TARAN SPACE

ZKsync

SECURITY AUDIT REPORT

of project **ZKsync OS** developed by **Matter Labs**

Initial version: August 4, 2025

Final version: October 15, 2025

TABLE OF CONTENTS

License/Disclaimer 3

About the Auditor Company 4

Understanding Audit Limits 5

Synopsis 6

- Audit Team .

About ZKsync OS 7

- General Architecture .
- Operation Layer .

Scope and Timeline 8

- Repository .
- Modules in Scope .
- Phase A .
- Phase B 9
- EVM Compatibility Requirements .
- Scope Limitations .

Findings Classification 10

- Severity Rating .
  - Likelihood .
  - Impact .
  - Final Severity 11
- Issue Statuses .

Summary of Issues 12

Detailed List of Issues 13

LICENSE

DISCLAIMER

The information contained within this audit report is made available on an "AS IS" basis, without any representations or warranties, express or implied. Neither the author(s) nor their employing entity shall be held liable for any losses or damages, direct or indirect, arising from or in any way connected to the use of this report. This document has been developed solely for the benefit and internal use of the specified client and does not establish any form of legal relationship with any third parties. The author(s) and their employer expressly disavow any liability or duty to any third parties, refusing to guarantee the report's accuracy or completeness. It is important to clarify that this report does not introduce any further obligations upon the company, including, but not limited to, warranties or liabilities. The copyright over this audit report is retained by the author(s).

This audit has been performed by:

Taran Space Co., Ltd. (ENG)
주식회사 타란스페이스 (KOR)

https://taran.space
info@taran.space

ABOUT THE AUDITOR COMPANY

Taran Space is a security research team offering reviews and other security-related services to Web3 projects, with the ultimate goal of fostering a stronger ecosystem. We conduct reviews of Layer 1 and Layer 2 protocols, cross-chain mechanisms, zero-knowledge circuits, web applications, mobile wallets, off-chain indexers, and more.

Importantly, Taran Space is not backed by any venture capital (VC) funds.

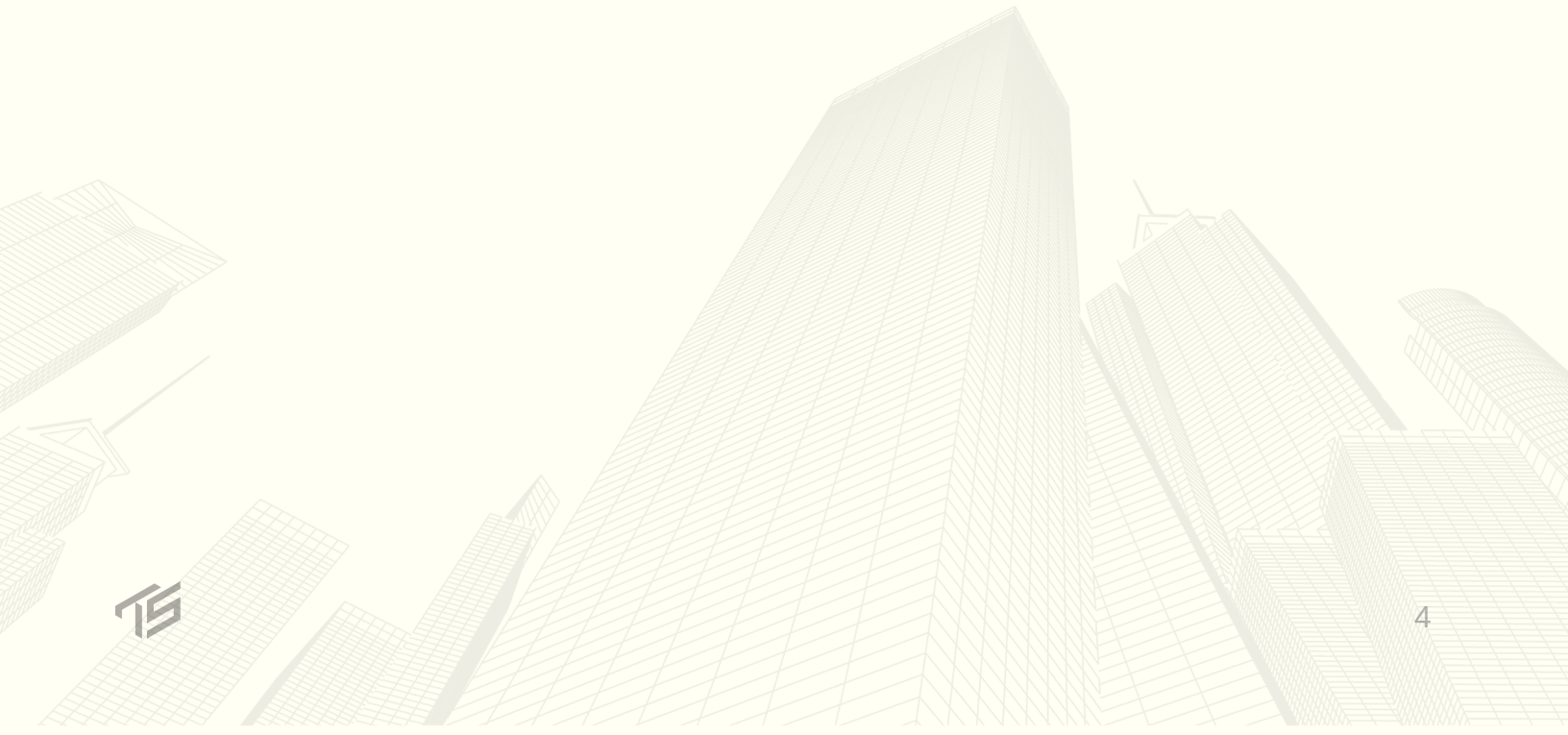
Discover more about us at our website **taran.space**.


Follow **@taran_space on X/Twitter**


Follow **@taran.space on BlueSky**


See other reports in our GitHub repository:
github.com/taran-space/audit-reports

If you're interested in exploring partnership opportunities with Taran.Space, please reach out to us at **hello@taran.space**.



UNDERSTANDING AUDIT LIMITS

While we diligently strive to identify as many issues as possible, by thoroughly examining every line of code, it is important to keep in mind that an audit may not uncover all potential vulnerabilities. No audit, including ours, can offer an absolute guarantee against the possibility of a hack. This inherent limitation stems from several factors:

- An audit represents a snapshot in time, capturing the state of the client's project at a specific commit. Given the dynamic nature of almost all software systems, which evolve and change over time, any modifications to the code post-audit necessitate a new security review to ensure continued safety.
- Even relatively small codebases can present a vast array of potential scenarios and code execution paths, each with its own complexities. This diversity means that there could always be parts of the attack surface that remain unexplored, no matter how comprehensive the audit.
- In the real world, the attack surface extends beyond the code itself to include deployment scripts, the supply chain, private key management systems, and potentially other components. To provide a more complete security assessment, such components should explicitly be included in the scope.
- Human factors also play a critical role in system security. Operators may alter deployment scripts, craft custom commands and governance proposals, or inadvertently compromise security through errors or oversights. While we endeavor to report any issues related to centralization, these may not always be flagged as of the highest severity. It is crucial for human operators to carefully review any scripts or governance proposals before executing them.

We have exerted our utmost effort to uncover all vulnerabilities within the system. However, our findings should not be considered an exhaustive list of all potential issues. The information provided in this document is for informational purposes only and should not be construed as investment or legal advice. The authors of this document cannot be held responsible for any decisions made based on its content.

Synopsis

In March 2025, Matter Labs engaged Taran Space to conduct a security assessment of their new product, ZKsync OS. Given the extensive and complex codebase, the audit was carried out in multiple phases. An initial four-week phase was conducted in April 2025 and resulted in private report. Following the remediation of the issues, Matter Labs decided to continue with the updated codebase and longer duration in order to facilitate a more thorough examination, discover as many potential issues as possible and to produce the public audit report.

This document summarizes the results of the next phase: a two-month audit conducted between June 9, 2025, and August 4, 2025. Following the audit, Matter Labs addressed the most severe security issues, and the implemented solutions were verified in September 2025.

Audit Team

The audit of ZKsync OS was conducted by a team comprising two full-time auditors:

- Lead Auditor: Kirill Taran
- Auditor: Tarek Elsayed

For specialized areas of the audit, particularly the EVM implementation and unsafe Rust techniques, additional experts were involved:

- Senior Auditor: Vitali Gorgut
- Auditor: Ilia Imeteo

About ZKsync OS

ZKsync OS, developed by Matter Labs, is a generalized RISC-V based state transition function for the ZKsync protocol. Implemented in Rust and compiled into a RISC-V binary, it can be proven using the `zksync-airbender`. This framework is designed to enhance Ethereum's scalability and composability through zero-knowledge rollup (zkRollup) technology, efficiently facilitating multiple execution environments (EVM, EraVM, Wasm, etc.) within a unified ecosystem.

Building upon the groundwork laid by zkSync Era, ZKsync OS features a modular architecture that not only supports the seamless operation of EVM and EraVM but enables the future integration of WebAssembly. This design includes a bootloader for aggregating these environments and increases the system's adaptability, enabling the customization of the framework to accommodate new execution environments and various storage models as per specific project needs.

General Architecture

The architecture of ZKsync OS can be considered as two main layers:

- **Operation layer** is a Rust-based program that can be upgraded independently of the proving circuits, enabling seamless upgrades and customization for app-specific chains.
- **Proving layer** can be fully optimized or replaced without affecting the operation layer, ensuring long-term flexibility and innovation.

Operation Layer

The operation layer serves as the system-level implementation of the chain's state transition function. It manages memory and I/O operations to maximize performance and efficiency. Additionally, it supports multiple virtual machines, each tailored to different use cases:

- **EVM (Ethereum Virtual Machine):** Ensures full compatibility with Ethereum.
- **EraVM:** Maintains backward compatibility with existing applications.
- **WASM (WebAssembly):** Expands possibilities by allowing smart contracts in various programming languages.
- **Native RISC-V Contracts:** Delivers maximum performance for high-efficiency use cases.

Scope and Timeline

Repository

The codebase is located in the repository by the following link:

- https://github.com/matter-labs/zksync-os

Modules in Scope

The scope is limited to specific modules of the operation layer:

- `basic_bootloader`
- `basic_system`
- `evm_interpreter`
- `zk_ee`

The proving layer is out-of-scope.

Phase A

Duration: 5 weeks

Dates: June 9, 2025 - July 14, 2025

Commit hash:

- `96d9d3701a5f5b7c47b42337ff4e70db2bd04223`
- coincides with the branch `taran-audit-phase-1`

Focus on general safety:

- State consistency, data integrity and validity
- EVM correctness and compatibility
- Funds safety, gas accounting
- Network security, performance and DoS vectors

Phase B

Duration: 3 weeks

Dates: July 14, 2025 - August 4, 2025

Commit hash:

- **`5e69d4483243bb0d166b7e2137fa54a8bb05925a`**
- coincides with the tag **`v0.0.5`**

Focus on L1 integration aspects as:

- L1 messaging
- L1 transactions
- L2 base token withdrawals
- Pubdata publishing and compression
- System upgrades

EVM Compatibility Requirements

For ZKsync OS, strict compatibility with the Ethereum Cancun virtual machine is essential. Compatibility deviations are classified as bugs unless they are pre-listed in the known issues document. Currently, Cancun's documentation is the main reference point for any virtual machine-related queries or clarifications.

Scope Limitations

During the audit, Account Abstraction (AA) was determined to be outside the audit's scope. The auditing team proceeded with the assumption that the **`AA_ENABLED`** configuration would be set to **`false`**, meaning that Account Abstraction features were not reviewed or considered during the review.

Findings Classification

Severity Rating

The severity rating is assigned to each issue after evaluating two factors: “likelihood” and “impact”.

Likelihood

Likelihood reflects the ease of exploitation, indicating how readily attackers could exploit a finding. This assessment considers the required level of access, the availability of exploitation information, the necessity of social engineering, the presence of race conditions or brute-forcing opportunities, and any additional barriers to exploitation.

This factor is categorized into one of three levels:

- **High** Exploitation is almost certain to occur, straightforward to execute, or challenging but highly incentivized. The issue can be exploited by virtually anyone under almost any condition. Attackers can exploit the finding unilaterally without needing special permissions or encountering significant obstacles.
- **Medium** Exploitation of the issue requires non-trivial preconditions. Once these preconditions are met, the issue is either easy to exploit or well incentivized. Attackers may need to leverage a third party, gain access to non-public information, exploit a race condition, or overcome moderate challenges to exploit the finding.
- **Low** Exploitation requires stringent preconditions, possibly requiring the alignment of several unlikely factors or a preceding attack. It might involve implausible social engineering, exploiting a challenging race condition, or guessing difficult-to-predict data, making it unlikely. There is little or no incentive to exploit the issue. Centralization issues, exploitable only by operators, on-chain governance, or development teams, fall into this category as well.

Impact

Impact considers the consequences of successful exploitation on the target system. It accounts for potential loss of confidentiality, integrity, and availability, as well as potential reputational damage.

This factor is classified into one of three levels:

- **High** Entails the loss of a significant portion of assets within the protocol, or causes substantial harm to the majority of users. Attackers can access or modify all data in a system, execute arbitrary code, escalate their privileges to superuser level, or disrupt the system’s ability to serve its users.

- **Medium** Leads to global losses of assets in moderate amounts or affects only a subset of users, which is still deemed unacceptable. Attackers can access or modify some unauthorized data, affect availability or restrict access to the system, or obtain significant internal technical insights.
- **Low** Losses are inconvenient but manageable. This applies to scenarios like easily remediable griefing attacks or gas inefficiencies. Attackers may access limited amounts of unauthorized information or marginally degrade system performance. This category also includes code quality concerns and non-exploitable issues that may still negatively impact public perception of security.

Final Severity

Upon assessing the issue's likelihood and impact, its severity is determined using the table below:

	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	QA

In exceptional cases, the total severity may be elevated to highlight the significance of the issue. Whenever this occurs, the rationale for applying an increased severity level is detailed in the report.

Issue Statuses

The status of an issue can be one of the following:

- Notified indicates that the client has been informed of the issue but has either not addressed it yet or has acknowledged the behavior without planning to remediate it soon. This inaction may stem from the client not viewing the behavior as problematic, lacking a feasible solution, or intending to address it at a later date.
- Addressed is used when the client has made an effort to address the issue with partial success. In the case of complex issues, the provided fix may only resolve one of several points described.
- Fixed means that the client has implemented a solution that completely resolves the described issue.

Summary of Issues

Title	Severity	Status
1. Access list decoding can be exploited to submit invalid transactions	High	Fixed
2. Returndata buffer overflow can cause DoS or exploitable memory corruption	High	Fixed
3. Incorrect native cost calculation causes DoS	High	Fixed
4. Non-deterministic decoding allows effective DoS attacks	High	Fixed
5. Flawed addresses validation can lead to token loss	Medium	Fixed
6. Merkle proofs serialization can produce malformed data	Medium	Fixed
7. Overflow during memory heap expansion	Medium	Fixed
8. Potential "Use-After-Free" issues in the preimage cache	Low	Notified
9. Invalid safety assumption when deserializing data from oracle	Low	Notified
10. L1 transactions can pose strong DoS potential	Low	Addressed
11. Merkle tree implementation concerns	Low	Addressed
12. Missing validation of token transfer during tokens burning	Low	Notified
13. Missing validations during system upgrades	Low	Notified
14. Potential extra validations	Low	Notified
15. Unchecked `base_fee_per_gas` downcast	Low	Fixed
16. Unsafe assumption that `pubdata` cannot decrease after validation	Low	Addressed
17. Decreased performance due to redundant `pubdata` calculation	Low	Fixed
18. Double resource accounting is a deviation from EVM semantics	Low	Addressed
19. Inconsistent developer documentation	Low	Addressed
20. Incorrect opcode name in debug output	Low	Fixed
21. Code duplicates	QA	Notified
22. Error handling issues	QA	Notified
23. Magic numbers	QA	Notified
24. Misleading docs and messages	QA	Notified
25. Redundant code	QA	Notified
26. Unimplemented functionality	QA	Notified
27. Using raw `u8` type instead of `enum`	QA	Notified

Detailed List of Issues

1 Access list decoding can be exploited to submit invalid transactions

Severity: High Commit: 96d9d37 Impact: High Likelihood: Medium Status: Fixed

The access list parser converts every ABI-encoded value of type **`uint256`** to the type **`usize`**, silently discarding the upper bits:

basic_bootloader/src/bootloader/transaction/access_list_parser.rs

```
fn new(slice: &'a [u8], offset: usize)
40    let bytestring_len = Self::parse_u256(slice, offset)?.as_limbs()[0] as usize;
```

```
52    // For now, it only has the access list
53    let outer_offset = Self::parse_u256(slice, offset)?.as_limbs()[0] as usize;
54    let outer_base = offset + outer_offset;
55    let outer_len = Self::parse_u256(slice, outer_base)?.as_limbs()[0] as usize;
```

This truncation affects 8 critical variables: **`access_list_rel_offset`**, **`bytestring_len`**, **`outer_offset`**, **`outer_len`**, **`item_ptr_offset`**, **`keys_ptr_offset`**, **`keys_len`** and **`count`**. Transaction hash calculation in the **`apply_access_list_encoding_to_hash()`** function uses these truncated values instead of the original bytes.

As a consequence, attackers can craft valid signatures for invalid transactions. For example, a transaction with **`keys_len`** set to **`0x10000000000000001`** is processed and verified as if **`keys_len`** had value **`1`**. However, the transaction retains the full value, a discrepancy that can be exploited for malicious purposes. Since the transaction hash is calculated using the truncated value, the signature verification succeeds.

Impacts include:

- Hash collisions between legitimate and malicious transactions
- Potential memory corruption if subsequent processing expects the actual data size to match the truncated count
- Potential DoS caused by non-determinism, if other protocol participants handle the truncation differently

Recommendation

Implement bounds checking in the `parse_u256()` function to reject any value exceeding `u32::MAX`. Add validation to ensure all access list length and offset values fit within expected bounds before processing and fail explicitly when this condition is not satisfied.

This issue has been resolved completely.

2 Returndata buffer overflow can cause DoS or exploitable memory corruption

Severity: High Commit: 96d9d37 Impact: High Likelihood: Medium Status: Fixed

The function `copy_into_return_memory` extends the `returndata_buffer` without validating its final size against the `MAX_RETURNDATA_BUFFER_SIZE` constant, which is currently defined as 128 MB.

basic_system/src/system_implementation/memory/basic_memory.rs
```
fn copy_into_return_memory
246    let new_returndata_start = self.returndata_buffer.len();
247
248    let Some(new_returndata_len) = new_returndata_start.checked_add(source.len()) else {
249        return Err(InternalError("OOM"));
250    };
251    let new_returndata_len = new_returndata_len.next_multiple_of(USIZE_SIZE);
252    self.returndata_buffer.extend_from_slice(source);
253    self.returndata_buffer.resize(new_returndata_len, 0);
```

basic_system/src/system_implementation/memory/basic_memory.rs
```
15    pub const MAX_RETURNDATA_BUFFER_SIZE: usize = 1 << 27; // 128 MB
```

Further, we will abbreviate the `MAX_RETURNDATA_BUFFER_SIZE` constant as *"128 MB."*

Impact of the issue

In the simplest scenario, a malicious actor can execute a smart contract that pushes unbounded data. This will exhaust memory and result in a Denial-of-Service condition.

Additionally, a more intricate attack is possible that forces reallocations and results in memory corruption, because the `copy_into_return_memory` function returns a structure of type `OSManagedResizableSlice`, which refers to the `returndata_buffer` before its expansion:

```
basic_system/src/system_implementation/memory/basic_memory.rs
fn copy_into_return_memory
250   self.returndata_buffer.extend_from_slice(source);
```

```
255   unsafe {
256       let start = self
257           .returndata_buffer
258           .as_mut_ptr()
259           .add(new_returndata_start);
260       let slice = OSManagedResizableSlice {
261           ptr: NonNull::new_unchecked(start),
262           len: source.len(),
263       };
264
265       Ok(slice)
266   }
```

The `returndata_buffer` variable is initialized as follows:

```
basic_system/src/system_implementation/memory/basic_memory.rs
fn new(allocator: Self::Allocator)
188   let returndata_buffer =
189       allocate_vec_usize_aligned(MAX_RETURNDATA_BUFFER_SIZE, allocator.clone());
```

When an attempt is made to grow the heap larger than *128 MB*, a call to `Vec::extend_from_slice` reallocates the data of the buffer, rendering all previously returned pointers as dangling. These dangling pointers can still be read later in the same transaction, e.g., when the VM passes the `returndata_buffer` to the caller. Dereferencing the dangling pointers leads to an immediate Use-After-Free condition, resulting in memory corruption potentially under the attacker's control.

Minimalistic Proof-of-Concept

The following test case demonstrates that the `returndata_buffer` variable is not protected from growing larger than `MAX_RETURNDATA_BUFFER_SIZE`:

```
#[cfg(test)]
mod tests {
    use super::*;
    use alloc::alloc::Global;
```

```
    #[test]
    fn returndata_buffer_can_grow_past_limit() {
        let mut mem = MemoryImpl::new(Global);

        // 150 MB payload (more than MAX_RETURNDATA_BUFFER_SIZE)
        let payload = vec![0u8; 150 << 20];

        // first call already exceeds the cap
        mem.copy_into_return_memory(&payload).unwrap();
        // the buffer keeps growing
        mem.copy_into_return_memory(&payload).unwrap();

        assert!(
            mem.returndata_buffer.len() <= MAX_RETURNDATA_BUFFER_SIZE,
            "returndata_buffer overflowed: {} bytes",
            mem.returndata_buffer.len()
        );
    }
}
```

To execute this test case, place it in the **`basic_system/src/system_implementation/memory/basic_memory.rs`** file.

Affected call chains

The **`copy_into_return_memory`** function is heavily utilized across the system, posing challenges for developers in ensuring its invocations do not push the **`returndata_buffer`** size over the **`MAX_RETURNDATA_BUFFER_SIZE`** limit.

Entry points leading to the **`copy_into_return_memory`** function include:

- The **`l1_messenger_hook_inner`** function, which could be called multiple times, accumulating message hashes in the **`returndata_buffer`**.
- The **`pure_system_function_hook_impl`** entry point, which could execute various system functions, two of which might request the allocation of huge data:
  - **`IdentityPrecompile`**, used for copying an arbitrary number of bytes.
  - **`ModExp`**, writing **`mod_len - output.len()`** bytes into the **`returndata_buffer`**.
- Multiple other functions reachable from the **`run_prepared`** entry point.

The function **`run_prepared`** initializes the **`returndata_buffer`** using **`System::init_from_oracle(oracle)`** before executing a batch of transactions and clears the buffer using **`self.clear_returndata_region()`** before executing every transaction. Thus, a single value of **`returndata_buffer`** never spans multiple transactions.

However, the buffer is not cleared between calls made within a single transaction. The `returndata_buffer` is only cleared after the transaction ends.

Difficulty of exploitation

A single malicious transaction could expand the `returndata_buffer` beyond *128 MB* without running into the Out-of-Gas condition via multiple attack paths. The most straightforward attack scenario is via the `IdentityPrecompile` system function. The attacker can choose a slice of call data to copy, and the `copy_into_return_memory` function appends it verbatim. For instance, copying 150 MB, or 4,710,912 words, would incur a gas cost of approximately 14.1M, according to the formula $15 + 3 \times \lceil len/32 \rceil$. This operation could be executed in one go or through multiple smaller segments, such as 10 chunks of 15 MB each.

Recommendation

Implementing size validation for the `returndata_buffer` or utilizing a pre-allocated fixed buffer would fully mitigate the risk.

This issue has been resolved completely.

3 Incorrect native cost calculation causes DoS

Severity: High Commit: 96d9d37 Impact: Medium Likelihood: High Status: Fixed

The `very_low_copy_cost` function miscalculates the native costs for copy operations. It incorrectly applies the constant `COPY_BASE_NATIVE_COST`, which is `80`, for both the per-byte multiplication and the base addition. The correct approach should utilize `COPY_BYTE_NATIVE_COST`, valued at `1`, for the per-byte cost.

```
evm_interpreter/src/utils.rs
pub fn very_low_copy_cost
306   let native = crate::native_resource_constants::COPY_BASE_NATIVE_COST
307       .checked_mul(len)?
308       .checked_add(crate::native_resource_constants::COPY_BASE_NATIVE_COST)?;
```

This results in significantly inflated native resource costs than intended:

- The expected cost for a copy operation is `1 * len + 80`
- However, the actual cost is calculated as `80 * len + 80`

Where `len` represents the length parameter in the `very_low_copy_cost` function.

For example, a 1 kilobyte copy operation would incur a charge of 82,000 native units instead of the anticipated 1,104 units — about 74 times more than expected.

The intended calculation method is evident from the naming convention and patterns observed elsewhere in the codebase. For instance, heap expansion costs follow a `base + (per_byte * bytes)` pattern, as it is observed in the same file:

```
evm_interpreter/src/utils.rsg
pub(crate) fn resize_heap
258   let net_cost_native = HEAP_EXPANSION_BASE_NATIVE_COST.saturating_add(
259       HEAP_EXPANSION_PER_BYTE_NATIVE_COST.saturating_mul(net_byte_increase as u64),
```

This miscalculation causes transactions with large copy operations to consume excessive native resources, risking transaction failure even with ample gas provisioned.

As a consequence, users can experience unexpected transaction failures for operations that should succeed. For instance, when deploying contracts with substantial bytecode or manipulating large data arrays.

While native resources are not directly deducted from the Ethereum balance of the sender, this overcharging leads to a higher rate of transactions failing without an apparent reason. This could be perceived as a Denial of Service (DoS) from the user's perspective.

Recommendation

Correct the pricing formula by replacing the first occurrence of `COPY_BASE_NATIVE_COST` with `COPY_BYTE_NATIVE_COST` to follow the anticipated `base + (per_byte * length)` pattern.

This issue has been resolved completely.

4 Non-deterministic decoding allows effective DoS attacks

Severity: High Commit: 5e69d44 Impact: Medium Likelihood: High Status: Fixed

This issue has been identified in two distinct system hooks, each with varying degrees of likelihood. First, we describe the Medium-severity instance, followed by an analysis of the High-severity case.

In the L1 Messenger hook

In the `l1_messenger_hook_inner` function, the variables `message_offset` and `length` are parsed into the type `usize`:

```
system_hooks/src/l1_messenger.rs
fn l1_messenger_hook_inner
154   let message_offset: usize = match U256::from_be_slice(&calldata[4..36]).try_into() {
```

```
187   let length =
188       match U256::from_be_slice(&calldata[length_encoding_end - 32..length_encoding_end])
189           .try_into()
```

Both conversions are explicitly validated to fit into the `usize` type, and the hook errors out with the message `"L1 messenger failure: sendToL1 called with invalid calldata"` if they do not.

Furthermore, the variables `length_encoding_end` and `message_end`, both of type `usize`, are introduced. The value of `length_encoding_end` is determined by validating `message_offset` to be `32` and then adding `36` to it. The value of `message_end` is calculated as the sum of two `usize` values, with potential overflow being addressed:

```
system_hooks/src/l1_messenger.rs
fn l1_messenger_hook_inner
198   let message_end = match length_encoding_end.checked_add(length) {
```

The snippets mentioned each have a potential failure related to the `usize` type: two stem from the `try_into` call and one from the `checked_add` call.

These failures are managed, but the use of `usize` introduces non-determinism into the protocol. If any of the `message_offset`, `length`, or `message_end` variables exceed `u32::MAX`—which would still be

accommodated by the `u64` type—the validation and execution via the `forward_system` would proceed successfully.

However, values exceeding `u32::MAX` would cause a failure in a node running the `proof_running_system`. Consequently, a single invalid transaction could fail the entire batch during the proving stage, long after resources have been expended on its validation.

In the L2 Base Token hook

The `l2_base_token.rs` file is subject to similar issues as well, particularly the `WITHDRAW_WITH_MESSAGE_SELECTOR` case. Similarly to the `l1_messenger_hook`, there are 3 failures related to the `usize` type:

```
system_hooks/src/l2_base_token.rs
fn l2_base_token_hook_inner
201   let message_offset: usize =
202       match U256::from_be_slice(&calldata[36..68]).try_into() {
```

```
222   let length =
223       match U256::from_be_slice(&calldata[length_encoding_end - 32..length_encoding_end])
224           .try_into()
```

```
232   let message_end =
233       match length_encoding_end.checked_add(length) {
```

For the L1 messenger, the likelihood of exploitation is reduced since the L1 contracts and governance must have validated all transactions before they are processed by the L2 nodes.

However, the `WITHDRAW_WITH_MESSAGE_SELECTOR` case of the L2 base token hook is a permissionless function, allowing any user to attach a message of dynamic size and trigger the issue. Unlike the transactions handled by the L1 messenger, these transactions do not undergo validation on L1 before being processed on L2.

Impact of the issue

This vulnerability opens a pathway for malicious actors to conduct a Denial-of-Service (DoS) attack at a very low cost.

An attacker could generate transactions that, while appearing valid, lead to failures during the L2 proving stage. This results in the rejection of entire blocks, containing not only the malicious transactions but also those of other users.

Recommendation

Consider using a fixed-sized type like `u32` for message-related variables and explicitly setting an even lower limit for the message size.

In L1 contracts, it is crucial to ensure that L1 transactions and system upgrade requests are properly validated. This validation should ensure that messages are not excessively long and that message parameters can comfortably fit within the `u32` range.

This issue has been resolved completely.

5 Flawed addresses validation can lead to token loss

Severity: Medium Commit: 5e69d44 Impact: Medium Likelihood: Medium Status: Fixed

In `l2_base_token.rs`, the `l2_base_token_hook_inner` function interprets the `calldata[4..36]` slice, consisting of 32 bytes, as an Ethereum address by taking the lowest 20 bytes. The slice is correctly validated, in the `WITHDRAW_SELECTOR` case, to not have non-zero bytes among the 12 highest bytes:

system_hooks/src/l2_base_token.rs

```
fn l2_base_token_hook_inner
170   // check that first 12 bytes in address encoding are zero
171   if calldata[4..4 + 12].iter().any(|byte| *byte != 0) {
172       return Ok(Err(
173           "Contract deployer failure: withdraw called with invalid calldata",
```

However, the alternative branch `WITHDRAW_WITH_MESSAGE_SELECTOR` of `l2_base_token_hook_inner` does not include a similar validation:

```
270   let mut message: alloc::vec::Vec<u8, S::Allocator> =
271       alloc::vec::Vec::with_capacity_in(message_length, system.get_allocator());
272   message.extend_from_slice(FINALIZE_ETH_WITHDRAWAL_SELECTOR);
273   message.extend_from_slice(&calldata[16..36]);
274   message.extend_from_slice(&nominal_token_value.to_be_bytes::<32>());
275   message.extend_from_slice(&caller.to_be_bytes::<20>());
276   message.extend_from_slice(additional_data);
```

The presence of non-zero bytes in the 12 highest bytes of an address could stem from errors in 3rd-party applications built on the rollup, or simply from a user's mistake. The corresponding Solidity smart contracts of the settlement layer would ignore these bytes, using only the lowest 20 bytes to determine the address. Such an inadvertent mismatch, whether it's an off-by-some offset or residual data in the upper 12 bytes, could result in the interpretation of a 20-byte address that doesn't align with the caller's intentions.

As a consequence of such a mistake, the user's tokens will be silently transferred to an invalid address, effectively resulting in the loss of those tokens without any immediate indication of the mistake to the user.

This issue has been resolved completely.

6 Merkle proofs serialization can produce malformed data

Severity: Medium Commit: 96d9d37 Impact: Medium Likelihood: Medium Status: Fixed

The structure **`FlatStorageLeaf`** implements trait **`UsizeSerializable`** and its function **`iter`** returning an iterator or type **`ExactSizeIterator`**. Such an iterator can provide its consumer with information about its length:

basic_system/src/system_implementation/flat_storage_model/simple_growable_storage.rs

```
52    impl<const N: usize> UsizeSerializable for FlatStorageLeaf<N> {
53        const USIZE_LEN: usize =
54            <Bytes32 as UsizeSerializable>::USIZE_LEN * 2 + <u64 as UsizeSerializable>::USIZE_LEN
* 2;
55
56        fn iter(&self) -> impl ExactSizeIterator<Item = usize> {
57            ExactSizeChain::new(
58                UsizeSerializable::iter(&self.key),
59                ExactSizeChain::new(
60                    UsizeSerializable::iter(&self.value),
61                    UsizeSerializable::iter(&self.next),
```

While the function **`iter`** is implemented correctly, the related constant **`USIZE_LEN`** is defined incorrectly.

The value of **`USIZE_LEN`** currently accounts for 2 values of type **`Bytes32`** and 2 values of type **`u64`**. However, the actual implementation provides one **`u64`** value less. The structure **`FlatStorageLeaf`** declares only one field typed as **`u64`**, named **`next`**.

Severity of the issue

In other implementations of the `UsizeSerializable` trait, the `iter` function validates the number of bytes consumed during serialization against the constant `USIZE_LEN`. This constant acts as a safeguard against invalid serialization, ensuring the integrity of the data being processed.

However, this validation step leveraging the `USIZE_LEN` constant is not implemented for `FlatStorageLeaf`, meaning there is no direct impact associated with the incorrect definition of it.

Still, the `USIZE_LEN` constant related to the structure `FlatStorageLeaf` is used to define similar constants for other structures:

- `ExistingReadProof`
- `ExistingWriteProof`
- `NewReadProof`
- `NewWriteProof`

These structures, in turn, influence the serialization of even more structures:

- `ValueAtIndexProof`
- `ReadValueWithProof`
- `WriteValueWithProof`

Data within these 7 data structures is critical for the correct functioning of Merkle tree-based storage. However, it is at risk of becoming malformed during the serialization process due to the potential influence of `USIZE_LEN`. For example, an incorrect definition of `USIZE_LEN` might lead to premature termination of serialization, among other potential issues.

Only because the incorrect definition is involved into serialization of 7 other critical proof-related structures, the likelihood and overall severity of the issue are both considered "Medium." This classification reflects a significantly broader attack surface compared to a similar issue reported in the previous audit, which was rated as "Low."

This issue has been resolved completely.

7 Overflow during memory heap expansion

Severity: Medium Commit: 96d9d37 Impact: Medium Likelihood: Medium Status: Fixed

The function `grow_heap` aims to increase the size of the specified region to match the parameter `new_size`, aligned to `usize`.

The function verifies if the region is managed by the `MemoryImpl` instance and positioned at the top of the managed heap; otherwise, it returns `InternalError`. It then aligns `new_size` to a multiple of `USIZE_SIZE` and checks that this aligned `new_size` does not exceed the `MAX_HEAP_BUFFER_SIZE`. If the capacity is exceeded, it returns `Ok(None)`.

basic_system/src/system_implementation/memory/basic_memory.rs
```
fn grow_heap
134         new_size: usize,
```

```
150     let new_size = new_size.next_multiple_of(USIZE_SIZE);
151
152     let current_heap_capacity = unsafe {
153         self.heap_buffer.len()
154             - self
155                 .current_heap_start
156                 .offset_from_unsigned(self.heap_buffer.cast())
157     };
158     if new_size > current_heap_capacity {
159         return Ok(None);
160     }
```

However, the validation against the heap's maximum capacity is ineffective due to the absence of a bounds check on the `new_size` parameter before aligning it to `next_multiple_of(USIZE_SIZE)`. This oversight can cause overflow and silent wrapping to zero in release mode when compiled with `overflow-checks=false`.

Overflow occurs when the caller-provided `new_size` exceeds `usize::MAX - (USIZE_SIZE - 1)`.

basic_system/src/system_implementation/memory/basic_memory.rs
```
fn grow_heap
171     Ok(Some(OSManagedResizableSlice {
172         ptr: self.current_heap_start,
173         len: new_size,
```

As a consequence of `new_size` wrapping to zero, heap expansion does not occur, and an `OSManagedResizableSlice` is returned with `len` set to zero. Any caller expecting the returned buffer to match the specified size would then run into an “Out-of-Bounds” write.

Minimalistic Proof-of-Concept

The following code snippet serves as an example of vulnerable usage of the `grow_heap` function:

```
let mut memory = ...;
let empty_region = memory.empty_managed_region();

// Returns a buffer with length 0
let mut resized_region = memory
  .grow_heap(empty_region, usize::MAX)
  .unwrap().unwrap();

let data = b"deadbeef";

// Panic: Range end index 8 out of range for slice of length 0
resized_region[..data.len()].copy_from_slice(data);
```

Actual usages in the codebase

However, there is only one potentially unsafe call chain in the codebase where the function `grow_heap` is called with arbitrary value of the `new_size` parameter.

In `evm_interpreter/src/utils.rs`, the `offset` parameter of the `resize_heap` function is directly influenced by certain opcodes, including `MSTORE`, `MLOAD` and `MCOPY`. After some manipulations, this parameter is utilized to call the `grow_heap` function at line 272.

This call path, though influenced by user input from a smart contract, is deemed safe in the specific case of the EVM execution environment, because the EVM interpreter constrains `new_size` values to `u32::MAX - 31`. However, the context can differ for other execution environments.

Recommendation

Return a well-typed error when the parameter `new_size` exceeds `usize::MAX - (USIZE_SIZE - 1)`.

This issue has been resolved completely.

8 Potential "Use-After-Free" issues in the preimage cache

Severity: Low Commit: 96d9d37 Impact: High Likelihood: Low Status: Notified

Several preimage cache operations bypass the lifetime limitations of the data stored in the `storage` field of type `BTreeMap`. This is done by employing the unsafe operation `core::mem::transmute`, which extends the lifetime to `'static` without guarantees. If the underlying `BTreeMap` reallocates, or if the actual data is freed, the `'static` reference will become dangling. Subsequent access to such a reference will cause a Use-After-Free condition, potentially leading to memory corruption, arbitrary code execution, or system crashes.

```
basic_system/src/system_implementation/flat_storage_model/preimage_cache.rs
fn expose_preimage
 82    if let Some(cached) = self.storage.get(hash) {
 83        unsafe {
 84            let cached: &'static [u8] = core::mem::transmute(cached.as_slice());
```

```
180   let inserted = self.storage.entry(*hash).or_insert(buffered);
181   // Safety: IO implementer that will use it is expected to live beoynd any frame (as it's part
of the OS),
182   // so we can extend the lifetime
183   unsafe {
184       let cached: &'static [u8] = core::mem::transmute(inserted.as_slice());
```

```
fn insert_verified_preimage(
199   let inserted = self.storage.entry(*hash).or_insert(preimage);
200
201   unsafe {
202       let cached: &'static [u8] = core::mem::transmute(inserted.as_slice());
```

In general, such approach can be considered a bad practice because it circumvents the borrow checker, which is a compiler mechanism designed to catch memory issues.

In the file `basic_system/src/system_implementation/flat_storage_model/preimage_cache.rs`, the primary entry points to this attack vector are the functions `get_preimage` and `record_preimage`.

The result of the `record_preimage` function is utilized only by `deploy_code`, where it lives for a short time and does not leave the function's scope. Although it is formally returned from the function, the actual callers discard the value.

A more detailed analysis is required to assess the risk posed by the `get_preimage` function. The value returned by this function is not only used by a few functions but is also stored in the `AccountData::bytecode` field during the execution of the `read_account_properties` function, which is called by multiple other functions during the transaction validation stage. Later, the value stored in `AccountData::bytecode` is retrieved by several functions, including the `evm_interpreter` module and execution handlers for both L1 and L2 transactions.

Matter Labs has internally verified the safety of these `transmute` operations but has not provided a technical justification for their confidence.

9 Invalid safety assumption when deserializing data from oracle

Severity: Low Commit: 96d9d37 Impact: Medium Likelihood: Low Status: Notified

The safety guarantee in the `materialize_element` function is dependent on the assumption that the `init_from_iter` function is correct and that `dst.assume_init()` is safe to call as long as `init_from_iter` has completed successfully:

basic_system/src/system_implementation/flat_storage_model/storage_cache.rs

```
fn materialize_element<'a>(
177     let mut dst =
178         core::mem::MaybeUninit::<InitialStorageSlotData<EthereumIOTypesConfig>>::uninit(
179         );
180     let mut it = oracle
181         .create_oracle_access_iterator::<InitialStorageSlotDataIterator<EthereumIOTypesConfig>>(
182             *address,
183         )
184         .expect("must make an iterator");
185     unsafe { UsizeDeserializable::init_from_iter(&mut dst, &mut it).expect("must initialize") };
186     assert!(it.next().is_none());
187
188     // Safety: Since the `init_from_iter` has completed successfully and there's no
189     // outstanding data as per line before, we can assume that the value was read
190     // correctly.
191     let data_from_oracle = unsafe { dst.assume_init() } ;
```

The `init_from_iter` function, in its turn, relies on the correct implementation of the `from_iter` method from the trait `UsizeDeserializable`:

```
zk_ee/src/kv_markers/mod.rs
pub trait UsizeDeserializable
50    unsafe fn init_from_iter(
51        this: &mut MaybeUninit<Self>,
52        src: &mut impl ExactSizeIterator<Item = usize>,
53    ) -> Result<(), InternalError> {
54        let new = UsizeDeserializable::from_iter(src)?;
```

This trait is implemented for multiple types but not all implementations enforce length of the output to match the corresponding `USIZE_LEN` constant. Specifically, the `materialize_element` function relies on the correct trait implementation for the `InitialStorageSlotData` structure.

Implementation of `UsizeDeserializable` for `InitialStorageSlotData`

In this particular case, the implementation does not validate the data being deserialized against the expected length defined by `USIZE_LEN`:

```
zk_ee/src/system_io_oracle/mod.rs
168   impl<IOTypes: SystemIOTypesConfig> UsizeSerializable for InitialStorageSlotData<IOTypes> {
169       const USIZE_LEN: usize = <bool as UsizeSerializable>::USIZE_LEN
170           + <u8 as UsizeSerializable>::USIZE_LEN
171           + <IOTypes::StorageValue as UsizeSerializable>::USIZE_LEN;
172       fn iter(&self) -> impl ExactSizeIterator<Item = usize> {
173           ExactSizeChain::new(
174               UsizeSerializable::iter(&self.is_new_storage_slot),
175               UsizeSerializable::iter(&self.initial_value),
176           )
177       }
178   }
179
180   impl<IOTypes: SystemIOTypesConfig> UsizeDeserializable for InitialStorageSlotData<IOTypes> {
181       const USIZE_LEN: usize = <Self as UsizeSerializable>::USIZE_LEN;
182
183       fn from_iter(src: &mut impl ExactSizeIterator<Item = usize>) -> Result<...
184           let is_new_storage_slot = UsizeDeserializable::from_iter(src)?;
185           let initial_value = UsizeDeserializable::from_iter(src)?;
186
187           let new = Self {
188               is_new_storage_slot,
189               initial_value,
190           };
191
192           Ok(new)
```

Incomplete `UsizeDeserializable` implementations

The **`UsizeDeserializable`** trait, utilizing the **`USIZE_LEN`** constant, offers additional protection against malformed data during deserialization by enabling developers to validate the number of bytes read. The constant **`USIZE_LEN`** is also usually shared with the **`UsizeSerializable`** trait to ensure data integrity during serialization.

```
zk_ee/src/kv_markers/kv_impls.rs
268   impl UsizeDeserializable for B160 {
269       const USIZE_LEN: usize = <B160 as UsizeSerializable>::USIZE_LEN;
270
271       fn from_iter(src: &mut impl ExactSizeIterator<Item = usize>) -> Result<Self,
InternalError> {
272           if src.len() < <Self as UsizeDeserializable>::USIZE_LEN {
273               return Err(InternalError("b160 deserialization failed: too short"));
274           }
```

Throughout the codebase, many types implement **`UsizeSerializable`** and **`UsizeDeserializable`**, including manually maintained values for the **`USIZE_LEN`** parameter. However, only two of such types implement the validation according to the aforementioned pattern: **`B160`** and **`Bytes32`** from the file **`zk_ee/src/kv_markers/kv_impls.rs`**.

Constants **`USIZE_LEN`** in all of the other types are not actually utilized, yet require being updated according to any internal structure changes.

Recommendation

Complete all **`UsizeDeserializable`** implementations by enforcing output lengths to match the corresponding **`USIZE_LEN`** constants. Ideally, these constants should not be maintained manually but an automated mechanism should derive them from the actual structure layouts.

Matter Labs has confirmed the issue and plans to implement a fix in upcoming releases.

10 L1 transactions can pose strong DoS potential

Severity: Low Commit: 5e69d44 Impact: Medium Likelihood: Low Status: Addressed

L1 transactions are initiated on the settlement layer: users can request “priority transactions” on the settlement layer, as well as the governance can schedule a system upgrade on L1. Hashes of such transactions are committed on-chain, so a batch containing them could be verified after publishing.

Such transactions are labeled using the `L1_L2_TX_TYPE` and `UPGRADE_TX_TYPE` constants and are processed using the `process_l1_transaction` function:

basic_bootloader/src/bootloader/process_transaction.rs

```
pub fn process_transaction
61    let tx_type = transaction.tx_type.read();
62
63    match tx_type {
64        ZkSyncTransaction::UPGRADE_TX_TYPE => {
65            if !is_first_tx {
66                Err(Validation(InvalidTransaction::UpgradeTxNotFirst))
67            } else {
68                Self::process_l1_transaction(
69                    system,
70                    system_functions,
71                    memories,
72                    transaction,
73                    false,
74                )
75            }
76        }
77        ZkSyncTransaction::L1_L2_TX_TYPE => {
78            Self::process_l1_transaction(system, system_functions, memories, transaction, true)
79        }
80
81        _ => Self::process_l2_transaction::<Config>(
```

For regular L2 transactions, the `process_l2_transaction` function ensures signature verification is performed via the calls to functions `transaction_validation`, `AA::validate` and `EOA::validate`. The latter function is defined in the file `basic_bootloader/src/bootloader/account_models/eoa.rs`.

However, the `process_l1_transaction` function does not include signature verification. Any user can attempt submitting a transaction in the L2 network which includes `L1_L2_TX_TYPE` or `UPGRADE_TX_TYPE` as the value of `tx_type`. Such values of the `tx_type` field would be invalid from the standard EVM perspective. If sequencer does not filter invalid values of the `tx_type` field, bogus L1 transactions could be validated, executed, proved and included into a batch for publication.

Since the settlement layer verifies the L1 transactions against the hashes, committed before, only authorized L1 transactions can actually be published. Unauthorized L1 transactions will be rejected together with all the other transactions included into the same batch. This leads to significant amount of computation lost and presents a strong DoS vector. An attacker needs to submit only one bogus L1 transaction to fail multiple legit transactions.

While the impact is significant, the likelihood of this issue depends on the specific implementation of the sequencer node, which is out of scope of this audit. From the bootloader and ZKsync OS perspective, this is a trust assumption that the sequencer rejects transactions with non-standard values of the `tx_type` field. It is not possible to mitigate this issue by means of the ZKsync OS since the L1 signers cannot be preliminary whitelisted.

Recommendation

Ensure that users cannot submit transactions with values of the `tx_type` equal to `L1_L2_TX_TYPE` or `UPGRADE_TX_TYPE` by rejecting such by the sequencer node.

The reference sequencer implementation appears to be safe since it does not implement permissionless relaying of L1 transactions, all L1 transactions are relayed by the sequencer itself. However, alternative sequencer implementations could be designed differently.

11 Merkle tree implementation concerns

Severity: Low Commit: 96d9d37 Impact: Medium Likelihood: Low Status: Addressed

Insufficient separation between testing and simulation code

One of the data structures related to the Merkle tree implementation is `FlatStorageBacking`, serving as a multi-purpose index and incorporating features for testing, such as the `RANDOMIZED` parameter:

```
basic_system/src/system_implementation/flat_storage_model/simple_growable_storage.rs
901   pub struct FlatStorageBacking<
902       const N: usize,
903       H: FlatStorageHasher,
904       const RANDOMIZED: bool,
```

This structure is utilized to define the `TestingTree` type, intended solely for testing purposes:

```
basic_system/src/system_implementation/flat_storage_model/simple_growable_storage.rs
1667  pub type TestingTree<const RANDOMIZED: bool> =
1668      FlatStorageBacking<TESTING_TREE_HEIGHT, Blake2sStorageHasher, RANDOMIZED>;
```

However, it also underpins the definition of the `InMemoryTree` type, employed in production code for transaction simulation:

```
api/src/lib.rs
pub fn run_batch_generate_witness
17    tree: InMemoryTree,
```

```
forward_system/src/run/test_impl/tree.rs
 8    pub struct InMemoryTree<const RANDOMIZED_TREE: bool = false> {
 9        /// Hash map from a pair of Address, slot into values.
10        pub cold_storage: HashMap<Bytes32, Bytes32>,
11        pub storage_tree: TestingTree<RANDOMIZED_TREE>,
12    }
```

Beyond concerns over code quality, there is a potential future issue given that the production structure `InMemoryTree` relies on the testing constant `TESTING_TREE_HEIGHT` instead of the production constant `TREE_HEIGHT`. Although both constants are currently set to `64`, this could change in the future and go unnoticed. Simulations of transactions should run against the most current, real state.

```
basic_system/src/system_implementation/flat_storage_model/mod.rs
55    pub const TREE_HEIGHT: usize = 64;
```

```
basic_system/src/system_implementation/flat_storage_model/simple_growable_storage.rs
1666  pub const TESTING_TREE_HEIGHT: usize = 64;
```

Recommendation

1. Rename the `TestingTree` structure and relocate the `InMemoryTree` definition out of the `test_impl` folder to clearly indicate the production usage of both types.
2. In the `TestingTree` definition, use the `TREE_HEIGHT` constant instead of the `TESTING_TREE_HEIGHT` constant.

Large function definitions are difficult to maintain

The single function definition `verify_and_apply_batch`, located in the `simple_growable_storage.rs` file, occupies 600 lines of code, which poses challenges for review and refactoring.

Besides merely size of the function, the definition also includes several bad practices, e.g. panicking instead of proper error handling, one-letter variable names and using tuples instead of dedicated structures. However, some of the code quality issues deserve to be reported separately.

Complex handling of iterators instead of relying on the standard `chunks` operation

While iterating over the `current_hashes_buffer` vector, overlapping windows are used together with custom skipping of every second window. This behaviour can be implemented using the standard iterator operation `chunks`:

basic_system/src/system_implementation/flat_storage_model/simple_growable_storage.rs

```
fn verify_and_apply_batch
692     for (idx, [a, b]) in current_hashes_buffer.array_windows::<2>().enumerate() {
693         if next_merged {
694             next_merged = false;
695             continue;
696         }
```

```
746     next_hashes_buffer.push((merged_index, a.1, read_hash, write_hash));
747     next_merged = true;
```

Using the `chunks` operation would also render the auxiliary function `can_merge` redundant, since it only determines that two leaves belong to the same chunk by validating the `a = b + 1` equation:

```
628     fn can_merge(pair: (u64, u64)) -> bool {
629         let (a, b) = pair;
630         debug_assert_ne!(a, b);
631         a & !1 == b & !1
```

Assertion on the number of writes can be strengthened

After verifying the state changes in a transaction batch, if the root hash remains unchanged, it is asserted that `num_new_writes` is zero. However, the `num_new_writes` variable represents only “non-existing writes.” The assertion should instead target the `num_total_writes` variable, as writes to already existing cells must result in a changed root hash, too.

```
basic_system/src/system_implementation/flat_storage_model/simple_growable_storage.rs
fn verify_and_apply_batch
775   if let Some(new_root) = current_hashes_buffer[0].3 {
776       if num_total_writes > 0 {
777           assert!(
778               new_root != self.root,
779               "hash collision on state root with non-zero number of writes"
780           );
781       }
782       self.root = new_root;
783   } else {
784       assert_eq!(num_new_writes, 0);
785       // root should not change in such case
786   }
```

Miscellaneous code quality concerns

```
basic_system/src/system_implementation/flat_storage_model/simple_growable_storage.rs
230   let num_new_writes = source
231       .clone()
232       .filter(|(_, v)| v.current_value != v.initial_value && v.is_new_storage_slot)
```

Instead of cloning the **`source`** iterator and filtering out reads again, it would be more efficient to clone the **`writes_iter`** which contains only writes.

```
basic_system/src/system_implementation/flat_storage_model/simple_growable_storage.rs
772   // if we have new root - use it
773   if let Some(new_root) = current_hashes_buffer[0].3 {
```

In multiple locations, getters **`.2`** and **`.3`** are used. In mission-critical code such should be replaced with a named field to improve maintainability.

*The client has addressed the first point of the issue, specifically the production constant **`TREE_HEIGHT`** is used now, instead of **`TESTING_TREE_HEIGHT`**. However the **`InMemoryTree`** structure is still located in **`forward_system/src/run/test_impl/tree.rs`** which can be confused with testing code. The type is also parameterized with **`RANDOMIZED_TREE`** flag, which is passed to **`FlatStorageBacking`** as-is. It can be considered to remove the the flag and always pass **`false`** for non-testing usages. Other points of the issue has not been addressed yet.*

12 Missing validation of token transfer during tokens burning

Severity: Low Commit: 5e69d44 Impact: Medium Likelihood: Low Status: Notified

The implementation of withdrawals from L2 to L1 relies on the implicit invariant that **`nominal_token_value`** has been transferred from the **`caller`** to **`L2_BASE_TOKEN_ADDRESS`**. Otherwise, any user could crash the node by attempting to withdraw non-existing tokens:

```
system_hooks/src/l2_base_token.rs
fn l2_base_token_hook_inner
// Burn nominal_token_value
match system.io.update_account_nominal_token_balance(
    ExecutionEnvironmentType::parse_ee_version_byte(caller_ee)
        .map_err(SystemError::LeafDefect)?,
    resources,
    &L2_BASE_TOKEN_ADDRESS,
    &nominal_token_value,
    true,
) {
    Ok(_) => Ok(()),
    Err(UpdateQueryError::NumericBoundsError) => Err(SystemError::LeafDefect(
        internal_error!("L2 base token must have withdrawal amount"),
```

The prerequisite token transfer is ensured earlier in the withdrawal flow, by another function **`external_call_before_vm`**, which invokes the **`transfer_nominal_token_value_inner`** function. This is happening right before the **`handle_requested_external_call_to_special_address_space`** function passes execution to **`try_intercept`**, which calls the **`l2_base_token_hook`** handler.

```
system_hooks/src/l2_base_token.rs
fn external_call_before_vm
if let Some(TransferInfo { value, target }) = transfer_to_perform {
    match actual_resources_to_pass.with_infinite_ergs(|inf_resources| {
        self.system.io.transfer_nominal_token_value(
            ExecutionEnvironmentType::NoEE,
            inf_resources,
            &call_request.caller,
            &target,
            &value,
```

However, when **`transfer_to_perform`** is **`None`**, no action is taken, despite certain call modifiers potentially setting it to this value. System hooks currently treat such modifiers as errors, thereby preventing **`None`** values. The concern here is that this invariant is not explicitly enforced in the code.

Recommendation

While the analysis revealed that this scenario is safe, it is still recommended to validate consistency between the `external_call_before_vm` and `l2_base_token_hook` functions:

- Explicitly handle the `None` case for the `transfer_to_perform` parameter.
- Additionally, verify that the transaction has adjusted the balances of the `caller` and the `L2_BASE_TOKEN_ADDRESS` by the burned amount.

Matter Labs has internally verified the safety of L2 withdrawals flow and considers the extra validations redundant.

13 Missing validations during system upgrades

Severity: Low Commit: 5e69d44 Impact: Medium Likelihood: Low Status: Notified

The following two issues are reported with "Low" severity since only the governance or operator could cause them, most probably as a consequence of deployment mistake. Given the importance of system upgrade operation, extra checks are advisable.

Observable bytecode hash is not verified

During system upgrades, the function `set_bytecode_details` is reached from the `contract_deployer_hook` function. The function utilizes parameters `observable_bytecode_hash` and `observable_bytecode_len` to set the corresponding fields of the contract's account:

basic_system/src/system_implementation/flat_storage_model/account_cache.rs

```
pub fn set_bytecode_details
795   observable_bytecode_hash: Bytes32,
796   observable_bytecode_len: u32,
```

```
876   account_data.update(|cache_record| {
877       cache_record.update(|v, m| {
878           v.observable_bytecode_hash = observable_bytecode_hash;
879           v.observable_bytecode_len = observable_bytecode_len;
```

These parameters receive values retrieved in the **`contract_deployer_hook_inner`** function as the result of parsing the calldata. The **`observable_bytecode_len`** parameter corresponds to the exact length of the byte code received, since the same value is used in the **`expose_preimage`** function, where the buffer is truncated at this length, and hash of the buffer would not match if the length was incorrect.

However, the **`observable_bytecode_hash`** is not verified against the byte code:

```
system_hooks/src/contract_deployer.rs
let observable_bytecode_hash =
168         Bytes32::from_array(calldata[96..128].try_into().expect("Always valid"));
```

```
182   system.set_bytecode_details(
183       resources,
184       &address,
185       ExecutionEnvironmentType::EVM,
186       bytecode_hash,
187       bytecode_length,
188       0,
189       observable_bytecode_hash,
190       bytecode_length,
191   )?;
```

Note that the **`bytecode_hash`** is a value computed using the *Blake2s* hash function and is verified in the **`expose_preimage`** function, while the **`observable_bytecode_hash`** should be a *Keccak256* hash but passed into the function **`set_bytecode_details`** without verification. As a consequence, potential governance mistakes can cause inconcistency in the final account data.

Validation of EIP-3541

```
system_hooks/src/contract_deployer.rs
fn contract_deployer_hook_inner
179   // Also EIP-3541(reject code starting with 0xEF) should be validated by governance.
```

Although it is required to be validated by the L1 governance, the extra check would not impose significant performance penalty. Given the rareness and importance of such operation as system upgrade, the extra check is advisable.

Additionally, the validations required by governance should be documented.

*Matter Labs prefers to keep the ability to pass the **`observable_bytecode_hash`** as-is, for generality. They are considering the addition of EIP-3541 validation. Both issues are planned to be documented for appropriate usage by the on-chain governance.*

14 Potential extra validations

Severity: Low Impact: Medium Likelihood: Low Status: Notified

Validate receiver when reading the EVM deployment flag

```
basic_bootloader/src/bootloader/account_models/eoa.rs
fn charge_additional_intrinsic_gas
485   let to = transaction.to.read();
486   let is_deployment =
487       !transaction.reserved[1].read().is_zero() || to == SPECIAL_ADDRESS_TO_WASM_DEPLOY;
```

Validate that **`transaction.to == null`** when the EVM deployment transaction flag is set. Auxiliary predicate **`isDeployment`** could be defined that examines value of the **`transaction.reserved[1]`** flag and also validates the **`transaction.to`** field.

```
basic_bootloader/src/bootloader/transaction/mod.rs
pub struct ZkSyncTransaction
 65   /// Now `reserved[0]` is used as a flag to distinguish EIP-155(with chain id) legacy
transactions.
 66   /// `reserved[1]` is used as EVM deployment transaction flag(`to` == null in such case).
 67   pub reserved: [ParsedValue<U256>; 4],
```

Unsafe function is called without validating preconditions

The function **`grow_heap`** includes call to the **`offset_from_unsigned`** function:

```
basic_system/src/system_implementation/memory/basic_memory.rs
fn grow_heap
152   let current_heap_capacity = unsafe {
153       self.heap_buffer.len()
154           - self
155               .current_heap_start
156               .offset_from_unsigned(self.heap_buffer.cast())
157   };
```

However, the function `offset_from_unsigned` is unsafe and requires the following condition to be validated before the call:

> *"The distance between the pointers must be non-negative (self >= origin)"*

In this particular case, this means that `self.current_heap_start` must be ensured to be not lesser than `self.heap_buffer`.

Otherwise, the call can lead to an Unsafe Behaviour.

15 Unchecked `base_fee_per_gas` downcast

Severity: Low Commit: 96d9d37 Impact: Medium Likelihood: Low Status: Fixed

The bootloader performs an unchecked downcast of a block-level parameter `base_fee_per_gas` from type `U256` to type `u64` using `try_into().unwrap()`:

```
basic_bootloader/src/bootloader/mod.rs
pub fn run_prepared
341   let block_header = BlockHeader::new(
```

```
350       base_fee_per_gas.try_into().unwrap(),
```

The oracle can set arbitrary values without validation, as confirmed by the comment:

```
basic_bootloader/src/bootloader/block_header.rs
117   pub fn new
118   // currently operator can set any base_fee_per_gas, in practice it's usually constant
119   base_fee_per_gas,
```

If the oracle supplies a `base_fee_per_gas` value exceeding `u64::MAX`, the bootloader will panic and halt all transaction processing. While economically impractical under normal conditions, this creates a real DoS vector if the oracle is misconfigured or compromised.

The value `base_fee_per_gas` is initialized using the call `system.get_eip1559_basefee()` that extracts the field `eip1559_basefee` from the structure `BlockMetadataFromOracle`.

A panic in this context halts the network until the oracle issue is resolved.

Recommendation

Handle gracefully the case of `base_fee_per_gas` value exceeding `u64::MAX` by returning an error of type `TxError::Internal`.

This issue has been resolved completely.

16 Unsafe assumption that `pubdata` cannot decrease after validation

Severity: Low Commit: 96d9d37 Impact: Medium Likelihood: Low Status: Addressed

The function `get_resources_to_charge_for_pubdata` computes a fair amount of native resources to charge for the pubdata used by a transaction. This function iterates through all state elements, comparing initial and final values to calculate the state difference and determine the amount of resources to charge, proportional to this difference. It is parameterized by an optional `base_pubdata` parameter. When `base_pubdata` is `None`, the result of the state iteration is returned as is.

However, when a value for `base_pubdata` is provided, it is used to adjust the result of the state iteration:

basic_bootloader/src/bootloader/gas_helpers.rs

```
pub fn get_resources_to_charge_for_pubdata
138   let current_pubdata_spent = system.net_pubdata_used()? - base_pubdata.unwrap_or(0);
139   let native_per_pubdata = u256_to_u64_saturated(&native_per_pubdata);
140   let native = current_pubdata_spent
141       .checked_mul(native_per_pubdata)
142       .ok_or(InternalError("cps*epp"))?;
143   let native = <S::Resources as zk_ee::system::Resources>::Native::from_computational(native);
144   Ok((current_pubdata_spent, S::Resources::from_native(native)))
```

Since unchecked arithmetic is employed, there is a potential risk of underflow.

Initially, the function is utilized by **`transaction_validation`** to compute **`validation_pubdata`**. This call is safe because no value is passed as **`base_pubdata`** yet:

```
basic_bootloader/src/bootloader/process_transaction.rs
fn transaction_validation
704   let (validation_pubdata, to_charge_for_pubdata) =
705       get_resources_to_charge_for_pubdata(system, native_per_pubdata, None)?;
```

The second call, which is more concerning, occurs during the gas refund process and passes the result of the first call, referred to as **`validation_pubdata`**, as the value for the **`base_pubdata`** parameter:

```
basic_bootloader/src/bootloader/process_transaction.rs
fn refund_transaction
964   let (_pubdata_spent, to_charge_for_pubdata) = get_resources_to_charge_for_pubdata(
965       system,
966       native_per_pubdata,
967       Some(validation_pubdata),
```

Severity of the issue

The impact of the underflow is that the **`current_pubdata_spent`** variable receives a value close to **`u64::MAX`**, leading to native resource exhaustion and an unexplained transaction reversal for the user.

However, triggering the underflow is unlikely with the currently implemented execution environments. The **`pubdata`** value post-execution does not decrease compared to its post-validation value. Yet, this assumption might not hold in other execution environments that could charge a conservative estimate of pubdata during validation, intending to refund any excess later.

Recommendation

To ensure a future-proof design, consider two approaches:

- Implement checked arithmetic to prevent underflow and reject scenarios where the post-execution pubdata value is less than the post-validation value. Additionally, document this behaviour as a requirement for future execution environments.
- Alternatively, explicitly permit a reduction in pubdata during later processing stages, and incorporate this difference into the gas refund calculation.

Matter Labs has addressed this issue by implementing saturated arithmetics. This potentially introduces another issue if any of execution environments charges excess pubdata prior to executing transactions. Refunding the excess is not implemented yet. Overall, the current usage is safe.

17 Decreased performance due to redundant `pubdata` calculation

Severity: Low Commit: 96d9d37 Impact: Low Likelihood: Medium Status: Fixed

The function **`get_resources_to_charge_for_pubdata`** serves as the entry point for computationally intensive calculations defined by the function **`net_pubdata_used`** and by functions implemented in several types of storage, all named **`calculate_pubdata_used_by_tx`**. It iterates through all versions of state elements and calculates the amount of change created by the transaction.

```
basic_bootloader/src/bootloader/gas_helpers.rs
pub fn get_resources_to_charge_for_pubdata
138   let current_pubdata_spent = system.net_pubdata_used()? - base_pubdata.unwrap_or(0);
```

```
basic_system/src/system_implementation/system/io_subsystem.rs
fn net_pubdata_used
357   Ok(self.storage.pubdata_used_by_tx() as u64
358       + self.logs_storage.calculate_pubdata_used_by_tx()? as u64)
359   }
```

Since the **`pubdata`** calculation involves multiple iterations through a potentially large set of changes, it negatively affects the performance of the node and should be used only when absolutely necessary. However, besides the lightweight **`pubdata`** calculation during the validation stage and the proper **`pubdata`** calculation during the refund stage, the **`get_resources_to_charge_for_pubdata`** function is also called during transaction execution:

```
basic_bootloader/src/bootloader/process_transaction.rs
fn transaction_execution
751   if !check_enough_resources_for_pubdata(
752       system,
753       native_per_pubdata,
754       resources,
755       Some(validation_pubdata),
756   )? {
757       let _ = system
758           .get_logger()
759           .write_fmt(format_args!("Not enough gas for pubdata after execution\n"));
760       Ok(execution_result.reverted())
```

```
basic_bootloader/src/bootloader/gas_helpers.rs
pub fn check_enough_resources_for_pubdata
160       let (_, resources_for_pubdata) =
161           get_resources_to_charge_for_pubdata(system, native_per_pubdata, base_pubdata)?;
```

The `pubdata` calculation is performed at the end of the actual transaction execution, so the result does not change until the refund stage:

```
basic_bootloader/src/bootloader/process_transaction.rs
fn refund_transaction
964   let (_pubdata_spent, to_charge_for_pubdata) = get_resources_to_charge_for_pubdata(
965       system,
966       native_per_pubdata,
967       Some(validation_pubdata),
968   )?;
```

Recommendation

Avoid excessive calculation by passing the result of the `pubdata` calculation from the execution stage to the refund stage. Alternatively, a more sophisticated but universal memoization technique could be implemented.

This issue has been resolved completely.

18 Double resource accounting is a deviation from EVM semantics

Severity: Low Commit: 96d9d37 Impact: Low Likelihood: Medium Status: Addressed

ZKsync OS implements double-accounting for computation resources. One accounting subsystem tracks, charges and refunds “ergs” which are trivially translated into “gas”, so this accounting subsystem ensures full compliance with EVM semantics when the node executes transactions. Another subsystem accounts “native” resources which represent cycles spent in the proving mode.

Each subsystem declares its own error type for situations when the node runs out of the corresponding resource:

- `OutOfErgs`, representing the "Out Of Gas" condition in the EVM
- `OutOfNativeResources`, representing the condition when the proving or publishing would exceed the allowed amount of computation resources

However, since the `OutOfNativeResources` can be reported during transaction validation or execution, this presents deviation from pure EVM semantics:

```
system_hooks/src/precompiles.rs
pub fn pure_system_function_hook_impl
131   Err(SystemFunctionError::System(SystemError::OutOfErgs))
132   | Err(SystemFunctionError::InvalidInput) => {
133       let _ = system
134           .get_logger()
135           .write_fmt(format_args!("Out of gas during system hook\n"));
136       resources.exhaust_ergs();
137       Ok(make_error_return_state(system, resources))
138   }
139   Err(SystemFunctionError::System(SystemError::OutOfNativeResources)) => {
140       Err(FatalError::OutOfNativeResources)
141   }
```

As a consequence, a transaction that executes correctly in EVM, without causing the "Out Of Gas" condition, can still revert or be rejected due to excessive native resources cost. This is expected, according to the docs:

> *"If a transaction execution runs out of native resources, the entire transaction is reverted. If the same happens during transaction validation, the transaction is considered invalid."*

However, this situation could look like a spontaneous Denial-of-Service to the user.

Similar issues have been observed in other locations:

- `basic_bootloader/src/bootloader/account_models/eoa.rs`, line 361
- `basic_bootloader/src/bootloader/account_models/eoa.rs`, line 503
- `basic_bootloader/src/bootloader/account_models/eoa.rs`, line 556

*This issue can be a challenge to remediate given current design of the system. It is also mentioned briefly in the **developer's docs**. However, it should also be documented on the official website and included into the list of deviations from the standard EVM semantics.*

19 Inconsistent developer documentation

Severity: Low Commit: 96d9d37 Impact: Low Likelihood: Medium Status: Addressed

Opcode `DIFFICULTY` is not consistent with the documentation

The **`DIFFICULTY`**/**`PREVRANDAO`** opcode in the EVM interpreter returns zero instead of the constant value **`2500000000000000`** specified in the documentation.

In the **`difficulty`** handler, the **`U256::ZERO`** value is pushed onto the stack:

```
evm_interpreter/src/instructions/environment.rs
32    pub fn difficulty(&mut self) -> InstructionResult {
33        self.spend_gas_and_native(gas_constants::BASE, DIFFICULTY_NATIVE_COST)?;
34        self.push_values(&[U256::ZERO])?;
35        Ok(())
36    }
```

This deviation from documented behavior could cause compatibility issues for contracts that check specific difficulty values or use it in non-critical calculations.

Documentation incorrectly lists `p256Verify` as precompile at address `0x100`

The documentation lists **`p256Verify`** as a precompile at address **`0x100`**. However, the auxiliary function **`is_precompile`** only recognizes addresses from 1 to 10 as the precompile addresses.

```
evm_interpreter/src/utils.rs
415   /// Helper to check if an address is an ethereum precompile
416   #[inline(always)]
417   pub fn is_precompile(address: &B160) -> bool {
418       let highest_precompile_address = 10;
419       let limbs = address.as_limbs();
420       if limbs[1] != 0u64 || limbs[2] != 0u64 {
421           return false;
422       }
423       limbs[0] > 0 && limbs[0] <= highest_precompile_address
424   }
```

The `p256Verify` function is actually implemented as a system function rather than an EVM precompile, making the documentation incorrect.

This discrepancy could mislead developers expecting standard precompile behavior at address `0x100`.

*Matter Labs responses that the mentioned docs are for the Era version of the protocol. The `DIFFICULTY` opcode is documented correctly in the **developer's docs**. However, the official website should contain the up-to-date documentation.*

20 Incorrect opcode name in debug output

Severity: Low Commit: 96d9d37 Impact: Low Likelihood: Medium Status: Fixed

The opcode `TSTORE` (transient storage) is defined as `0x5d` as it is observed in the `opcodes.rs` file:

```
evm_interpreter/src/opcodes.rs
52    pub const TSTORE: u8 = 0x5d;
```

However, the `OPCODE_JUMPMAP` array, defined in the same file and used for debug string mapping, contains a typo where the opcode `0x5d` is incorrectly labeled as `"STORE"`.

```
evm_interpreter/src/opcodes.rs
pub const OPCODE_JUMPMAP
283    /* 0x5d */ Some("STORE"),
```

This discrepancy only affects debug output and logging — the actual opcode execution is unaffected since the interpreter uses numeric constants rather than their text labels.

Recommendation

Update the string mapping to correctly identify the opcode as `"TSTORE"` for accurate debug output.

This issue has been resolved completely.

21 Code duplicates

Severity: QA Impact: Low Likelihood: Low Status: Notified

Auxiliary function `nb_rounds`

```
basic_system/src/system_functions/sha256.rs
28    fn nb_rounds(len: usize) -> u64 {
29        let full_chunks = len / SHA256_CHUNK_SIZE;
30        let tail = len % SHA256_CHUNK_SIZE;
31        let num_rounds: u64 = full_chunks as u64;
32        if tail <= 55 {
33            num_rounds + 1
34        } else {
35            num_rounds + 2
36        }
37    }
```

Implementations of both **`sha256`** and **`ripemd160`** system functions include such auxiliary function.

Implementations of the `Drop` trait

The **`ElementPool`** type implements the **`Drop`** trait:

```
zk_ee/src/common_structs/history_map/element_pool.rs
17    impl<V, A: Allocator + Clone> Drop for ElementPool<V, A> {
18        fn drop(&mut self) {
19            if let Some(head) = self.head {
20                let mut elem = unsafe { Box::from_raw_in(head.as_ptr(), self.alloc.clone()) };
21
22                while let Some(n) = elem.previous.take() {
23                    let n = unsafe { Box::from_raw_in(n.as_ptr(), self.alloc.clone()) };
24
25                    elem = n;
26                } // `n` is dropped here.
27            } // Last elem is dropped here.
28        }
29    }
```

The same code snippet is duplicated for the **`Drop`** trait implementation of the **`ElementWithHistory`** type, within lines 23-34 of the **`zk_ee/src/common_structs/history_map/element_with_history.rs`** file. This code can be extracted into an auxiliary function.

Module `modexp` and the buffer-fill logic

The following code copies the **`base`** value:

```
basic_system/src/system_functions/modexp.rs
131   let mut input_it = input.iter();
132   let mut base = Vec::try_with_capacity_in(base_len, allocator.clone())
133     .map_err(|_| SystemError::Internal(InternalError("alloc")))?;
134   base.resize(base_len, 0);
135   for (dst, src) in base.iter_mut().zip(&mut input_it) {
136     *dst = *src;
137   }
```

Exactly same code blocks are used in the same file, for **`exponent`** and **`modulus`** values:

- lines 139-144
- lines 146-151

Derivation of `is_static` flag

```
system_hooks/src/l1_messenger.rs
pub fn l1_messenger_hook
 43   let mut error = false;
 44   // There is no "payable" methods
 45   error |= nominal_token_value != U256::ZERO;
 46   let mut is_static = false;
 47   match modifier {
 48       CallModifier::Constructor => {
 49           return Err(
 50               internal_error!("L1 messenger hook called with constructor modifier").into(),
 51           )
 52       }
 53       CallModifier::Delegate
 54       | CallModifier::DelegateStatic
 55       | CallModifier::EVMCallcode
 56       | CallModifier::EVMCallcodeStatic => {
 57           error = true;
 58       }
 59       CallModifier::Static | CallModifier::ZKVMSystemStatic => {
 60           is_static = true;
 61       }
 62       _ => {}
 63   }
 64
 65   if error {
 66       return Ok((make_error_return_state(available_resources), return_memory));
 67   }
```

The same or very similar code snippets can also be observed in:

- **`system_hooks/src/l2_base_token.rs`** (lines 38-60)
- **`system_hooks/src/contract_deployer.rs`** (lines 38-61)

Burning tokens from `L2_BASE_TOKEN_ADDRESS`

While there is no immediate impact, the following code snippets are important enough to reduce chance of a mistake during refactoring.

```
system_hooks/src/l2_base_token.rs
fn l2_base_token_hook_inner
146   // Burn nominal_token_value
147   match system.io.update_account_nominal_token_balance(
148       ExecutionEnvironmentType::parse_ee_version_byte(caller_ee)
149           .map_err(SystemError::LeafDefect)?,
150       resources,
151       &L2_BASE_TOKEN_ADDRESS,
152       &nominal_token_value,
153       true,
154   ) {
155       Ok(_) => Ok(()),
156
157       Err(UpdateQueryError::NumericBoundsError) => Err(SystemError::LeafDefect(
158           internal_error!("L2 base token must have withdrawal amount"),
159       )),
160       Err(UpdateQueryError::System(e)) => Err(e),
161   }?;
```

The same code snippet can be also observed within lines 246-262 of the same file.

Bytecode hash validation

The bytecode hash is recomputed and validated against the expected value using this code:

```
basic_system/src/system_implementation/flat_storage_model/preimage_cache.rs
fn expose_preimage
105   use crypto::blake2s::Blake2s256;
106   use crypto::MiniDigest;
107   let digest = Blake2s256::digest(buffered.as_slice());
108   let mut result = Bytes32::uninit();
109   let recomputed_hash = unsafe {
110       result
111           .assume_init_mut()
112           .as_u8_array_mut()
113           .copy_from_slice(digest.as_slice());
114       result.assume_init()
```

```
115   };
116
117   if recomputed_hash != *hash {
118       return Err(internal_error!("Account hash mismatch").into());
119   }
```

This code is duplicated without any difference for `PreimageType::AccountData` and `PreimageType::Bytecode` branches of the `match` expression. The whole `match` expression is duplicated again in two branches of the `if PROOF_ENV` statement.

Validation of zero bytes in Ethereum addresses

This code preforms the validation of the 12 highest bytes in Ethereum addresses:

```
system_hooks/src/contract_deployer.rs
fn contract_deployer_hook_inner
147   // check that first 12 bytes in address encoding are zero
148   if calldata[0..12].iter().any(|byte| *byte != 0) {
149       return Ok(Err(
150           "Contract deployer failure: setBytecodeDetailsEVM called with invalid calldata",
151       ));
152   }
```

Almost exact duplicate is located in `system_hooks/src/l2_base_token.rs:170-175`.

22 Error handling issues

Severity: QA Impact: Low Likelihood: Low Status: Notified

The codebase heavily relies on panics instead of returning error values.

While panics may be appropriate in certain scenarios, it is usually a trade-off:

- *Missed error handling*: panics are not subject to type-checking, so it is easier to overlook error-handling mechanisms.
- *Unrecoverable failures*: if a panic signals a recoverable error and is not properly handled, the program may terminate unnecessarily.
- *Reduced readability and debugging complexity*: codebases that rely on panics are harder to follow and debug, increasing maintenance overhead.

- *Performance overhead*: handling the panic requires unwinding the stack, which decreases program performance.

Consequently, relying on panics as the primary error-handling mechanism increases the difficulty of ensuring codebase safety and maintainability. It is better to approach error handling with meaningful error messages rather than using panics. This approach improves code maintainability, debugging efficiency, and overall developer experience.

Examples in the EVM interpreter

The EVM interpreter contains multiple instances of **`panic!`** and **`assert!`** statements, instead of returning proper errors. Although these statements represent internal invariants that should not be violated in correct implementations, relying on panics reduces code robustness and renders the system vulnerable to future changes.

```
evm_interpreter/src/interpreter.rs
28    if let Some(call) = external_call {
29        assert!(exit_code == ExitCode::ExternalCall);
```

```
evm_interpreter/src/ee_trait_impl.rs
119   if scratch_space_len != 0 || decommitted_bytecode.len() != bytecode_len as usize {
120       panic!("invalid bytecode supplied, expected padding");
121   }
```

```
evm_interpreter/src/ee_trait_impl.rs
182   a => {
183       panic!("modifier {:?} is not expected", a);
184   }
```

```
evm_interpreter/src/u256.rs
pub(crate) fn log2floor
28        assert!(value != &U256::ZERO);
```

These panic conditions complicate maintenance. In contrast, other errors in the same files are handled appropriately, returning **`FatalError::Internal`**.

Examples in the rest of the system

basic_bootloader/src/bootloader/runner.rs
```
pub fn run_till_completion
 37    assert!(callstack.is_empty());
```

basic_system/src/system_implementation/system/io_subsystem.rs
```
fn finish
477   // TODO (EVM-989): read only state commitment
478   let fsm_state =
479       <BasicIOImplementerFSM::<FlatStorageCommitment<TREE_HEIGHT>> as
UsizeDeserializable>::from_iter(&mut initialization_iterator).unwrap();
```

basic_system/src/system_implementation/flat_storage_model/simple_growable_storage.rs
```
pub fn set_next
180   assert!(self.next.is_none());
181   self.next = Some(value);
```

system_hooks/src/lib.rs
```
pub fn add_hook
 97    let existing = self.inner.insert(for_address_low, hook);
 98    // TODO: internal error?
 99    assert!(existing.is_none());
```

system_hooks/src/precompiles.rs
```
fn new
 33    let buffer = system.memory.empty_managed_region();
 34    let buffer = system
 35        .memory
 36        .grow_heap(buffer, Self::INITIAL_LEN)
 37        .expect("must grow buffer for precompiles")
 38        .expect("must grow buffer for precompiles");
```

system_hooks/src/precompiles.rs
```
fn extend
 56    self.buffer = self
 57        .system
 58        .memory
 59        .grow_heap(buffer, new_len)
 60        .expect("must grow buffer for precompiles")
 61        .expect("must grow buffer for precompiles");
 62    }
```

```
system_hooks/src/precompiles.rs
pub fn pure_system_function_hook_impl
117   system
118       .memory
119       .copy_into_return_memory(&buffer[..offset])
120       .expect("must copy into returndata")
121       .take_slice(0..offset)
```

Recommendation

Unless when it is supposed to be an unbreakable invariant, replace all `panic!` and `assert!` statements with proper error handling by returning typed error messages. This would prevent unnecessary node crashes.

23 Magic numbers

Severity: QA Impact: Low Likelihood: Low Status: Notified

In system hooks implementation

Numerous numeric literals are used without proper documentation in files `contract_deployer.rs`, `l1_messenger.rs` and `l2_base_token.rs`. Some of them reflect fundamental properties of Solidity ABI, other are specific to calldata and messages layout and are re-used in similar parts of the codebase.

```
system_hooks/src/contract_deployer.rs
fn contract_deployer_hook_inner
140   calldata = &calldata[4..];
141   if calldata.len() < 128 {
```

```
148   if calldata[0..12].iter().any(|byte| *byte != 0) {
```

```
153   let address =
154       B160::try_from_be_slice(&calldata[12..32]).ok_or(SystemError::LeafDefect(
```

```
159   let bytecode_hash =
160       Bytes32::from_array(calldata[32..64].try_into().expect("Always valid"));
```

```
169   let observable_bytecode_hash =
170       Bytes32::from_array(calldata[96..128].try_into().expect("Always valid"));
```

```
system_hooks/src/l2_base_token.rs
fn l2_base_token_hook_inner
143   // following solidity abi for withdraw(address)
144   if calldata.len() < 36 {
```

The number `36` means the sum of `32` and `4` but it should be better to adopt the "self-documented code" and explicitly compose the expression from named constants.

```
169   message[0..4].copy_from_slice(FINALIZE_ETH_WITHDRAWAL_SELECTOR);
170   // check that first 12 bytes in address encoding are zero
171   if calldata[4..4 + 12].iter().any(|byte| *byte != 0) {
```

```
195   // following solidity abi for withdrawWithMessage(address,bytes)
196   if calldata.len() < 68 {
```

And multiple other similar issues.

Length of word

```
system_hooks/src/precompiles.rs
fn execute
161   cycle_marker::wrap_with_resources!("id", resources, {
162       let cost_ergs =
163           ID_STATIC_COST_ERGS + ID_WORD_COST_ERGS.times((src.len() as u64).div_ceil(32));
```

While the length of word in EVM is always `32`, it is still worthwhile to declare an explicit constant for better maintainability. This is relevant to implementations of system functions `identity`, `keccak256`, `ripemd160` and `sha256`.

Modexp system function

```
basic_system/src/system_functions/modexp.rs
180   let ic = if exp_size <= 32 && exp_highp.is_zero() {
181       0
182   } else if exp_size <= 32 {
183       exp_highp.bit_len() as u64 - 1
184   } else {
185       8u64.checked_mul(exp_size - 32)
```

Similarly, in the implementation of the `modexp` system function, the number `32` is used on lines 112, 120, 121, 180, 182, 185.

basic_system/src/system_functions/modexp.rs
```
pub fn ergs_cost
197   let gas = core::cmp::max(200, computed_gas);
198   let ergs = gas
199       .checked_mul(ERGS_PER_GAS)
200       .ok_or(SystemError::OutOfErgs)?;
201   Ok(Ergs(ergs))
```

Additionally, the number `200` which acts as a minimum base price to charge on EVM could be declared as dedicated constant, too.

Access list gas costs

Gas costs `1900` for storage and `2400` for accounts should be declared as named constants:

basic_system/src/system_implementation/system/mod.rs
```
fn charge_cold_storage_read_extra
 68   if is_access_list {
 69       Ergs(1900 * ERGS_PER_GAS)
```

basic_system/src/system_implementation/flat_storage_model/account_cache.rs
```
fn materialize_element
129   if is_access_list {
130       resources.charge(&R::from_ergs(Ergs(2400 * ERGS_PER_GAS)))?
```

```
202   if is_access_list {
203       resources.charge(&R::from_ergs(Ergs(2400 * ERGS_PER_GAS)))?
```

In pubdata calculation

The number `32` is used in the `calculate_pubdata_used_by_tx` function without proper documentation:

basic_system/src/system_implementation/flat_storage_model/storage_cache.rs
```
pub fn calculate_pubdata_used_by_tx
628   if initial_value != current_value {
629       // TODO(EVM-1074): use tree index instead of key for repeated writes
630       pubdata_used += 32; // key
```

Undocumented callstack limitation

The number `1024` is used twice in the deployment handler:

```
basic_bootloader/src/bootloader/runner.rs
fn handle_requested_deployment
473   if self.callstack_height > 1024 {
474       return Ok(Some(CallResult::Failed {
```

```
660   if self.callstack_height > 1024 {
661       resources_for_deployer.reclaim(launch_params.external_call.available_resources);
```

There is the corresponding constant declared:

```
zk_ee/src/system/mod.rs
 25   pub const MAX_GLOBAL_CALLS_STACK_DEPTH: usize = 1024; // even though we do not have to
formally limit it,
 26                                                         // for all practical purposes (63/64) ^
1024 is 10^-7, and it's unlikely that one can create any new frame
 27                                                         // with such remaining resources
```

However, the constant `MAX_GLOBAL_CALLS_STACK_DEPTH` is not used.

Undocumented length of blocks history

```
zk_ee/src/system/mod.rs
pub fn get_blockhash
102   let current_block_number = self.metadata.block_level_metadata.block_number;
103   if block_number >= current_block_number
104       || block_number < current_block_number.saturating_sub(256)
```

The number `256` reflects the maximum number of hashes to consider and is chosen arbitrary.

Undocumented border of precompile addresses

```
evm_interpreter/src/utils.rs
pub fn is_precompile
165   let highest_precompile_address = 10;
```

The number `10` means the border between precompile addresses and regular addresses. It should be extracted as a named constant.

Keccak256 gas cost

```
basic_bootloader/src/bootloader/transaction/mod.rs
fn eip712_tx_calculate_signed_hash
963   let domain_separator = Self::domain_hash_struct(chain_id, resources)?;
964   let hs = self.hash_struct(resources)?;
965   charge_keccak(2 + 2 * U256::BYTES, resources)?;
```

The number `2` should represent a computation price for Keccak256 in RISC-V cycles. However, this semantics should be properly documented by using named constants.

24 Misleading docs and messages

Severity: QA Impact: Low Likelihood: Low Status: Notified

The `identity` precompile does not stay in the caller’s frame

```
system_hooks/src/precompiles.rs
74    /// It parses call request, calls system function, and creates execution result.
75    ///
76    /// NOTE: "pure" here means that we do not expect to trigger any state changes (and calling
with static flag is ok),
77    /// so for all the purposes we remain in the callee frame in terms of memory for efficiency
```

```
98    // NOTE: we did NOT start a frame here, so we are in the caller frame in terms of memory, and
must be extra careful
99    // here on how we will make returndata
```

In the description of the `identity` system function, it is stated that the execution does not create additional frame and stays in the “caller’s” frame. However, besides the typo on the line 77, this note also seems to be incorrect, because there is the frame creation further in the flow:

```
106   // cheat
107   let snapshot = system.memory.start_memory_frame();
```

Miscellaneous outdated comments

basic_system/src/system_functions/modexp.rs
```
24    /// Returns `InvalidInput` error if `base_len` > usize max value
25    /// or `mod_len` > usize max value
26    /// or (`exp_len` > usize max value and `base_len` != 0 and `mod_len` != 0).
```

If both **`base_len != 0`** and **`mod_len != 0`** are zero, **`Ok(())`** is returned, otherwise the execution flow proceeds to **`exp_len`** validation, i.e. if at least one is non-zero, not necessarily both.

basic_system/src/system_functions/bn254_ecadd.rs
```
18    /// If the input size is less than expected - it will be padded with zeroes.
19    /// If the input size is greater - redundant bytes will be ignored.
20    ///
21    /// If output len less than needed(64) returns `InternalError`.
```

Technically, the output is always a vector of 64 bytes, although there can be zero bytes. The **`InternalError`** error is not utilized in this file.

basic_system/src/system_functions/p256_verify.rs
```
13    /// In case of invalid input `Ok(0)` will be returned and resources will be charged.
14    ///
15    /// If dst len less than needed(1) returns `InternalError`.
```

While the docs mention the **`Ok(0)`** value being returned in case of an invalid input, in fact, the **`Ok(())`** value is returned. The same inaccuracy affects inline docs of the **`ecrecover`** system function. Additionally, the **`dst`** parameter is not validated to be non-empty.

basic_system/src/system_functions/sha256.rs
```
13    impl<R: Resources> SystemFunction<R> for Sha256Impl {
14        /// If output len less than needed(32) returns `InternalError`.
```

The **`InternalError`** error is not utilized in this file, same as the length of the output buffer is not validated.

basic_bootloader/src/bootloader/process_transaction.rs
```
56    // Safe to unwrap here, as this should have been validated in the
57    // previous call.
58    let tx_type = transaction.tx_type.read();
```

This states “Safe to unwrap here” but no **`unwrap`** is actually used.

EIP-158 is mentioned instead of EIP-170

```
system_hooks/src/contract_deployer.rs
fn contract_deployer_hook_inner
171   // Although this can be called as a part of protocol upgrade,
172   // we are checking the next invariants, just in case
173   // EIP-158: reject code of length > 24576.
174   if bytecode_length as usize > MAX_CODE_SIZE {
```

The `MAX_CODE_SIZE` cap is defined by EIP-170.

Duplicated error message in “L2 Base Token” system hook

```
system_hooks/src/l2_base_token.rs
fn l2_base_token_hook_inner
172   return Ok(Err(
173       "Contract deployer failure: withdraw called with invalid calldata",
174   ));
```

The message is duplicate from the file `contract_deployer.rs` and contains incorrect prefix `Contract deployer failure`.

Bytecodes for system upgrades are not passed via calldata

In the documentation of system hooks, it is stated:

```
Bytecodes will be published separately with Ethereum calldata.
```

However, during system upgrades, only the value `bytecode_hash` is decoded from the `calldata`:

```
system_hooks/src/contract_deployer.rs
fn contract_deployer_hook_inner
158   let bytecode_hash =
159       Bytes32::from_array(calldata[32..64].try_into().expect("Always valid"))
```

The bytecode itself is retrieved from the preimage storage.

25 Redundant code

Severity: QA Commit: 96d9d37 Impact: Low Likelihood: Low Status: Notified

Unreachable code in EVM arithmetic operations

The EVM interpreter contains several unreachable code locations that can never happen to be executed due to early returns or exhaustive conditions.

In `i256_div` function

In pattern matching expression, implemented in the function **`i256_div`**, several clauses are impossible because the function returns early when the divisor is zero:

```
evm_interpreter/src/i256.rs
pub fn i256_div
76    let second_sign = i256_sign::<true>(&mut second);
77    if second_sign == Sign::Zero {
78        return U256::ZERO;
79    }
```

```
95    match (first_sign, second_sign) {
96        (Sign::Zero, Sign::Plus)
97        | (Sign::Plus, Sign::Zero)
98        | (Sign::Zero, Sign::Zero)
99        | (Sign::Plus, Sign::Plus)
100       | (Sign::Minus, Sign::Minus) => d,
101       (Sign::Zero, Sign::Minus)
102       | (Sign::Plus, Sign::Minus)
103       | (Sign::Minus, Sign::Zero)
104       | (Sign::Minus, Sign::Plus) => two_compl(d),
```

Since the function returns early when **`second_sign`** is **`Sign::Zero`**, these clauses can be removed.

In the `log2floor` function

Similarly, the return statement of the **`log2floor`** function is unreachable because all paths through the loop return before reaching it.

```
evm_interpreter/src/u256.rs
27    pub(crate) fn log2floor(value: &U256) -> u64 {
28        assert!(value != &U256::ZERO);
```

```
29        let mut l: u64 = 256;
30        for i in 0..4 {
31            let i = 3 - i;
32            if value.as_limbs()[i] == 0u64 {
33                l -= 64;
34            } else {
35                l -= value.as_limbs()[i].leading_zeros() as u64;
36                if l == 0 {
37                    return l;
38                } else {
39                    return l - 1;
40                }
41            }
42        }
43        l
44    }
```

The line 33 may look like there can be a scenario, in which the loop does not return early. However, if all the 4 iteration of the loop reach the line 33, that means that the **`value`** is zero what contradicts to the assertion on line 28.

Potential for simplification

system_hooks/src/precompiles.rs

```
106   // cheat
107   let snapshot = system.memory.start_memory_frame();
108   let mut buffer = QuasiVec::new(system);
109   let result = F::execute(&calldata, &mut buffer, &mut resources, allocator);
```

system_hooks/src/precompiles.rs

```
122   } else {
123       system.memory.empty_immutable_slice()
124   };
125   system.memory.finish_memory_frame(Some(snapshot));
```

The value **`snapshot`** is not actually utilized, since no other frames are being started between the frame creation and release. The **`snapshot`** value is simply the number of frames on the stack at the moment of opening new fram. Passing **`None`** during the frame release would have the same effect, in the current scenario. It is possible to discard the **`snapshot`** variable completely and simplify the flow:

```
system.memory.start_memory_frame();
...
system.memory.finish_memory_frame(None);
```

Unnecessary validations

```
basic_system/src/system_functions/mod.rs
18    #[inline(always)]
19    fn bytereverse(input: &mut [u8]) {
20        assert!(input.len() % 2 == 0);
```

This validation does not prevent any issue. If the `input` slice contains odd number of elements, the central element remains on its place, as expected.

Structure `WarmStorageValue` and its field `initial_value_used`

```
basic_system/src/system_implementation/flat_storage_model/storage_cache.rs
pub fn iter_as_storage_types(
571   // Using the WarmStorageValue temporarily till it's outed from the codebase. We're
572   // not actually 'using' it.
573   WarmStorageValue {
574       current_value: *current_record.value(),
575       is_new_storage_slot: initial_record.appearance() == Appearance::Unset,
576       initial_value: *initial_record.value(),
577       initial_value_used: true,
578       ..Default::default()
579   }
```

The `initial_value_used` field of the `WarmStorageValue` structure is never initialized with values other than `true`. Additionally, the comment, located in the only location where the `WarmStorageValue` structure is constructed, states that this structure should be removed.

```
basic_system/src/system_implementation/flat_storage_model/simple_growable_storage.rs
242   for (key, value) in reads_iter {
243       let flat_key = derive_flat_storage_key(&key.address, &key.key);
244       // reads
245       let expect_new = value.is_new_storage_slot;
246       assert!(value.initial_value_used);
```

As a consequence, the validation of the `initial_value_used` field, located in the batch verification code, is redundant.

Commented-out code

basic_system/src/system_implementation/flat_storage_model/simple_growable_storage.rs

```
247   if expect_new {
248       // assert_eq!(
249       //     value.initial_value,
250       //     WarmStorageValue::TRIVIAL_VALUE,
251       //     "initial value of empty slot must be trivial"
252       // );
253       num_nonexisting_reads += 1;
```

Unused error messages

These error variants are not utilized in the codebase. This is not only a redundancy but, sometimes, can indicate a missed corner case:

basic_bootloader/src/bootloader/errors.rs

```
pub enum InvalidTransaction
28    /// EIP-1559: `gas_price` is less than `basefee`.
29    GasPriceLessThanBasefee,
```

```
32    /// Initial gas for a Call is bigger than `gas_limit`.
33    ///
34    /// Initial gas for a Call contains:
35    /// - initial stipend gas
36    /// - gas for access list and input data
37    CallGasCostMoreThanGasLimit,
```

```
45    /// Overflow payment in transaction.
46    OverflowPaymentInTransaction,
```

```
64    /// Transaction chain id does not match the config chain id.
65    InvalidChainId,
```

```
66    /// Access list is not supported for blocks before the Berlin hardfork.
67    AccessListNotSupported,
```

Recommendation

Remove unreachable code branches to improve code clarity and maintainability. Consider using exhaustive pattern matching without impossible cases. Simplify complex data flows and remove unnecessary validations, unused error messages, commented-out code and outdated comments.

26 Unimplemented functionality

Severity: QA Impact: Low Likelihood: Low Status: Notified

Block timestamp and state root hash validation

basic_system/src/system_implementation/system/io_subsystem.rs

```
fn finish
486   // chain state before
487   let chain_state_commitment_before = ChainStateCommitment {
488       state_root: state_commitment.root,
489       next_free_slot: state_commitment.next_free_slot,
490       block_number: block_metadata.block_number - 1,
491       last_256_block_hashes_blake: blocks_hasher.finalize().into(),
492       // TODO(EVM-1080): we should set and validate that current block timestamp >= previous
493       last_block_timestamp: 0,
494   };
```

```
526   // chain state after
527   let chain_state_commitment_after = ChainStateCommitment {
528       state_root: state_commitment.root,
529       next_free_slot: state_commitment.next_free_slot,
530       block_number: block_metadata.block_number,
531       last_256_block_hashes_blake: blocks_hasher.finalize().into(),
532       // TODO(EVM-1080): we should set and validate that current block timestamp >= previous
533       last_block_timestamp: 0,
534   };
```

```
let mut full_root_hasher = crypto::sha3::Keccak256::new();
full_root_hasher.update(self.logs_storage.tree_root().as_u8_ref());
620   full_root_hasher.update([0u8; 32]); // aggregated root 0 for now
```

Backward-compatibility events related to withdrawals

system_hooks/src/l2_base_token.rs
```
// TODO: emit event for withdrawal for Era compatibility
186   Ok(Ok(&[]))
```

```
fn l2_base_token_hook_inner
286   // TODO: emit event for Era compatibility
287   Ok(Ok(&[]))
```

Nonce holder

system_hooks/src/lib.rs
```
56    // Temporarily disabled, only used for AA.
57    // pub mod nonce_holder;
```

```
212   // /// Adds nonce holder system hook.
213   // ///
214   // pub fn add_nonce_holder(&mut self) {
215   //     self.add_hook(NONCE_HOLDER_HOOK_ADDRESS_LOW, nonce_holder_hook)
216   // }
```

system_hooks/src/nonce_holder.rs
```
13    // it's actually stateless since we can immediately delegate a functionality to system itself
```

The contract is not actually utilized. During transaction processing, account nonces are retreived from the oracle via the **`basic_system`** layer. Constants **`NONCE_HOLDER_HOOK_ADDRESS`** and **`ACCOUNT_PROPERTIES_STORAGE_ADDRESS`** have the same value of **`x8003`**.

system_hooks/src/nonce_holder.rs
```
pub fn nonce_holder_hook
61    // TODO: ensure onlySystemCall
```

Factory deps during L1 transaction preparation

basic_bootloader/src/bootloader/process_transaction.rs
```
fn process_l1_transaction
147   // TODO: l1 transaction preparation (marking factory deps)
148   let chain_id = system.get_chain_id();
```

```
fn transaction_execution
771   // TODO: factory deps? Probably fine to ignore for now
```

27 Using raw `u8` type instead of `enum`

Severity: QA Commit: 96d9d37 Impact: Low Likelihood: Low Status: Notified

In **`runner.rs:1001-1009`**, raw **`u8`** values is used for **`ExecutionEnvironmentType`** instead of enum:

basic_bootloader/src/bootloader/runner.rs

```
1001  fn create_ee<S: EthereumLikeTypes>(
1002      ee_type: u8,
1003      system: &mut System<S>,
1004  ) -> Result<Box<SupportedEEVMState<'static, S>, S::Allocator>, InternalError> {
1005      Ok(Box::new_in(
1006          SupportedEEVMState::create_initial(ee_type, system)?,
1007          system.get_allocator(),
1008      ))
1009  }
```

Consider using **`ExecutionEnvironmentType`** enum throughout for better type safety and code maintainability.

TARAN SPACE

Security Auditing, Research, and Advisory
for the Decentralized Web