inconsistent for aliased accounts that only interact with the L2 via L1 -> L2 priority transactions. Specifically, the account's nonce (i.e., the deployment nonce) is

only incremented when deploying a contract on L2 and left unchanged for non-deployment transactions. Consequently, *getCodeHash()* differentiates based on whether the aliased account has deployed a contract and not on whether it has made a transaction in general.

This inconsistency does not cause any immediate security impact, but it could lead to unexpected behavior when *AccountCodeStorage.getCodeHash()* or the **extcodehash** opcode is used by external contracts.

Recommended mitigation

To maintain consistency between L2 EOAs and aliased accounts, it is recommended to mark the aliased account during the processing of L1 -> L2 priority transactions. This information should then be utilized in *AccountCodeStorage.getCodeHash()*. If it is valuable to discriminate between L1 contracts and EOAs, the L1 Mailbox could include this distinction in the priority transaction data, allowing it to be stored on L2 for future use.

It may also be reasonable to acknowledge this issue without mitigating it, as only *getCodeHash()* is affected, and there is no immediate security impact.

Team response

TBD

TRST-L-9 Operator can make L1 -> L2 transactions fail by overcharging for pubdata

- **Category:** Censorship issues
- **Source:** bootloader.yul
- **Status:** Open

Description

Transactions submitted via the L1 *Mailbox* have their gas price and gas per pubdata price determined on L1. The transaction is executed on L2 with these same parameters, without the operator being able to affect them, e.g., by increasing the gas price or refusing to execute the transaction with the given parameters. The operator is only able not to execute L1 transactions at all, but then all of them must be censored. Selective censorship is not possible.

The issue is that the operator can charge up to **65 bytes** of pubdata per changed storage slot. A non-malicious operator would compress the pubdata such that one storage slot costs significantly less than **65 bytes**, while a malicious operator might opt not to compress the pubdata and charge the maximum **65 bytes**.

By overcharging for pubdata in this way, considering the constant **800 gas per pubdata** and **72 million gas limit** (this is the default gas limit for L1 -> L2 transactions that new chains are initialized with; it may be changed by the chain admin), the operator can restrict a L1 transaction to **72 million gas / 800 gas per pubdata / 65 pubdata per storage slot = 1384 storage slots**. As a result, L1 transactions that write to more than around 1384 storage slots don't have the same censorship guarantee as L1 transactions that write to fewer storage slots.

Recommended mitigation

As a short-term solution, this limitation for the censorship resistance of L1 transactions should be better documented. In the long-term, it is recommended to make optimal pubdata compression mandatory, such that the operator is unable to overcharge for pubdata.

Team response

TBD

TRST-L-10 Admin of StateTransitionManager can set validators and achieve instant finality of batches

- **Category:** Privilege escalation
- **Source:** StateTransitionManager.sol
- **Status:** Open

Description

The admin in *StateTransitionManager* is allowed to call *setValidator()* and *setValidatorTimelock()*. It is set to a hot multisig which should not be able to cause an unrecoverable state, e.g., it should not be able to cause a loss of funds.

However, being able to call *setValidator()* or *setValidatorTimelock()* (which sets the validator that new chains are initialized with), allows the admin to achieve instant finality of batches.

Usually, validators use a timelock such that batches can't be executed immediately. However, the admin can set a validator that doesn't have a timelock and execute batches immediately. The admin can thus leverage an existing vulnerability as soon as it is discovered, front-running the owner before it can be fixed, or collude with the chain admin to execute an upgrade that puts the chain in an inconsistent state.

Recommended mitigation

A possible solution is to implement a delay that applies to admin (but not to owner) when calling *setValidator()* or *setValidatorTimelock()*.

Additionally, a function is needed for owner to revert a pending action made by admin and a function for admin to execute a pending action.

Team response

TBD

TRST-L-11 Enforcing pubdata and log constraints at the end of a batch can cause re-orgs

- **Category:** Logical issues
- **Source:** L1Messenger.sol, PubdataChunkPublisher.sol
- **Status:** Open

Description

At the end of a batch, the *bootloader* processes all the pubdata that must be published and calls *L1Messenger.publishPubdataAndClearState()*, which in turn calls *PubdataChunkPublisher.chunkAndPublishPubdata()*. In *L1Messenger*, it is checked that the number of logs does not exceed the capacity of the Merkle tree.

```
require(numberOfL2ToL1Logs <= L2_TO_L1_LOGS_MERKLE_TREE_LEAVES, "Too many L2->L1
logs");
```

And in *PubdataChunkPublisher*, it is checked that the pubdata does not exceed the capacity of the maximum of 6 blobs.

```
require(_pubdata.length <= BLOB_SIZE_BYTES * MAX_NUMBER_OF_BLOBS, "pubdata should fit
in 6 blobs");
```

It is unsafe to perform these checks only at the end of the batch since at the time the code will be executed, the operator has already committed to all the transactions within the batch, and so excluding transactions from the batch to meet the above requirements means to cause a re-org of the L2 chain.

Recommended mitigation

The re-org can currently not be exploited, since after each transaction it is checked in the seal criteria within the node that the batch can't consume too much pubdata, and by extension logs. Nonetheless, on-chain code should not have security backed by off-chain configuration. the practice of placing the checks only at the end of the *bootloader* execution is unsafe, and it is recommended to also perform the checks after every transaction in the *bootloader*. If the checks fail, the *bootloader* should be reverted. Consider also that when sequencer is decentralized, it cannot be trusted to perform the seal checks above.

Team response

TBD

TRST-L-12 bootloader performs calls with less gas than intended due to 63/64 rule

- **Category:** Gas issues
- **Source:** bootloader.yul
- **Status:** Open

Description

Bootloader keeps track of the gas that a transaction (both L1 -> L2 and L2 transactions) consumes by passing the gas that a transaction has available to near calls inside the bootloader. The near calls then continue to execute the transaction steps, e.g., execute the transaction and validate the transaction.

To illustrate the issue, suppose a L2 transaction has **100,000 gas** available, which is then passed to *ZKSYNC_NEAR_CALL_executeL2Tx()*. This near call then goes on to call *executeL2Tx()* -> *callAccountMethod()*. The call into the account is then made with a low level *call()*. This low-level *call()* reserves 1/64th of the available gas for the caller context, i.e., the bootloader, and the account receives only 63/64th of the available gas. Assume that once the low level *call()* is reached, there are still **80,000 gas** available from the initial **100,000 gas**. The account is then

called with **63/64 * 80,000 gas = 78,750 gas** instead of the full amount. In absolute terms, the difference can become more significant for large gas amounts. E.g., if the available gas is 80 million instead of 80 thousand, the reserved gas is over 1 million.

Recommended mitigation

The issue is structural, it originates from how the low-level calls are implemented. To address the finding, the *Bootloader* needs access to low-levels calls that don't reserve gas. In addition, there are other contracts involved in the preparation for a transaction (*MsgValueSimulator*, *L2BaseToken*, …) for which it is not expected to reserve gas, because they are part of the system, and a transaction should not be cut short due to them reserving gas.

It can be argued that by providing more gas, transactions are always able to execute, and the issue is not significant because the gas is refunded in the end. Additionally, while reserving 1/64th of the available gas can become significant in absolute terms, it remains less than 2% in relative terms. But there is a counterargument for both points. For one, there are issues TRST-PR-4 and TRST-PR-6 which show that refunds are not always performed. Secondly, by having the gas reserving behavior in multiple frames before the actual transaction execution is reached, the reserved gas compounds. The *bootloader* reserves gas when calling *MsgValueSimulator* which in turn reserves gas when mimicking the transaction sender and executing the transaction.

Team response

TBD

TRST-L-13 Function access privileges for chain admin are inconsistent

- **Category:** Privilege escalation
- **Source:** Admin.sol
- **Status:** Open

Description

The five functions that a chain admin is allowed to call are *Admin.changeFeeParams()*, *Admin.setTokenMultiplier()*, *Admin.setPubdataPricingMode()*, *Admin.setTransactionFilterer()* and *Admin.upgradeChainFromVersion()*. Conversely, the chain admin is not allowed to call *Admin.setPriorityTxMaxGasLimit()*. The severity of an issue that a malicious chain admin can cause by abusing the five functions that it is allowed to call, is equal to what can be caused by abusing *Admin.setPriorityTxMaxGasLimit()*.

This can be understood by first assessing the attack surface for *Admin.setPriorityTxMaxGasLimit()*, and then constructing the same impact from the five functions that the chain admin can call.

priorityTxMaxGasLimit, the value set in *Admin.setPriorityTxMaxGasLimit()*, is used in *TransactionValidator.validateL1ToL2Transaction()* to restrict the amount of gas that can be requested for a L1 -> L2 transaction.

```
function validateL1ToL2Transaction(
    L2CanonicalTransaction memory _transaction,
    bytes memory _encoded,
    uint256 _priorityTxMaxGasLimit,
```

```
    uint256 _priorityTxMaxPubdata
) internal pure {
    uint256 l2GasForTxBody = getTransactionBodyGasLimit(_transaction.gasLimit,
_encoded.length);


    // Ensuring that the transaction is provable
    require(l2GasForTxBody <= _priorityTxMaxGasLimit, "ui");
    // Ensuring that the transaction cannot output more pubdata than is processable
    require(l2GasForTxBody / _transaction.gasPerPubdataByteLimit <=
_priorityTxMaxPubdata, "uk");

    [...]
```

The scenario that this check prevents is that a transaction could be created that consumes so much gas that it can't be proved.

Note that the **FeeParams** struct contains the **priorityTxMaxPubdata** field, which can be set by the chain admin. And **priorityTxMaxPubdata** is used to a similar effect as **priorityTxMaxGasLimit**. **priorityTxMaxPubdata** ensures that a L1 -> L2 transaction can't publish more pubdata than can be handled within one batch.

Both **priorityTxMaxGasLimit** and **prorityTxMaxPubdata** can thus be used to create a L1 -> L2 transaction that exceeds the pubdata capacity of L2 and can't be proved.

Also, the remaining four functions can be used in creative ways to achieve the same impact. For example, **baseTokenGasPriceMultiplierNominator** can be set to **type(uint128).max** and **baseTokenGasPriceMultiplierDenominator** to **1**, such that the L2 gas price exceeds **type(uint128).max** and fails validation in the bootloader.

Recommended mitigation

To address the issue, it must be decided first whether it is an issue that the chain admin can allow the creation of L1 -> L2 transactions that can't be proved. After all, the impact is limited to a specific chain and the parameters can be reversed by the owner of *StateTransitionManager*. If a L1 -> L2 TX blocks the priority transaction queue, it can also be removed by performing an upgrade. The impact is not irreversible.

Thus, the first step is to clearly define the privileges of the chain admin. If it is determined to be acceptable to allow the chain admin the creation of unprovable transactions, *Admin.setPriorityTxMaxGasLimit()* should be made accessible to the chain admin. If the impact is not acceptable, the chain admin must be restricted from calling *Admin.changeFeeParams()*, specifically the **priorityTxMaxPubdata** field, and any other functions and parameters for which it can't be ruled out that the same or a similar impact can be created.

It is recommended to accept that the chain admin can set parameters such that transactions can't be proved. Essentially, it is the same impact as filtering all L1 -> L2 transactions, since L1 -> L2 transactions are blocked, but batches can still be created without them.

Team response

TBD

TRST-L-14 Gas constants are outdated leading to less revenue and failing L1 -> L2 transactions

- **Category:** Configuration issues
- **Source:** Config.sol
- **Status:** Open

Description

In *TransactionValidator.validateL1ToL2Transaction()*, it is checked that the L1 to L2 transaction has a sufficient gas limit to pay for the transaction overhead and the transaction preparation. In particular, the gas cost for the transaction preparation is calculated in *TransactionValidator.getMinimalPriorityTransactionGasLimit()*.

```
function getMinimalPriorityTransactionGasLimit(
    uint256 _encodingLength,
    uint256 _numberOfFactoryDependencies,
    uint256 _l2GasPricePerPubdata
) internal pure returns (uint256) {
    uint256 costForComputation;
    {
        // Adding the intrinsic cost for the transaction, i.e. auxiliary prices which
cannot be easily accounted for
        costForComputation = L1_TX_INTRINSIC_L2_GAS;


        // Taking into account the hashing costs that depend on the length of the
transaction
        // Note that L1_TX_DELTA_544_ENCODING_BYTES is the delta in the price for
every 544 bytes of
        // the transaction's encoding. It is taken as LCM between 136 and 32 (the
length for each keccak256 round
        // and the size of each new encoding word).
        costForComputation += Math.ceilDiv(_encodingLength *
L1_TX_DELTA_544_ENCODING_BYTES, 544);


        // Taking into the account the additional costs of providing new factory
dependencies
        costForComputation += _numberOfFactoryDependencies *
L1_TX_DELTA_FACTORY_DEPS_L2_GAS;


        // There is a minimal amount of computational L2 gas that the transaction
should cover
        costForComputation = Math.max(costForComputation, L1_TX_MIN_L2_GAS_BASE);
    }


    uint256 costForPubdata = 0;
    {
        // Adding the intrinsic cost for the transaction, i.e. auxiliary prices which
cannot be easily accounted for
        costForPubdata = L1_TX_INTRINSIC_PUBDATA * _l2GasPricePerPubdata;


        // Taking into the account the additional costs of providing new factory
dependencies
        costForPubdata += _numberOfFactoryDependencies *
L1_TX_DELTA_FACTORY_DEPS_PUBDATA * _l2GasPricePerPubdata;
    }


    return costForComputation + costForPubdata;
}
```

The goal of the function is to ensure the transaction is always supplied with enough gas that the transaction executes successfully until after the transaction preparation step in the bootloader.

To correctly account for the gas cost of *bootloader.l1TxPreparation()*, it must hold that:

$$\text{gasUsedOnPreparation} + \text{pending pubdata cost} <= \text{Math.ceilDiv}(\text{\_encodingLength} * \text{L1\_TX\_DELTA\_544\_ENCODING\_BYTES}, 544) + \text{\_numberOfFactoryDependencies} * \text{L1\_TX\_DELTA\_FACTORY\_DEPS\_L2\_GAS} + \text{\_numberOfFactoryDependencies} * \text{L1\_TX\_DELTA\_FACTORY\_DEPS\_PUBDATA} * \text{\_l2GasPricePerPubdata}$$

It can be observed that the gas constants in *Config.sol* underestimate the gas cost of TX preparation, causing two impacts. Firstly, the operator is not properly compensated if all TX gas is insufficient for the preparation step. Secondly, users can create L1 -> L2 transactions expecting the main logic to execute with a certain amount of gas, but then having it execute with less gas, causing unexpected reverts.

The conclusion that the gas constants are too low can be reached straightforwardly. Each published bytecode is accounted for with **L1_TX_DELTA_FACTORY_DEPS_L2_GAS = 2473 gas**, however the storage write in *KnownCodesStorage._markBytecodeAsPublished()* alone already consumes at least 5500 gas.

Recommended mitigation

Since the gas constants are outdated, the gas costs need to be benchmarked again, and the constants updated accordingly.

Team response

TBD

TRST-L-15 Gas price calculation in _deriveL2GasPrice() is not compatible with common base tokens with less than 18 decimals

- **Category:** Math issues
- **Source:** Mailbox.sol
- **Status:** Open

Description

For L1 -> L2 transactions, the gas price on L2 is calculated in *Mailbox._deriveL2GasPrice()*.

```
function _deriveL2GasPrice(uint256 _l1GasPrice, uint256 _gasPerPubdata) internal view
returns (uint256) {
    FeeParams memory feeParams = s.feeParams;
    require(s.baseTokenGasPriceMultiplierDenominator > 0, "Mailbox:
baseTokenGasPriceDenominator not set");
    uint256 l1GasPriceConverted = (_l1GasPrice *
s.baseTokenGasPriceMultiplierNominator) /
        s.baseTokenGasPriceMultiplierDenominator;
    uint256 pubdataPriceBaseToken;
    if (feeParams.pubdataPricingMode == PubdataPricingMode.Rollup) {
        // slither-disable-next-line divide-before-multiply
        pubdataPriceBaseToken = L1_GAS_PER_PUBDATA_BYTE * l1GasPriceConverted;
    }

    // slither-disable-next-line divide-before-multiply
```

```
    uint256 batchOverheadBaseToken = uint256(feeParams.batchOverheadL1Gas) *
l1GasPriceConverted;
    uint256 fullPubdataPriceBaseToken = pubdataPriceBaseToken +
        batchOverheadBaseToken /
        uint256(feeParams.maxPubdataPerBatch);


    uint256 l2GasPrice = feeParams.minimalL2GasPrice + batchOverheadBaseToken /
uint256(feeParams.maxL2GasPerBatch);
    uint256 minL2GasPriceBaseToken = (fullPubdataPriceBaseToken + _gasPerPubdata - 1)
/ _gasPerPubdata;


    return Math.max(l2GasPrice, minL2GasPriceBaseToken);
}
```

While the logic works correctly if the base token on L2 is ETH, it can be shown that it does not work for other common base tokens. It is important that the calculation supports different base tokens, as the L1 contracts of ZKsync allow for the creation of different chains with different base tokens.

Consider the calculation of **l1GasPriceConverted**, which is the gas price on L1 converted into L2 base tokens. It is the cost of 1 gas on L1 in wei of the L2 base token.

```
uint256 l1GasPriceConverted = (_l1GasPrice * s.baseTokenGasPriceMultiplierNominator) /
    s.baseTokenGasPriceMultiplierDenominator;
```

A typical **l1GasPrice** is **5 gwei**. Suppose the L2 base token is WBTC, then converting **5 gwei** of ETH into WBTC results in roughly **5 gwei ETH / 18 ETH per WBTC / 1e10** which rounds down to zero.

For a less extreme case, consider using USDC for the L2 base token. Converting **5 gwei** of ETH into USDC results in roughly **5 gwei ETH / (1/2400 ETH per USDC) / 1e12** which is equal to **12 wei USDC**. With such a number, the gas price becomes sensitive to any rounding that occurs later in the function.

Recommended mitigation

It Is clear that any L2 gas price formula has its limits and there is always a token which can't be used as a base token. The grounds on which the above is considered an issue, is that USDC and WBTC are well established tokens which should be safe to use as L2 base tokens.

The root cause is likely the fact that if a token on L1 is used as a base token on L2, the decimals of the token on L1 are the same as the decimals of the token on L2. This is enforced by the equality of the token amount minted on L2 and deposited on L1. It should be considered whether it is possible to configure L2 base tokens in such a way as to scale their decimals on L2. E.g., WBTC has 8 decimals on L1, but it could be scaled to 18 decimals on L2 such that it can be safely used as a base token, without breaking gas price calculations.

An additional recommendation, which is noteworthy but not sufficient to address the issue, is to round up in the following three instances to overestimate the gas price rather than to underestimate it.

```
@@ -163,7 +163,7 @@ contract MailboxFacet is ZkSyncHyperchainBase, IMailbox {
     function _deriveL2GasPrice(uint256 _l1GasPrice, uint256 _gasPerPubdata) internal
view returns (uint256) {
         FeeParams memory feeParams = s.feeParams;
```

```
        require(s.baseTokenGasPriceMultiplierDenominator > 0, "Mailbox:
baseTokenGasPriceDenominator not set");
-       uint256 l1GasPriceConverted = (_l1GasPrice *
s.baseTokenGasPriceMultiplierNominator) /
+       uint256 l1GasPriceConverted = (_l1GasPrice *
s.baseTokenGasPriceMultiplierNominator + s.baseTokenGasPriceMultiplierDenominator - 1)
/
            s.baseTokenGasPriceMultiplierDenominator;
        uint256 pubdataPriceBaseToken;
        if (feeParams.pubdataPricingMode == PubdataPricingMode.Rollup) {
@@ -174,10 +174,10 @@ contract MailboxFacet is ZkSyncHyperchainBase, IMailbox {
        // slither-disable-next-line divide-before-multiply
        uint256 batchOverheadBaseToken = uint256(feeParams.batchOverheadL1Gas) *
l1GasPriceConverted;
        uint256 fullPubdataPriceBaseToken = pubdataPriceBaseToken +
-           batchOverheadBaseToken /
+           (batchOverheadBaseToken + feeParams.maxPubdataPerBatch - 1) /
            uint256(feeParams.maxPubdataPerBatch);

-       uint256 l2GasPrice = feeParams.minimalL2GasPrice + batchOverheadBaseToken /
uint256(feeParams.maxL2GasPerBatch);
+       uint256 l2GasPrice = feeParams.minimalL2GasPrice + (batchOverheadBaseToken +
feeParams.maxL2GasPerBatch - 1) / uint256(feeParams.maxL2GasPerBatch);
        uint256 minL2GasPriceBaseToken = (fullPubdataPriceBaseToken + _gasPerPubdata
- 1) / _gasPerPubdata;

        return Math.max(l2GasPrice, minL2GasPriceBaseToken);
```

Team response

TBD

TRST-L-16 Mailbox._deriveL2GasPrice() overcharges users by not considering reduced blob fees

- **Category:** Logical issues
- **Source:** Mailbox.sol
- **Status:** Open

Description

In *Mailbox._deriveL2GasPrice()*, if pubdata pricing mode Is **Rollup**, i.e., either blobs or calldata, the price of one byte of pubdata is calculated as **L1_GAS_PER_PUBDATA_BYTE * l1GasPriceConverted**.

This calculation is only correct when the pubdata is published as calldata. However, the policy for the operator is to choose either blobs or calldata, depending on which option is cheaper. Therefore, if blob space is cheaper than calldata, which can be expected to be mostly true, users are overcharged and must pay as though the pubdata was published in the more expensive calldata.

Recommended mitigation

It is recommended to config the current main transport mode in the Mailbox and determine based on its cost.

Team response

TBD

TRST-L-17 Underutilized L2ToL1Log Merkle tree results in undercompensated hashing costs

- **Category:** Gas issues
- **Source:** L1Messenger.sol
- **Status:** Open

Description

L1Messenger.sendL2ToL1Log() and *L1Messenger.sendToL1()* charge for the later construction of the fixed-size Merkle tree. If the Merkle tree has **N** leaves, there are **N-1** additional internal nodes for which hashes must be calculated. *L1Messenger* accounts for these hashing costs of the internal nodes by charging each **L2ToL1Log** one hashing cost.

However, this assumes that the Merkle tree always utilizes all leaves, i.e., the batch contains the maximum number of logs. This is not guaranteed, resulting in not being compensated for the gas costs of the remaining "empty" leaves.

Recommended mitigation

To improve the accounting for hashing costs it is recommended to charge hashes based on the historic utilization of the Merkle tree. For example, if the Merkle tree is filled on average 50%, each **L2ToL1Log** should be charged two hashing costs instead of one hashing cost. The utilization rate should be kept up to date automatically to ensure overall gas spending is in check.

Team response

TBD

TRST-L-18 gasSpentOnExecution can exceed the provided gasForExecution resulting in slightly underpaid L1 -> L2 priority transaction gas fees

- **Category:** Logical issues
- **Source:** bootloader.yul
- **Status:** Open

Description

Processing a L1 -> L2 priority transaction with *processL1Tx()* internally calls *getExecuteL1TxAndNotifyResult()*, which is responsible for the actual execution, and provides **gasForExecution** as the amount of gas available for the execution. The return value **gasSpentOnExecution** is calculated based on the delta of *gas()* before and after the execution.

However, due to calling *notifyExecutionResult()* right before calculating **gasSpentOnExecution**, the gas consumed by this call is included in the return value, which results in a larger value than the initially provided **gasForExecution**. Subsequently, calculating **potentialRefund** in *processL1Tx()*, by using a saturated subtraction, leads to silently ignoring the uncompensated gas for the *notifyExecutionResult()* call. Consequently, the user slightly underpaid for the L1 -> L2 priority transaction execution on L2.

```
// It is assumed that `isNotEnoughGasForPubdata` ensured that the user did not publish
too much pubdata.
```

```
let potentialRefund := saturatingSub(
    safeAdd(reservedGas, gasForExecution, "safeadd: potentialRefund1"),
    safeAdd(gasSpentOnExecution, ergsSpentOnPubdata, "safeadd: potentialRefund2")
)
```

Notably, without using *saturatingSub()*, this gas accounting inaccuracy could possibly lead to a revert when subtracting the larger **gasSpentOnExecution** from the smaller **gasForExecution** value, manifesting as a DoS as the L1 -> L2 priority transaction cannot get executed and thus blocks the priority queue.

Similarly, it is observed that calculating **gasSpentByPostOp** in *refundCurrentL2Transaction()* includes the gas costs for *getErgsSpentForPubdata()*, resulting in a larger than expected value and thus an inaccurate **gasLeft**.

Recommended mitigation

To improve the accuracy of **gasSpentOnExecution**, it is recommended to calculate the value before calling *notifyExecutionResult()* so that the *gas()* snapshot does not include this call's consumed gas. Additionally, it must be ensured that the intrinsic L2 gas costs for a L1 -> L2 priority transaction (**L1_TX_INTRINSIC_L2_GAS**) include the gas costs for *notifyExecutionResult()*.

```
function getExecuteL1TxAndNotifyResult(
    txDataOffset,
    gasForExecution,
    basePubdataSpent,
    gasPerPubdata
) -> gasSpentOnExecution, success {
    debugLog("gasForExecution", gasForExecution)

    let callAbi := getNearCallABI(gasForExecution)
    debugLog("callAbi", callAbi)

    checkEnoughGas(gasForExecution)

    let gasBeforeExecution := gas()
    success := ZKSYNC_NEAR_CALL_executeL1Tx(
        callAbi,
        txDataOffset,
        basePubdataSpent,
        gasPerPubdataf
    )
+   gasSpentOnExecution := sub(gasBeforeExecution, gas())
    notifyExecutionResult(success)
-   gasSpentOnExecution := sub(gasBeforeExecution, gas())
}
```

Moreover, in *refundCurrentL2Transaction(),* the call to *getErgsSpentForPubdata()* should be moved to before the **gasBeforePostOp** gas snapshot.

```
  let nearCallAbi := getNearCallABI(gasLeft)

+ let spentOnPubdata := getErgsSpentForPubdata(
+   basePubdataSpent,
+   gasPerPubdata
+ )

  let gasBeforePostOp := gas()

- let spentOnPubdata := getErgsSpentForPubdata(
-   basePubdataSpent,
```

```
-    gasPerPubdata
- )
```

Team response

TBD

TRST-L-19 A L2 hyperchain can be retroactively registered with a different base token by the Bridgehub admin

- **Category:** Privilege escalation
- **Source:** StateTransitionManager.sol
- **Status:** Open

Description

The *StateTransitionManager (STM)* owner can register an already deployed hyperchain by using *registerAlreadyDeployedHyperchain()* and specifying the **_chainId** and the address of the chain's state transition diamond contract. This hyperchain L2 is then registered in the **hyperchainMap** and used to determine if a chain has been already registered when attempting to register it again.

To enable asset deposits to this L2 via *Bridgehub,* it is required that the *Bridgehub* owner or admin calls *createNewChain()* to register the base token and the STM for the given chain id. Internally, *StateTransitionManager.createNewChain()* is called and returns early when the chain has been registered already:

```
function createNewChain(
    uint256 _chainId,
    address _baseToken,
    address _sharedBridge,
    address _admin,
    bytes calldata _diamondCut
) external onlyBridgehub {
    if (getHyperchain(_chainId) != address(0)) {
        // Hyperchain already registered
        return;
    }
```

However, the *Bridgehub* admin can call *createNewChain()* with a different **_baseToken** or chain **_admin** than those initially used during the hyperchain diamond contract initialization. Consequently, an incorrect base token will be stored in **baseToken** which can be exploited by depositing a low-value token on L1, minting base token on L2, exchanging it for a non-base token on L2, and withdrawing it to L1 for profit.

Furthermore, if multiple active *StateTransitionManager* contracts exist, a L2 might be registered in *Bridgehub* with a different STM than the one used to register the hyperchain with *registerAlreadyDeployedHyperchain()*. This prevents registering another chain with the same ID due to the uniqueness check.

Overall, this presents a privilege escalation of the *Bridgehub* admin.

Recommended mitigation

To ensure hyperchains are correctly registered, it is recommended to change the access control of *StateTransitionManager.registerAlreadyDeployedChain()* to only be callable from *Bridgehub* by using the *onlyBridgehub* modifier and adding the accompanying caller function to *Bridgehub*.

This new function, only callable by the *Bridgehub* owner, should call *registerAlreadyDeployedHyperchain()* and register the chain similar to *Bridgehub.createNewChain()*, but without calling *StateTransitionManager.createNewChain()*. As a result, an already deployed hyperchain can be registered with a single function call.

Additionally, instead of having the owner provide **_baseToken**, **_admin**, and **_stateTransitionManager**, it is recommended to use the hyperchain's already initialized storage variables via the *Getters* facet to prevent any discrepancies.

Team response

TBD

TRST-L-20 Minor gas overspending in ZKSYNC_NEAR_CALL_executeL1Tx() when checking whether the remaining gas is sufficient to cover pubdata

- **Category:** Gas issues
- **Source:** bootloader.yul
- **Status:** Open

Description

L1 -> L2 priority transactions are executed in the *bootloader* via *ZKSYNC_NEAR_CALL_executeL1Tx()* with gas consumption limited by **gasForExecution**. At the end of the function, *isNotEnoughGasForPubdata()* is called to check whether sufficient gas is available to cover the pubdata.

However, by supplying the current *gas()* snapshot as the **computeGas** parameter without considering a small gas buffer for the remaining execution, it might lead to the situation where the current gas is barely enough for the pubdata but not enough to cover overhead costs such as calling *isNotEnoughGasForPubdata()*. This results in overspending a tiny amount of gas instead of reverting due to insufficient gas.

Recommended mitigation

To ensure remaining execution gas is reliably checked in *ZKSYNC_NEAR_CALL_executeL1Tx()*, it is recommended to apply a small constant gas buffer to the *gas()* snapshot that is supplied to *isNotEnoughGasForPubdata()*. The new pattern should be adopted for all instances of *isNotEnoughGasForPubdata()*.

Team response

TBD

TRST-L-21 L1Messenger does not account for gas that is consumed by iterating over all L2_TO_L1_LOGS_MERKLE_TREE_LEAVES

- **Category:** Gas issues
- **Source:** L1Messenger.sol
- **Status:** Open

Description

publishPubdataAndClearState() builds the Merkle tree with the fixed size of **L2_TO_L1_LOGS_MERKLE_TREE_LEAVES = 16_384** leaves. If the actual number of logs is less than **L2_TO_L1_LOGS_MERKLE_TREE_LEAVES**, the remaining Merkle tree leaves are filled up with **L2_L1_LOGS_TREE_DEFAULT_LEAF_HASH**. Overall, it is required to iterate over all 16,384 leaves, which consumes a significant amount of gas for which the operator is not compensated at the time when the log has been added via *sendToL1()*. Instead, only the costs for hashing are charged.

Recommended mitigation

To ensure that the gas costs for iterating over the Merkle tree leaves are compensated as much as possible, it is recommended to charge a fixed amount of gas on top of the hashing costs in *sendToL1()* and *sendL2ToL1Log()*. These additional costs should be calculated upfront based on historical batch logs utilization to account for batches with a low volume of logs that still require iterating over all Merkle tree leaves so that gas costs are fully covered.

Team response

TBD

TRST-L-22 Updating the verifier contract during an upgrade on L1 is error-prone

- **Category:** Logical issues
- **Source:** BaseZkSyncUpgrade.sol
- **Status:** Open

Description

BaseZkSyncUpgrade._setVerifier() changes the address of the verifier contract on L1 as part of an upgrade.

```
function _setVerifier(IVerifier _newVerifier) private {
    // An upgrade to the verifier must be done carefully to ensure there aren't
batches in the committed state
    // during the transition. If verifier is upgraded, it will immediately be used to
prove all committed batches.
    // Batches committed expecting the old verifier will fail. Ensure all committed
batches are finalized before the
    // verifier is upgraded.
    if (_newVerifier == IVerifier(address(0))) {
        return;
    }

    IVerifier oldVerifier = s.verifier;
    s.verifier = _newVerifier;
    emit NewVerifier(address(oldVerifier), address(_newVerifier));
}
```

As pointed out in the code comments, it is important to make sure that the verifier is only updated when there are no batches in the committed state, i.e., batches that require proving by the verifier contract and subsequent execution. Otherwise, batches that require the old verifier will fail and must be reverted.

However, this requirement is not checked in the code, instead, it must be manually done and preventive measures must be taken, such as pausing the *ExecutorFacet* of the hyperchain's diamond contract.

Additionally, after the verifier contract is updated, batches that are not compatible with the new verifier can still be committed as it is not possible to associate a batch with a specific verifier contract. Such batches must get reverted and committed again with support for the new verifier, which causes an additional delay imposed by *ValidatorTimelock* and resulting in a delayed batch finalization.

Recommended mitigation

To prevent batch finalization delays and to ensure a smooth transition when upgrading the verifier contract, it is recommended to add a versioning mechanism for verifier contracts and to specify the expected verifier contract version in a batch at commitment time. It should then be checked if the batch is compatible with the current verifier and otherwise the batch should be rejected.

Team response

TBD

TRST-L-23 eraLegacyBridgeLastDepositTxNumber is off-by-one resulting in the inability to claim the last deposit before the upgrade in case of a failed deposit

- **Category:** Off-by-one errors
- **Source:** L1SharedBridge.sol
- **Status:** Open

Description

Processing the claim of a failed L2 deposit in *L1SharedBridge* checks whether the deposit happened before the v24 upgrade. Deposits initiated prior to the upgrade must be claimed via the legacy *L1ERC20Bridge* contract. Specifically, *_isEraLegacyDeposit()* takes the deposit's **_l2BatchNumber** and **_l2TxNumberInBatch** and compares it to **eraLegacyBridgeLastDepositBatch** and **eraLegacyBridgeLastDepositTxNumber**, which are configured by governance.

```
function _isEraLegacyDeposit(
    uint256 _chainId,
    uint256 _l2BatchNumber,
    uint256 _l2TxNumberInBatch
) internal view returns (bool) {
    require(
        (_chainId != ERA_CHAIN_ID) || (eraLegacyBridgeLastDepositBatch != 0),
        "ShB: last deposit time not set for Era"
    );
    return
        (_chainId == ERA_CHAIN_ID) &&
        (_l2BatchNumber < eraLegacyBridgeLastDepositBatch ||
```

```
            (_l2TxNumberInBatch < eraLegacyBridgeLastDepositTxNumber &&
                _l2BatchNumber == eraLegacyBridgeLastDepositBatch));
}
```

The variable name **eraLegacyBridgeLastDepositTxNumber** implies that it holds the last deposit transaction number that is still considered to have happened in the legacy bridge.

However, the **_l2TxNumberInBatch < eraLegacyBridgeLastDepositTxNumber** check is exclusive, i.e., if **l2TxNumberInBatch == eraLegacyBridgeLastDepositTxNumber** the deposit is incorrectly determined to have happened in the new shared bridge. As a result, if this deposit failed, the claim is fully processed in *L1SharedBridge* and will fail as there is no such deposit tracked in the **depositHappened** mapping. This deposit cannot be claimed back.

Recommended mitigation

The upgrade of the legacy Era bridges has already occurred, and the variable has been set to the first TX number after the upgrade, such that there does not exist an issue even though the code is vulnerable. The below change should be made if the same code or a similar upgrade pattern is used in a different context in the future.

```
    return
        (_chainId == ERA_CHAIN_ID) &&
        (_l2BatchNumber < eraLegacyBridgeLastDepositBatch ||
-           (_l2TxNumberInBatch < eraLegacyBridgeLastDepositTxNumber &&
+           (_l2TxNumberInBatch <= eraLegacyBridgeLastDepositTxNumber &&
                _l2BatchNumber == eraLegacyBridgeLastDepositBatch));
```

Team response

TBD

Previously-known findings

The findings below have already been reported in previous audits, and are presented for user transparency and the value of the recommendations offered. It should not be assumed the list below is comprehensive of all known issues.

TRST-PR-1 Contracts on L1 cannot access their native balance on their aliased L2 address

- **Category:** Logical issues
- **Source:** Mailbox.sol
- **Status:** Open

Description

Submitting an L1 -> L2 priority transaction allows specifying a **refundRecipient** address, the recipient of the native token refund for a failed transaction on L2. If this address is an existing smart contract on L1 (i.e., an address with non-empty code), it will be modified by adding a predefined offset to create an alias.

Externally owned accounts (EOAs) have identical addresses on both L1 (Ethereum) and L2. Using the private key to sign a transaction on L2 allows EOAs to access and transfer native tokens.

On the other hand, smart contracts on L1 must use the *Mailbox* to initiate L1 -> L2 priority transactions. These transactions are executed on L2 with the **msg.sender** set to the aliased address. However, L1 -> L2 transactions are not able to access existing native funds on the L2 sender's address. This limitation is caused by verifying **request.mintValue >= baseCost + request.l2Value** in *Mailbox._requestL2Transaction()*, which ensures that the caller provides enough native tokens on L1 that will be subsequently minted on L2.

```
function _requestL2Transaction(WritePriorityOpParams memory _params) internal returns
(bytes32 canonicalTxHash) {
    BridgehubL2TransactionRequest memory request = _params.request;

    require(request.factoryDeps.length <= MAX_NEW_FACTORY_DEPS, "uj");
    _params.txId = s.priorityQueue.getTotalPriorityTxs();

    // Checking that the user provided enough ether to pay for the transaction.
    _params.l2GasPrice = _deriveL2GasPrice(tx.gasprice,
request.l2GasPerPubdataByteLimit);
    uint256 baseCost = _params.l2GasPrice * request.l2GasLimit;
    require(request.mintValue >= baseCost + request.l2Value, "mv"); // The `msg.value`
doesn't cover the transaction cost

    // [...]
```

Consequently, it is currently not possible for an L1 smart contract to access and transfer the native tokens on their aliased L2 address.

Recommended mitigation

It is recommended to provide a mechanism for smart contracts on L1 to access their native token balance on their aliased L2 address.

Team response

TBD

TRST-PR-2 Refund recipient aliasing can lead to loss of native funds

- **Category:** Logical issues
- **Source:** AddressAliasHelper.sol
- **Status:** Open

Description

It has been reported in previous audits that the calculation of **refundRecipient** for L1 -> L2 transactions with the logic implemented in *AddressAliasHelper.actualRefundRecipient()* allows for edge cases where the minted native tokens on L2 cannot be accessed.

Fundamentally, it is not necessary to apply an alias to **refundRecipient** at all. Such aliasing is only needed for the **sender** of the transaction to prevent it from impersonating accounts on L2. Since, by design, native tokens can be sent to any address, it is not necessary to apply aliases.

In TRST-PR-1, it has been shown that aliasing the **refundRecipient**, together with the inability of a contract on L1 to access the native funds of its alias on L2, can lead to a loss of funds.

Recommended mitigation

Due to the possible loss of funds, it is recommended to reconsider if aliasing **refundRecipient** is needed and has the desired effect.

A possible mitigation, although it might not be possible to implement it due to being a breaking change, could be to not alias **refundRecipient**. This gives anyone executing an L1 -> L2 transaction full control over the refunded funds, and they can be directed to an L2 address that can access them. Any mitigation must be in sync with the mitigation for TRST-PR-1 to ensure refunded funds on L2 can be accessed by a user using the protocol as intended.

Team response

TBD

TRST-PR-3: Operator can burn gas of L2 transactions with malformed bytecode compression

- **Category:** Centralization issues
- **Source:** Compressor.sol
- **Status:** Open

Description

It is known that the operator is trusted to provide the bytecodes that are published by L2 transactions in compressed form, such as not to overcharge the users.

However, beyond this trust assumption, there are security measures in place to restrict the operator. For example, the operator is not allowed to charge for a bytecode twice.

```
if eq(bytecodeHash, currentExpectedBytecodeHash) {
    // Here we are making sure that the bytecode is indeed not yet know and needs to
be published,
    // preventing users from being overcharged by the operator.
    let marker := getCodeMarker(bytecodeHash)


    if marker {
        assertionError("invalid republish")
    }


    dataInfoPtr := sendCompressedBytecode(dataInfoPtr, bytecodeHash)
}
```

The problem is that a malformed bytecode compression leads to a panic. Due to the panic, all remaining gas of the L2 transaction is burned and it is marked as failed, and the user is charged for it.

Recommended mitigation

It is not possible to replace the panic with an **assertionError()** since this would mean that a L2 TX that runs out of gas can revert the bootloader, which must not be possible.

A more elaborate solution is for the *Compressor.publishCompressedBytecode()* function to return a **flag** indicating whether the compression was faulty. An unsuccessful *call()* (out of gas) can then be handled with a panic, while **flag=false** can trigger an **assertionError()**. All ways in which *Compressor.publishCompressedBytecode()* can revert due to a faulty compression must return **flag=false** instead of reverting.

Team response

TBD

TRST-PR-4 Unused gas is not refunded for failed L1 -> L2 priority transactions

- **Category:** Logical issues
- **Source:** bootloader.yul
- **Status:** Open

Description

ZKSYNC_NEAR_CALL_executeL1Tx() in the bootloader executes a L1 -> L2 priority transaction on L2. This function is called as part of a near call, and the available gas is restricted to the execution gas limit configured and paid for by the user. If the execution fails, i.e., **success == 0**, the near call is reverted by calling *nearCallPanic()*. However, this exhausts all the remaining execution gas and results in no gas being refunded to the user in *processL1Tx()*. This behavior

is inconsistent with regular L2 transactions where unused execution gas is refunded to the user in case of an error.

Recommended mitigation

During discussion with the client, it was revealed that it could be possible to modify the inner workings of *nearCallPanic()* as desired. The goal is to revert the current frame's state changes without exhausting the remaining gas. Additionally, it is important to ensure that the error does not bubble up, causing the entire bootloader to revert. Such a possible change could be using *verbatim("throw")*.

Team response

TBD

TRST-PR-5 Broken DefaultAccount equivalence with EOAs

- **Category:** Logical issues
- **Source:** DefaultAccount.sol
- **Status:** Open

Description

DefaultAccount aims to mimic the behavior of an EOA with regards to accepting any calldata without reverting when being externally called. This is supposed to be achieved by using the *ignoreNonBootloader()* and *ignoreInDelegateCall()* modifiers, resulting in the contract being indistinguishable from an EOA.

```
/**
  * @dev Simulate the behavior of the EOA if the caller is not the bootloader.
  * Essentially, for all non-bootloader callers halt the execution with empty return
data.
  * If all functions will use this modifier AND the contract will implement an empty
payable fallback()
  * then the contract will be indistinguishable from the EOA when called.
  */
```

However, if a function with such a modifier is called with incorrectly ABI encoded calldata, it reverts, unlike an EOA on Ethereum, which accepts any calldata without reverting. This happens because the ABI decoding fails before the modifier is triggered. As a result, EOA equivalence is not fully achieved.

It is worth pointing out that in the context of an EOA, there is no such thing as incorrect ABI encoded calldata. For any calldata, the call is always successful and returns empty data.

Recommended mitigation

To ensure this behavior is known to developers, it is recommended to highlight the deviation from EOA behavior in the documentation.

Team response

TBD

TRST-PR-6 Paymaster receives refund even when postTransaction() fails or is not executed

- **Category:** Logical issues
- **Source:** bootloader.yul
- **Status:** Open

Description

IPaymaster.postTransaction() is a function that is called after the L2 TX has been executed, if the L2 TX has set a paymaster. In addition to having set a paymaster in the first place, **gasLeft** must be greater than zero.

The purpose of *postTransaction()* is to provide a refund to the sender of the L2 transaction, since the refund of the remaining base tokens is made to the paymaster. Importantly, the refund of base tokens to the paymaster is made regardless of whether *postTransaction()* is called and whether the call succeeds.

Therefore, if **gasLeft = 0** or when *postTransaction()* fails, even though in both cases there can still be gas available in **reservedGas**, the sender of the L2 transaction does not receive a refund.

Recommended mitigation

reservedGas is an amount of gas that, by definition, should not be used to pay for anything other than pubdata. Therefore, allowing *postTransaction()* to execute by consuming **reservedGas** is not possible.

One option is to refund the base tokens to the sender of the L2 transaction in the case when *postTransaction()* is not successful or is not called, but this assumes the sender of the L2 transaction can deal with the base tokens, and it may not necessarily be the same address that the paymaster has pulled funds from in *validateAndPayForPaymasterTransaction()* at the start of the transaction.

At the very least, the fact that **reservedGas** may not be refunded correctly, should be documented in *IPaymaster*.

Team response

TBD

Additional recommendations

TRST-R-1 Hardcoded name(), symbol() and decimals() values for L2BaseToken as well as hardcoded 18 decimals for L2WrappedBaseToken should be set dynamically

The *L2BaseToken*, which represents the native base token on L2 (e.g., ETH), returns the hardcoded values "Ether" and "ETH" from *name()* and *symbol()*, respectively, as well as 18 from *decimals()*. Consequently, hyperchains with a base token different from ETH or using a different number of decimals cannot have *L2BaseToken* return the correct values. Moreover, *L2WrappedBaseToken* does not override *ERC20Upgradeable.decimals()* and thus uses by default 18 decimals.

It is recommended to add a *constructor*() to *L2BaseToken* to set those values. Additionally, *ERC20Upgradeable.decimals()* in *L2WrappedBaseToken* should be overridden to enable customization of the decimals.

TRST-R-2 Consistently retrieve the Bootloader's native token balance using the same method

The *bootloader* retrieves, in multiple instances, its native token balance in two distinct ways:

1. *balance(BOOTLOADER_FORMAL_ADDR())*
2. *selfbalance()*

Internally, *selfbalance()* is implemented using the BALANCE opcode with ADDRESS as its argument. Thus, it is indifferent to using *balance(BOOTLOADER_FORMAL_ADDR())*. For consistency, it is recommended to choose one of the two methods and to stick to it throughout the *bootloader* code.

TRST-R-3 New hyperchains are possibly deployed with old initialCutHash but with new protocolVersion

Creating a new hyperchain with *StateTransitionManager.createNewChain()* requires providing the diamond cut data that initializes the chain's diamond proxy. The keccak256 hash of this data must match the stored **initialCutHash**. Additionally, the current **protocolVersion** is supplied to the diamond initialization. However, if the protocol is upgraded with *setNewVersionUpgrade()*, while **initialCutHash** remains unchanged, a new chain will be created with a mismatching diamond cut and protocol version. As a result, a subsequent upgrade of the chain's diamond proxy is not possible as the diamond's **protocolVersion** is already set to the latest version causing the check in *BaseZkSyncUpgrade. setNewProtocolVersion()* to revert. It is recommended to remove *setChainCreationParams()* and to supply the chain creation parameters to *setNewVersionUpgrade()* to ensure that the protocol version and the diamond cut data match.

TRST-R-4 Add documentation that the ETH depositor in Bridgehub.requestL2TransactionTwoBridges() must accept ETH transfers to claim failed L2 deposits

Non-base tokens - ERC-20 tokens and ETH - are deposited to L2 with *Bridgehub .requestL2TransactionTwoBridges()*. Failed L2 deposits can be reclaimed with *L1SharedBridge.claimFailedDeposit()* by providing a valid Merkle proof. If ETH is claimed, the receiver of the funds, **_depositSender**, which is the original depositor, must be able to receive ETH, otherwise claiming errors. Specifically, if the depositor is a contract, a *fallback()* or *receive()* function must be implemented. As this requirement is not clearly documented, it is recommended to clearly highlight this behavior by adding a comment to *requestL2TransactionTwoBridges()*.

TRST-R-5 Cleanup FacetToSelectors.facetPosition when removing facet from diamond

If there are no remaining selectors for a diamond facet after removing or replacing a function selector, the facet is removed from the **DiamondStorage.facets** storage array in *Diamond._removeFacet()*. However, **DiamondStorage.facetToSelectors[facet].facetPosition** is not deleted and continues to store the facet's original position in the **DiamondStorage.facets** array. While not causing any further implications, it is recommended to delete the value, similarly to how it is done in Nick Mudge's *DiamondLib* implementation of the EIP-2535 Diamond Standard.

TRST-R-6 Adjust MAX_L2_TO_L1_LOGS_COMMITMENT_BYTES constant

The constant **MAX_L2_TO_L1_LOGS_COMMITMENT_BYTES** in Config.sol sets the upper limit for the byte length of the L2->L1 system logs at 512 times **L2_TO_L1_LOG_SERIALIZE_SIZE**.

```
/// @dev The maximum length of the bytes array with L2 -> L1 logs
uint256 constant MAX_L2_TO_L1_LOGS_COMMITMENT_BYTES = 4 + L2_TO_L1_LOG_SERIALIZE_SIZE
* 512;
```

Previously, the system log length was encoded in the byte array, requiring an additional "4" bytes. Now, since the log length is no longer encoded in the byte array, this extra "4" bytes is no longer necessary and can be safely removed.

TRST-R-7 Deprecate the ZkSyncHyperchainStorage.blobVersionedHashRetriever state variable

The **ZkSyncHyperchainStorage.blobVersionedHashRetriever** address refers to a utility contract used to retrieve the EIP-4844 blob versioned hashes before official Solidity support. This contract is now obsolete because the **BLOBHASH** EVM opcode is available in yul. It is

recommended to deprecate the variable by adopting the __DEPRECATED_ naming convention and removing it from the diamond initialization process.

TRST-R-8 Use custom Solidity types to safely differentiate between uint256 L2 batch and message numbers in ZkSyncHyperchainStorage.isEthWithdrawalFinalized

The **ZkSyncHyperchainStorage.isEthWithdrawalFinalized** nested storage mapping uses uint256 for both the L2 batch number and the message number keys. To avoid accidentally using the wrong key, it is recommended to use custom Solidity types:

```
type L2BatchNumber is uint256;
type L2ToL1MessageNumber is uint256;

struct ZkSyncHyperchainStorage {
    // [...]

    mapping(L2BatchNumber l2BatchNumber => mapping(L2ToL1MessageNumber
l2ToL1MessageNumber => bool isFinalized)) isEthWithdrawalFinalized;

    // [...]
}
```

TRST-R-9 secondBridgeAddress check must be performed with an aliased address

In *Bridgehub.requestL2TransactionTwoBridges()*, it is checked that **secondBridgeAddress** is greater than **0xffff**. The purpose of this check is to ensure that **secondBridgeAddress** is not an address within the system contracts range of L2.

However, this check is wrong because in the L1 -> L2 transaction, **secondBridgeAddress** is aliased, and so **secondBridgeAddress > 0xffff** does not ensure that **alias(secondBridgeAddress) > 0xffff**. For example, the alias of **0xEeeeffFfFffFFFfFFFFFFFFffffffFfFFFfEEeF** is **0x0000000000000000000000000000000000000000**.

By calculating the probability of finding an **address** for which **alias(address) <= 0xffff**, it can be shown that the check is unnecessary in the first place. The probability that a random address is a system contract address is **2^16/2^160 = 2^-144**. Therefore, the probability to get at least one system contract address from **n** addresses is **1** minus the probability of getting only non-system addresses, i.e., **P("at least one system contract") = 1 - (1 - 2^-144)^n**. It is not feasible to compute a sufficient number of addresses.

In conclusion, since the **secondBridgeAddress > 0xffff** check is incorrect and unnecessary, it is recommended to remove it. If it should be kept, it is recommended to add it in all places where a L1 -> L2 transaction is created. *Bridgehub.requestL2TransactionTwoBridges()* is not the only place where the check would be needed.

TRST-R-10 Explicitly revert in L2WrappedBaseToken.bridgeBurn()

Bridging *L2WrappedBaseToken* is currently not supported. *L2WrappedBaseToken.bridgeMint()*, called by *L2SharedBridge.finalizeDeposit()*, explicitly reverts with the **"bridgeMint is not implemented! Use deposit/depositTo methods instead"** error. Initiating a withdrawal to L1 with *L2SharedBridge.withdraw()* calls *L2WrappedBaseToken.bridgeBurn()*, which does not fail outright. Instead, it implements the burning mechanism. However, sending unwrapped Ether to *L2SharedBridge* will revert anyway because it is missing a *receive()* or *fallback()* function. Therefore, to ensure consistent behavior, it is recommended to explicitly revert in *bridgeBurn()*.

TRST-R-11 Check protocol version deadline in AdminFacet.upgradeChainFromVersion()

protocolVersionDeadline is the timestamp after which it is no longer possible to commit batches for this protocol version. At that time, the version is considered outdated.

However, it is possible to upgrade to such an outdated version with *AdminFacet.upgradeChainFromVersion()*, while not being able to commit new batches anymore. It is recommended to implement a sanity check which ensures that the upgraded **s.protocolVersion** is not regarded as outdated by the *StateTransitionManager* (STM).

TRST-R-12 Outdated DiamondInit.initialize() flow

DiamondInit.initialize() fails to initialize the storage variables **baseTokenGasPriceMultiplierNominator** and **baseTokenGasPriceMultiplierDenominator** without which the *Mailbox* cannot operate. Those variables are instead initialized at a later point using Admin.*setTokenMultiplier()*. Moreover, **__DEPRECATED_verifierParams** is still being initialized despite being deprecated and no longer in use. It is recommended to remove all instances of **__DEPRECATED_verifierParams** and to ensure that **baseTokenGasPriceMultiplierNominator** and **baseTokenGasPriceMultiplierDenominator** are initialized in *DiamondInit.initialize()*.

TRST-R-13 Function selector collision in DiamondProxy between fallback() function and legitimate 0x00000000 selector

In *DiamondProxy*, both a *fallback()* function and a function identified by the selector 0x00000000 can be enabled by registering a facet for the selector 0x00000000. In Solidity, empty **msg.data** is treated differently from a selector with all zeros, while *DiamondProxy* interprets both identically. This can cause a problem where both the *fallback()* and the function with the 0x00000000 selector are unintentionally exposed, even if governance intended to expose only one, accidentally allowing users to call both functions.

TRST-R-14 Support changes in the EIP-4844 point evaluation precompile

ExecutorFacet._pointEvaluationPrecompile() verifies that the last 32 bytes returned by the point evaluation precompile match the predefined constant **BLS_MODULUS = 52435875175126190479447740508185965837690552500527637822603658699938581184513**.

According to EIP-4844, it is possible that the precompile will change, for example to switch to Merkle trees + STARKs.

It is recommended to monitor any changes to the Ethereum point evaluation precompile and to update the check in *ExecutorFacet* accordingly.

TRST-R-15 Clean higher bits in UnsafeBytes.readUint32()

UnsafeBytes.readUint32() returns a **uint32** value read from a 32-byte memory slot, which might not have the high bits set to zero. In version 0.8.26 of the Solidity compiler, the function currently masks these higher bits to zero without any errors. However, future versions might throw an exception for this behavior. To prevent such errors from causing an impact on the protocol (if and when the contract would be upgraded or redeployed on an affected compiler), it is recommended to explicitly clean the higher bits in *UnsafeBytes.readUint32()*. Similarly, *UnsafeBytes.readAddress()* would benefit from this mitigation as well.

TRST-R-16 Update Solidity compiler version

The Solidity version used for *l1-contracts* is 0.8.24 and the Solidity version used for *l2-contracts* and *system-contracts* is 0.8.20.

In the communication with the client, it has been determined that a more recent Solidity version should be used to mitigate non-deterministic compiler behavior.

TRST-R-17 Use the assertion error's actual length for the revert statement

The *assertionError()* utility function in the bootloader reverts the current execution with the provided error string literal **err**. The string literal is ABI encoded and put into memory before providing the memory locations to *revert(p, s)*. However, while the actual string's length is used for encoding, the specified memory location range length is hardcoded to 32 bytes, i.e., **s = 32**. To use the exact data range, it is recommended to use *getStrLen(err)* instead.

TRST-R-18 Remove the duplicate _setL2BlockHash() call in _upgradeL2Blocks()

In the case of an upgrade, *setL2Block()* in the bootloader calls *_upgradeL2Blocks()* followed by *_setNewL2BlockData(),* both of which internally call *_setL2BlockHash()* with the same parameters, resulting in an unnecessary duplicate function call. The *_setL2BlockHash() call in _upgradeL2Blocks()* can be safely removed.

TRST-R-19 Allow upgrade transactions to use base assets on L2

The L1 -> L2 upgrade transaction cannot use a non-zero **L2CanonicalTransaction.value** due to the *TransactionValidator.validateUpgradeTransaction()*, called in *BaseZkSyncUpgrade*, preventing it. As a result, the upgrade transaction on L2 holds no native assets, i.e., **msg.value = 0**. Therefore, without explicitly disabling the *TransactionValidator.validateUpgradeTransaction()* check during the initialization of the upgrade on L1, it is not possible to use *ContractDeployer.forceDeployOnAddresses()* with native assets.

When removing this restriction, it is recommended that upgrade transactions also deposit base assets in the shared bridge to ensure that L2 base assets are fully backed by the corresponding assets on L1.

TRST-R-20 Include gas consumed by setPubdataInfo() in the preparation gas accounting

In the bootloader, *setPubdataInfo()* is called in *l1TxPreparation()* before snapshotting the current gas in **gasBeforePreparation**. Consequently, the consumed gas by calling *setPubdataInfo()* is not included in the preparation gas. To ensure the used gas is accurately determined, it is recommended to snapshot the current gas before calling *setPubdataInfo()*.

Note that by implementing this change, the corresponding code in *TransactionValidator.getMinimalPriorityTransactionGasLimit()* must also account for the increased cost in *l1TxPreparation()*.

TRST-R-21 Change L1_TX_DELTA_FACTORY_DEPS_PUBDATA from 64 to 65 bytes

The **L1_TX_DELTA_FACTORY_DEPS_PUBDATA** constant represents the maximum number of pubdata bytes required to publish the state diff for a factory dependency. This value is currently set to 64, reserving 32 bytes for the derived storage key and 32 bytes for the value. To also include the 1-byte state diff metadata and to ensure conservative gas fee calculations, it is recommended to change the value from 64 to 65.

TRST-R-22 Enforce that state diff metadata length bits are set to zero in Compressor.verifyCompressedStateDiffs()

In *Compressor.verifyCompressedStateDiffs()*, if the state diff value is uncompressed and thus the operation is **0**, the 5 metadata bits that otherwise represent the length of the compressed value are unused and should be validated to be 0 to prevent random data being added to the state diff.

```
diff --git a/system-contracts/contracts/Compressor.sol b/system-
contracts/contracts/Compressor.sol
index f52c18ed..0bb6531c 100644
--- a/system-contracts/contracts/Compressor.sol
+++ b/system-contracts/contracts/Compressor.sol
@@ -144,6 +144,7 @@ contract Compressor is ICompressor, ISystemContract {
             stateDiffPtr++;
             uint8 operation = metadata & OPERATION_BITMASK;
             uint8 len = operation == 0 ? 32 : metadata >> LENGTH_BITS_OFFSET;
+            if (operation == 0) require(metadata >> LENGTH_BITS_OFFSET == 0);
             _verifyValueCompression(
                 initValue,
                 finalValue,
@@ -175,6 +176,7 @@ contract Compressor is ICompressor, ISystemContract {
             stateDiffPtr += 1;
             uint8 operation = metadata & OPERATION_BITMASK;
             uint8 len = operation == 0 ? 32 : metadata >> LENGTH_BITS_OFFSET;
+            if (operation == 0) require(metadata >> LENGTH_BITS_OFFSET == 0);
             _verifyValueCompression(
                 initValue,
                 finalValue,
```

TRST-R-23 Nested force contract upgrades can lead to immutable values unexpectedly being set

In the unlikely event that a system contract upgrade using *ContractDeployer.forceDeployOnAddresses()* results in nested upgrades, where *forceDeployOnAddresses()* is called from within the newly upgraded contract's constructor and attempts to upgrade the contract at the same address again, the outermost upgrade would overwrite the immutables that were set by the "inner" upgrades. This is possible because *ImmutableSimulator.setImmutables()* does not verify whether an immutable key-value pair has already been set.

TRST-R-24 Always use revertWithReason() in case of an error in appendTransactionHash()

L2 transaction validation is done by the bootloader in *l2TxValidation()*. This function calls *appendTransactionHash()* with the second parameter **isL1Tx** set to **false**. In case of an error, this results in a panic by exhausting the current frame's gas. However, this behavior is usually only intended for near calls, while in this case, *appendTransactionHash()* is always called in the bootloader's frame. Consequently, it is not necessary to differentiate the erroring method

in *appendTransactionHash*() based on **isL1Tx**. Instead, it is recommended to always use *revertWithReason()*.

TRST-R-25 StateTransitionManager.registerAlreadyDeployedHyperchain() should validate _chainId is less than type(uint48).max

Contrary to *Bridgehub.createNewChain()*, the owner-protected *StateTransitionManager.registerAlreadyDeployedHyperchain()* does not check whether the provided **_chainId** is less than **type(uint48).max**. If a chain with a very high id is registered, the bootloader will revert in encodeLegacyTransactionHash() when calculating **8 + block.chainid * 2**, caused by an arithmetic overflow error as the resulting value exceeds the maximum uint256 value. Therefore, it is recommended to enforce that the **_chainId** is less than **type(uint48).max** in *registerAlreadyDeployedHyperchain()*.

TRST-R-26 Revert with a custom error message only if data.length < 4 during diamond initialization

In *Diamond._initializeDiamondCut()*, if the *delegatecall()* failed, it checks if **data.length <= 4** to determine whether an error message was returned. However, as custom errors are ABI encoded and the first 4 bytes contain the error selector, this check would lead to custom errors that don't have additional values encoded to not get propagated. Instead, it reverts with the rather meaningless error message "I". To ensure such custom errors are correctly propagated, it is recommended to change the check to **data.length < 4**.

TRST-R-27 Provide AccountInfo to ContractDeployer.forceDeployOnAddresses() instead of using hardcoded values

The governance-protected *ContractDeployer.forceDeployOnAddresses()* sets the deployed contract's account info to **supportedAAVersion = AccountAbstractionVersion.None** and **nonceOrdering = AccountNonceOrdering.Sequential**. Instead of using hardcoded values, it would allow for more flexibility if those values could be supplied to *forceDeployOnAddresses().*

TRST-R-28 Accounts that use sequential nonce ordering can deadlock themselves

The *NonceHolder* contract allows an account with sequential nonce ordering to call *setValueUnderNonce()* if **key – 1** is a used min nonce.

Accordingly, an account can set a value under a nonce for which the min nonce has not been used yet. If it then tries to use *incrementMinNonceIfEquals()* for verifying new transactions, the account is deadlocked. The next min nonce is the nonce for which a value has been set, so the bootloader doesn't accept the min nonce.

To illustrate this with an example, assume that the latest min nonce is **10**. The current transaction that has used min nonce **10** then sets a value under nonce **11**. Subsequent transactions won't be possible if the account uses *incrementMinNonceIfEquals()* to verify nonces because the next min nonce is **11** which is not allowed by the bootloader.

Note that this scenario relies on custom account implementations, and as such is not an issue itself, but rather a behavior to be aware of and to document.

TRST-R-29 Parameters in EfficientCall and SystemContractHelper libraries are not masked

Some functions in the *EfficientCall* and *SystemContractHelper* libraries mask the parameters that have a length of less than 256 bits. However, it is done inconsistently, and some parameters are not cleaned up. For example, *EfficientCall.rawMimicCall()* and *EfficientCall.rawCall()* accept an **_address** parameter that is not cleaned up.

It is recommended to be consistent in the libraries, and to clean all parameters that occupy less than 256 bits. The libraries with inconsistent masking are *EfficientCall* and *SystemContractHelper*. Other libraries, that currently don't use masking at all, need to be assessed whether they should mask the parameters.

TRST-R-30 MAX_NUMBER_OF_HYPERCHAINS can cause DOS when chain creation is made permissionless

Once chain creation is made permissionless, which is stated as a long-term goal in the ZKsync documentation, the **MAX_NUMBER_OF_HYPERCHAINS** parameter that is used in *StateTransitionManager* to limit the number of chains that can be created, can be abused in a DOS attack. An attacker can create the maximum number of chains, preventing legitimate users from creating new chains. Before chain creation is made permissionless, any restrictions on the number of chains that can be abused in a DOS attack, must be removed.

TRST-R-31 Remove unused code

The *getCurrentCompressedBytecodeHash()* function in the *bootloader* is unused and can be removed.

The constants for the state diff compression in *Constants.sol* are outdated and never used. Also, some of them are incorrect. For example, **INITIAL_WRITE_STARTING_POSITION** should be **2** instead of **4**. Since the constants are never used, it is recommended to remove them.

TRST-R-32 Check that new facet address is not equal to old facet address in Diamond._replaceFunctions()

It is recommended to check that the new **_facet** address is not equal to **oldFacet.facetAddress** in *Diamond._replaceFunctions()*. The name of the function, *_replaceFunctions()*, suggests that functions are replaced. If the old facet and the new facet are the same, there is no replacement, and it shouldn't be accepted.

```
            SelectorToFacet memory oldFacet = ds.selectorToFacet[selector];
             require(oldFacet.facetAddress != address(0), "L"); // it is impossible to
replace the facet with zero address
+            require(oldFacet.facetAddress != _facet);

             _removeOneFunction(oldFacet.facetAddress, selector);
```

TRST-R-33 Getters.isFacetFreezable() returns false for non-existing facet

If **_facet** does not exist, i.e., when there is no selector registered for it, it is recommended to revert. This also makes *isFacetFreezable()* consistent with *isFunctionFreezable()*.

```
         uint256 selectorsArrayLen = ds.facetToSelectors[_facet].selectors.length;
-        if (selectorsArrayLen != 0) {
-            bytes4 selector0 = ds.facetToSelectors[_facet].selectors[0];
-            isFreezable = ds.selectorToFacet[selector0].isFreezable;
-        }
+        require(selectorsArrayLen > 0);
+        bytes4 selector0 = ds.facetToSelectors[_facet].selectors[0];
+        isFreezable = ds.selectorToFacet[selector0].isFreezable;
+
     }
```

TRST-R-34 address(0) is allowed to finalize L2 deposits if l1Bridge is not initialized

L2SharedBridge.finalizeDeposit() can be called by **l1Bridge** and **l1SharedBridge**. In the case that **block.chainid != ERA_CHAIN_ID**, **l1Bridge** is never initialized, and so it is possible for **address(0)** to finalize deposits without depositing on L1. Since **address(0)** can't be controlled by anyone except governance, the finding is informational. Still, it is recommended to check that **l1Bridge** has been initialized.

```
         require(
-            AddressAliasHelper.undoL1ToL2Alias(msg.sender) == l1Bridge ||
+            (AddressAliasHelper.undoL1ToL2Alias(msg.sender) == l1Bridge && l1Bridge
!= address(0)) ||
                 AddressAliasHelper.undoL1ToL2Alias(msg.sender) == l1SharedBridge,
             "mq"
         );
```

TRST-R-35 L2StandardERC20.reinitializeToken() allows to set ignoreDecimals but not decimals

L2StandardERC20.reinitializeToken() accepts a **_availableGetters** parameter which is a struct that contains the **ignoreDecimals** field. However, **decimals_** can't be changed by the function, so it is possible that **decimals_** and **_availableGetters.ignoreDecimals** become out of sync. It is recommended to check that **_availableGetters.ignoreDecimals** is the same as the previous **availableGetters.ignoreDecimals** value in storage or to accept a parameter that allows to change the decimals.

TRST-R-36 L1SharedBridge.bridgehubDeposit() should check that l2Value is zero

L1SharedBridge.bridgehubDeposit() is only called by *Bridgehub.requestL2TransactionTwoBridges()*. Since *L1SharedBridge* makes deposits with *L2SharedBridge* as its L2 counterpart, and *L2SharedBridge.finalizeDeposit()* is not payable, it is recommended to check that **l2Value** is equal to zero such as not to create deposits that fail on L2.

```
        require(l2BridgeAddress[_chainId] != address(0), "ShB l2 bridge not
deployed");
+       require(_l2Value == 0);

        (address _l1Token, uint256 _depositAmount, address _l2Receiver) = abi.decode(
            _data,
```

TRST-R-37 L1SharedBridge._parseL2WithdrawalMessage() should check that finalizeWithdrawal messages don't use baseToken as l1Token

It is possible to provide parameters to *L1SharedBridge._checkWithdrawal()* such that the **l1Token** that is encoded in the **_message** parameter is equal to the chain's base token, while the function signature that is encoded in the **_message** is **L1ERC20Bridge.finalizeWithdrawal.selector**.

As a result, a message that starts with **L1ERC20Bridge.finalizeWithdrawal.selector** can be attempted to be proven with **L2_BASE_TOKEN_SYSTEM_CONTRACT_ADDR** as the sender. This will fail, but it is recommended to add a sanity check in *L1SharedBridge._checkWithdrawal()* so that it's not possible to attempt to prove such messages.

```
            (l1Receiver, offset) = UnsafeBytes.readAddress(_l2ToL1message, offset);
            (l1Token, offset) = UnsafeBytes.readAddress(_l2ToL1message, offset);
            (amount, offset) = UnsafeBytes.readUint256(_l2ToL1message, offset);
+           require(l1token != BRIDGE_HUB.baseToken(_chainId));
        } else {
            revert("ShB Incorrect message function selector");
        }
```

TRST-R-38 bootloader.getFarCallABI() accepts forwardingMode parameter but does not allow to set dataOffset and memoryPage

Bootloader.getFarCallABI() is only used with **forwardingMode=0** (**UseHeap**), however it accepts a **forwardingMode** parameter. It is not safe to use the function with **forwardingMode=ForwardFatPointer** since **dataOffset** and **memoryPage** can't be set. To make the function compatible with all forwarding modes, it is recommended to also pass a **dataOffset** and **memoryPage** parameter.

TRST-R-39 Getters.storedBatchHash() can return hash of reverted batch

When batches are reverted, their information is not deleted from the **s.storedBatchHashes** mapping. Therefore, querying the hashes in this mapping via the *Getters.storedBatchHash()* and *Getters.storedBlockHash()* functions can return invalid results, which is a footgun for integrators.

It is recommended to revert if **_batchNumber > s.totalBatchesCommitted**. Reverting for invalid parameters is a pattern already adopted in *Getters.isFunctionFreezable()*.

TRST-R-40 Make safety checks in BaseZkSyncUpgradeGenesis specific to the genesis upgrade

BaseZkSyncUpgradeGenesis is only needed for the genesis upgrade. It is implemented by extending *BaseZkSyncUpgrade*, and making minor changes to its *_setNewProtocolVersion()* function.

It is possible to implement the genesis upgrade in a more straightforward way, and with tighter safety checks. Even if a separate genesis upgrade implementation is not feasible, the following change can be made:

```
require(
-     _newProtocolVersion >= previousProtocolVersion,
+     _newProtocolVersion == previousProtocolVersion,
    "New protocol version is not greater than the current one"
);
```

TRST-R-41 bootloader.ceilDiv() should revert for zero divisor instead of returning zero

Bootloader.ceilDiv() returns zero if the divisor is equal to zero. Division by zero is undefined, so it is better to revert in *ceilDiv()* instead of returning zero. Note that *ceilDiv()* is only called in *getFeeParams()* and *gasPerPubdataFromBaseFee()* where the divisor can only be zero if the operator provides invalid fee parameters.

TRST-R-42 TransactionHelper.payToTheBootloader() can be optimized to reduce redundant transfers

TransactionHelper.payToTheBootloader() is called in *DefaultAccount.payForTransaction()* to pay the bootloader for processing the transaction. The amount of base tokens that is paid is calculated as **_transaction.maxFeePerGas * _transaction.gasLimit**.

However, the amount of funds that needs to be sent to the bootloader is equal to **tx.gasprice * _transaction.gasLimit** which is less than or equal to the funds that are currently sent.

Even though the additional funds are immediately refunded by the bootloader, the refund incurs additional gas cost for which the user has to pay.

It is more efficient to only send **tx.gasprice * _transaction.gasLimit** in the first place.

```
    function payToTheBootloader(Transaction calldata _transaction) internal returns
(bool success) {
        address bootloaderAddr = BOOTLOADER_FORMAL_ADDRESS;
-       uint256 amount = _transaction.maxFeePerGas * _transaction.gasLimit;
+       uint256 amount = tx.gasprice * _transaction.gasLimit;

        assembly {
            success := call(gas(), bootloaderAddr, amount, 0, 0, 0, 0)
```

TRST-R-43 Use of secondBridgeAddress in Bridgehub can theoretically lead to impersonation

A user can call the *requestL2TransactionTwoBridges()* function in *Bridgehub* to use a secondary bridge as part of the bridging call. Note that it will make an external call to the **secondBridgeAddress**, validate that the output begins with **TWO_BRIDGES_MAGIC_VALUE**, and proceed to make an L2 call on behalf of that address using the output of the call as parameters for **l2Calldata**, **l2Contract** and **factoryDeps**. The apparent risk is that if an attacker can find a victim contract such that a call to *bridgehubDeposit()* could be made to return the magic value, followed by several somewhat controllable bytes (for example, an "echo" contract which outputs the input bytes), they would be able to transact on behalf of the victim on L2. This risk is disclosed in the *Bridgehub* documentation, however two improvements are suggested:

- Since the risk is generic and is not a risk specific to the *Bridgehub* integration, it should be disclosed to users in a more accessible place, to better reach potential victims.
- The risk could be greatly reduced by requiring implementers of **secondBridgeAddress** targets to adhere to ERC165 and have a discoverable interface. Indeed, it is not desired to interact with an address if it is not aware of itself being used as a hyperchain bridge.

TRST-R-44 Outdated or misleading comments and documentation

1. The comment in *IGetters* should refer to the last processed index + 1. **head** is incremented after popping the first unprocessed item from the *PriorityQueue*, pointing to the next unprocessed item in the queue.

```
- /// @dev If all the transactions were processed, it will return the last
processed index, so
```

```
+ /// @dev If all the transactions were processed, it will return the last
processed index + 1, so
```

2. The comment for **L1_TX_DELTA_FACTORY_DEPS_PUBDATA** is copy-pasted from the **L1_TX_DELTA_FACTORY_DEPS_L2_GAS** constant above. It should refer to the additional pubdata that is required per factory dependency.

```
/// @dev The number of L2 gas an L1->L2 transaction gains with each new
factory dependency
uint256 constant L1_TX_DELTA_FACTORY_DEPS_L2_GAS = 2473;

/// @dev The number of L2 gas an L1->L2 transaction gains with each new
factory dependency
uint256 constant L1_TX_DELTA_FACTORY_DEPS_PUBDATA = 64;
```

3. The NatSpec comment for *executeUpgrade()* is incorrect about the caller and should correctly mention that only the *StateTransitionManager* admin is permitted.

```
/// @notice Executes a proposed governor upgrade
/// @dev Only the current admin can execute the upgrade
/// @param _diamondCut The diamond cut parameters to be executed
function executeUpgrade(Diamond.DiamondCutData calldata _diamondCut) external;
```

   Similarly, the comment for *unfreezeDiamond()* should also only refer to the *StateTransitionManager* admin as the sole permitted caller.

```
/// @notice Unpause the functionality of all freezable facets & their
selectors
/// @dev Both the admin and the STM can unfreeze Diamond Proxy
function unfreezeDiamond() external;
```

4. The named return values **blockNumber** and **blockTimestamp** of the *getBatchNumberAndTimestamp()* function in *ISystemContext* should be renamed to **batchNumber** and **batchTimestamp**
5. The comment for **StateTransitionManagerInitializeData** should mention that the STM owner can actually perform critical updates, such as *executeUpgrade()* which force-upgrades a hyperchain's diamond contract.
6. Outdated NatSpec **@param** comments for **contractAddressL2** and **valueToMint**. The params are actually named **contractL2** and **mintValue** in the **BridgehubL2TransactionRequest** struct.
7. All occurrences of *IL2Messenger* in *L2ContractHelper* should be renamed to *IL1Messenger*.
8. *IContractDeployer.create2()* declared in *L2ContractHelper* should be marked **payable** to match the implementation in *ContractDeployer*.
9. The comment for **MAX_MEM_SIZE()** in the bootloader, that provides a reasoning for the hardcoded 63800000 value is incorrect. There exists no such derivation and the value is chosen arbitrarily (but still safe).
10. The *projectivePointCoordinatesAreOnFieldOrder()* function in *EcMul* is unused and can be safely removed.
11. The *montgomerySub()* function in *EcAdd* is unused and can be safely removed.
12. Some comments in the elliptic curve precompiles are duplicated and refer to the Montgomery multiplication when the respective function either performs a Montgomery addition or Montgomery subtraction. Those references in

EcAdd.yul#L247, EcAdd.yul#L235, EcMul.yul#L213, and EcMul.yul#L225 should be removed to avoid confusion.

13. *EcMul* has two comments defining the point at infinity in projective form as **(0,0,0)**. However, only the **z-coordinate** (third coordinate) matters for a point to be recognized as the point at infinity. It is recommended to adjust the two comments accordingly:
    - EcMul.yul#L307
    - EcMul.yul#L317
14. The comment for **MAX_POSTOP_SLOTS()** should refer to "slots" instead of "bytes".

```
/// @dev Each tx must have at least this amount of unused bytes before them to
be able to /// encode the postOp operation correctly. function
MAX_POSTOP_SLOTS() -> ret {
```

15. The comment in *getCanonicalL1TxHash()* incorrectly mentions that it is not enforced by system invariants that a ABI encoded transaction structure is prefixed with 0x20.

```
function getCanonicalL1TxHash(txDataOffset) -> ret {
    // Putting the correct value at the `txDataOffset` just in case, since
    // the correctness of this value is not part of the system invariants.
    // Note, that the correct ABI encoding of the Transaction structure starts
with 0x20
    mstore(txDataOffset, 32)
```

However, this invariant is checked in *validateAbiEncoding():*

```
function validateAbiEncoding(txDataOffset) -> ret {
    if iszero(eq(mload(txDataOffset), 32)) {
        assertionError("Encoding offset")
    }
```

16. The limit for **pubdataPrice** and **fairL2GasPrice** is not the same, as mentioned in the comment. The former is capped by **MAX_ALLOWED_FAIR_PUBDATA_PRICE()** while the latter is capped by **MAX_ALLOWED_FAIR_L2_GAS_PRICE()**. Additionally, instead of referring to "L1 gas", it should correctly refer to "L2 gas".

```
function validateOperatorProvidedPrices(fairL2GasPrice, pubdataPrice) {
    // The limit is the same for pubdata price and L1 gas price
    if gt(pubdataPrice, MAX_ALLOWED_FAIR_PUBDATA_PRICE()) {
        assertionError("Fair pubdata price too high")
    }

    if gt(fairL2GasPrice, MAX_ALLOWED_FAIR_L2_GAS_PRICE()) {
        assertionError("L2 fair gas price too high")
    }
}
```

17. The opcode to set the pubdata price no longer exists in VM version 1.5. Consequently, the **SET_PUBDATA_PRICE_CALL_ADDRESS** constant should be removed from *SystemContractsCaller*.
18. *MsgValueSimulator* should correctly mention here and here that the **extraAbi** parameters are located in three registers instead of two.
19. In *EfficientCall._loadFarCallABIIntoActivePtr()*, the **shrinkTo** variable has the meaning of "shrink-by", being the difference by which the active pointer is subtracted to get the resulting pointer length. It should be considered to rename the variable to **shrinkBy** to correctly convey its usage.

20. In *EcMul*, many links that reference the Wikipedia page regarding the REDC algorithm are broken due to the anchor being prefixed with an extra "The_". For example, here and here.
21. The notes section in the ZKsync Era Extension Simulation (verbatim) documentation mentions *"setting ergs per pubdata is done by separate opcode now"*. However, this OpContextSetErgsPerPubdataByte opcode was removed and does not exist in version 1.5, requiring an update of the documentation. In contrast, this is correctly highlighted in the formal specification.
22. According to the Contract size limit and format of bytecode hash documentation, the contract bytecode size is limited by the VM to 2^{16} 32-byte words, i.e. 2^{21} bytes. However, the compressor, which has a dictionary size of 2^{16} and 8 bytes item size, has a maximum bytecode limit of $2^{16} * 8 = 2^{19}$ bytes. A contract with bytecode larger than that cannot be compressed and must be published uncompressed. For completeness, this technical limit should be listed in the documentation.
23. The image used in the formal specification to explain narrowing of a fat pointer incorrectly refers to shrinking a fat pointer.
24. The formal specification for OpLoadPtrInc explains *"Read 32 consecutive bytes as a Big Endian 256-bit word from address **offset** in heap variant."*. However, it is read from **start + offset**.
25. The formal specification for *OpPrecompileCall* is missing the spending of the pubdata portion.
26. The formal specification states that in case the extra cost of a precompile call cannot be paid, no gas for the extra cost will be subtracted: *"If the cost is unaffordable, do not pay anything beyond base_cost of this instruction."*. But in log.rs it is observed that **ergs_remaining** is set to zero in this case, i.e., all available gas is spent.
27. The precompiles listed in Constants.sol and in the formal specification are incomplete and cover a different subset of precompiles. Both should have a complete reference of all available precompile addresses.
28. The comment in *BaseZkSyncUpgradeGenesis* incorrectly mentions that there is only a single change compared to the inherited *_setNewProtocolVersion()* implementation in *BaseZkSyncUpgrade*. However, lines 41-42 have also changed**.**

```
// IMPORTANT Genesis Upgrade difference: Note this is the only thing change >
to >=
require(
    _newProtocolVersion >= previousProtocolVersion,
    "New protocol version is not greater than the current one"
);
```

TRST-R-45 Mailbox should explicitly restrict the transaction gas according to the circuit costs

It is possible that a L1 -> L2 transaction uses too many circuits and thus becomes unprovable, halting the transaction queue. Circuits are paid for in gas, so limiting the gas of a L1 -> L2 transaction is an implicit protection. However, it is recommended to provide explicit protection. Since the circuits were not in scope of the audit, the calculation for the maximum gas amount must be made by the client.

TRST-R-46 Bootloader does not enforce FORBID_ZERO_GAS_PER_PUBDATA flag

Bootloader defines *FORBID_ZERO_GAS_PER_PUBDATA()*, which the name suggests is a flag that should prevent execution if **gasPerPubdataByteLimit = 0**. However, the flag is never enforced. There is no impact to not enforcing this check, but if it is considered to be a useful sanity check, it should be enforced. Otherwise, the flag should be removed.

TRST-R-47 Chain admin can abuse and front-run upgrades

The admin of a chain must not be able to cause irreversible damage to either its own chain or any other chain. Due to the flexibility of the upgrade mechanism, where the owner of *StateTransitionManager* creates new upgrades by calling *StateTransitionManager.setNewVersionUpgrade()*, which then become available for all chains and can be executed with *Admin.upgradeChainFromVersion()*, chain admins likely encounter situations in which they can harm their own chain.

setNewVersionUpgrade() sets the upgrade for a specific **_oldProtocolVersion**, and as long as the version of the chain matches **_oldProtocolVersion**, the chain admin can execute the upgrade. The upgrade is universal for all chains and can be applied by any chain admin for its own chain.

There are different scenarios that can cause problems. Suppose that the STM owner has a specific chain in mind that should receive the upgrade. If the wrong chain executes the upgrade, the upgrade can be applied to an unintended state, leading to undefined, and potentially dangerous, behavior.

More broadly speaking, the chain owner can apply the upgrade on any state, and it is possible that in combination with other actions that the chain admin can take (e.g., changing fee parameters, token multipliers, transaction filterer, performing L1 -> L2 priority transactions or L2 transactions), the upgrade is applied to an unintended state.

For a specific scenario, suppose that the upgrade makes old L1 -> L2 priority transactions incompatible with the new *bootloader* or *DefaultAccount* as defined by the **l2DefaultAccountBytecodeHash** and **l2BootloaderBytecodeHash** state variables. Now, the chain admin can execute an L1 priority transaction before the upgrade to intentionally enter an inconsistent state.

In the above scenario, it can be argued, it is the fault of the STM owner to make an upgrade available that allows an issue to arise in the first place. While it is true that an omniscient STM owner can prevent such scenarios, it is not practical to assume the STM owner can foresee all possibilities.

Yet another vector for the chain admin is to front-run a call to *StateTransitionManager.executeUpgrade()* (which allows STM owner to execute an upgrade immediately, without consent from chain admin), and to execute another upgrade prior to it, such that the overall state of the two upgrades applied after one another is inconsistent.

Since the chain admin is supposed to not be able to cause irreversible damage, additional guardrails need to be put in place to guide the upgrade process.

It is possible to acknowledge that STM owner must be very careful when making upgrades available to all chains via *setNewVersionUpgrade()*, and only such upgrades must be made available, that cannot possibly be exploited by the chain admin, users or any other actors.

On the other hand, there must be a way for the STM owner to execute upgrades that are potentially dangerous if the state of the chain changes before the upgrade is applied. It is important to note, and perhaps the root cause of the issue, that the *Admin* facet can't be frozen. That is because if it is frozen, the *unfreezeDiamond()* function also becomes unavailable, and the *Admin* facet and all freezable facets remain frozen forever.

A possible solution is to implement functions for privileged STM owner access in a separate facet from the chain admin functionality, such that the chain admin's access can be frozen, while the STM owner can still execute upgrades.

Centralization risks

TRST-CR-1 Protocol owner is fully trusted

All owner roles within the protocol are held by a decentralized governance. This includes the owner of *Bridgehub*, *StateTransitionManager* and *L1SharedBridge*. The decentralized governance also owns the proxies behind which some contracts are deployed. As such, the decentralized governance Is fully trusted and can change any component of the protocol.

TRST-CR-2 Bridgehub admin can create new chains

The owner of *Bridgehub* is allowed to call *Bridgehub.createNewChain()*, which allows to create new chains with a trusted base token and *StateTransitionManager*. Therefore, the parameters with which chains can be created are validated and the trust in the *Bridgehub* admin should be minimized. A malicious admin should not be able to cause damage apart from creating useless chains. However, in TRST-L-19 an issue has been discovered that can allow the *Bridgehub* admin to escalate its privileges and to steal funds.

TRST-CR-3 StateTransitionManager admin can set validators and revert batches

In *StateTransitionManager*, the admin is allowed to call *setValidatorTimelock()*, which is the default validator with which new chains are created, *revertBatches()* and *setValidator()*.

It is intended that the admin is not able to cause irreversible damage. However, TRST-L-10 describes how such irreversible damage can be achieved in some situations. The finding suggests taking additional measures to mitigate the impact of a malicious admin.

TRST-CR-4 Chain admins are trusted to manage their own chains

Admins of specific chains can call the following functions for their own chain:

- *Admin.changeFeeParams()*
- *Admin.setTokenMultiplier()*
- *Admin.setPubdataPricingMode()*
- *Admin.setTransactionFilterer()*
- *Admin.upgradeChainFromVersion()*

The goal is for the chain admin to govern its own chain without harming the ZKsync ecosystem. Upgrades to its chain must be scheduled by the protocol owner, and so the chain admin is only able to set chain parameters and make changes to its chain within the bounds that the protocol owner allows. Finding TRST-L-13 highlights a discrepancy with regards to which functions the chain admin is allowed and is not allowed to call, and recommendation TRST-R-

47 describes how upgrades can be abused to interfere with and cause damage during upgrades.

TRST-CR-5 Validators can commit, prove, execute and revert batches

Validators are set per chain by the admin or owner of *StateTransitionManager* and are allowed to call the following functions:

- *ExecutorFacet.commitBatches()*
- *ExecutorFacet.commitBatchesSharedBridge()*
- *ExecutorFacet.executeBatchesSharedBridge()*
- *ExecutorFacet.executeBatches()*
- *ExecutorFacet.proveBatches()*
- *ExecutorFacet.proveBatchesSharedBridge()*
- *ExecutorFacet.revertBatches()*
- *ExecutorFacet.revertBatchesSharedBridge()*

By reverting batches, validators can impact the availability of the chain, and by executing batches without delay, it is possible that batches are executed that exploit new attack vectors or abuse an inconsistent state. To mitigate instant finality, the validators will be timelocks such that malicious batches can be caught and reverted. The ability for the admin of *StateTransitionManager* to provide instant finality for batches of any chain has been reported in TRST-L-10.

TRST-CR-6 Operator of bootloader has limited powers in assembling batches

The operator is the entity that builds batches that are then committed on L1. Logically, it can be the same entity that commits and executes the batches on the L1 *ExecutorFacet* contract, although in practice there is a *ValidatorTimelock* as an intermediary such that the operator does not directly interact with the *ExecutorFacet*. The powers of the operator must thus be seen in the context of the *ValidatorTimelock* which could revert obviously malicious batches before they get executed.

The specific known operator abilities are listed below:

- L1 -> L2 transactions must be executed in order, but batches may contain none. In other words, L1 -> L2 transaction inclusion is not enforced.
- Choose inefficient bytecode and pubdata compression to inflate pubdata costs.
- Choose gas refunds for warm storage and memory access.
- Affect the execution of transactions by influencing gas costs.
- Freely choose gas per pubdata for all L2 transaction types except EIP712, which is the only L2 transaction type for which users can specify **gasPerPubdataByteLimit**.

Overall, the operator is trusted not to censor L2 transactions and to provide accurate gas refunds and pubdata costs. The contents of transactions as such can’t be tampered with, although the transaction execution can be indirectly influenced by the gas costs.

TRST-CR-7 Users must validate and trust paymaster logic

Users can choose to submit transactions that have a paymaster, i.e., a contract that is supposed to pay for the gas cost of the transaction.

In *DefaultAccount.prepareForPaymaster()*, the *DefaultAccount* is prepared for the paymaster by giving it the necessary token allowance (specified by the user in the transaction data) which the paymaster is supposed to exchange for L2 base tokens. Before interacting with a paymaster, users must validate the paymaster logic to make sure the transaction is paid for as expected.

Assuming there will be standard paymaster implementations provided by ZKsync, these can be used without additional trust assumptions. However, the concept of paymasters is general, and can lead to unexpected outcomes if not used properly.

It must also be considered that paymasters may not correctly handle token refunds, as described in TRST-PR-6.

Systemic risks

TRST-SR-1 Smart contracts developed for EVM can encounter unexpected differences on ZKsync Era

Unlike Ethereum, and many other Layer 2s, ZKsync Era smart contracts are compiled for and executed on a custom VM called EraVM. The differences that developers need to be aware of are provided in the ZKsync documentation. Being aware of just the differences in this section of the documentation is sufficient for most applications. However, specifically the gas fee model can cause problems. Beyond just different costs of opcodes, the gas fees for the EraVM introduce new trust assumptions between users and the operator (operator can choose gas refunds and pubdata costs) and callers and callees (natively, calls don't limit the pubdata expenditure of the callee; to mitigate this, there exists the *GasBoundCaller*).

TRST-SR-2 ZKsync Era chain can experience downtimes

It is possible that the ZKsync Era chain becomes unavailable. At such a time, new L2 transactions can't be submitted. While L1 -> L2 transactions can still be submitted in the L1 *Mailbox*, these won't be included in batches until the operator comes back online.

Depending on the intentions of the operator, L1 -> L2 transactions must not be executed before new L2 transactions are executed. In other words, users would not necessarily benefit from submitting L1 -> L2 transactions during a chain downtime. If the operator is not malicious, it can be expected that L1 -> L2 transactions will get executed first when the downtime is over, such that L1 -> L2 transactions provide an effective way to still interact with the L2 chain.

In the future, this risk may be mitigated by decentralizing the operator role.

TRST-SR-3 Overall protocol complexity

ZKsync overall is a complex protocol with different components involved. Of these, only the smart contracts on Ethereum and ZKsync Era were part of this audit. All the components must be in sync and maintained over time. For example, bug like this one in an external component (LLVM) which ZKsync uses, can impact the on-chain security of Era.

Users interacting with ZKsync must be aware that new vulnerabilities can be found in the components of ZKsync themselves and their third-party dependencies. While this is true for any software system, the complexity and uniqueness of ZKsync makes the risk worth highlighting.

TRST-SR-4 Gas prices at batch commitment can fluctuate

The L2 gas price for L1 -> L2 transactions is calculated at the time when they are submitted in the *Mailbox*. And the gas price for L2 transaction is set at the start of each batch. In theory, when the batch that includes the transaction is committed to L1, the price could have changed significantly to even cause a loss for the operator.

By keeping track of revenue over time, and averaging out short term fluctuations, a stable revenue can be generated. Still, the risk of a short-term loss must be considered.

For users, sudden spikes in the L1 gas price can lead to unexpectedly high L1 -> L2 transaction fees.

TRST-SR-5 External token risk in ZKsync bridges

The native ZKsync bridge allows to deposit and withdraw any token. There is no restriction like a whitelist for trusted tokens and the fact that a token has been deposited into the L1 bridge or the token's L2 counterpart has been deployed by the L2 bridge, does not mean that the token is trustworthy, or even compatible with the bridge. Therefore, users must do their own due diligence which tokens to trust.