Zellic

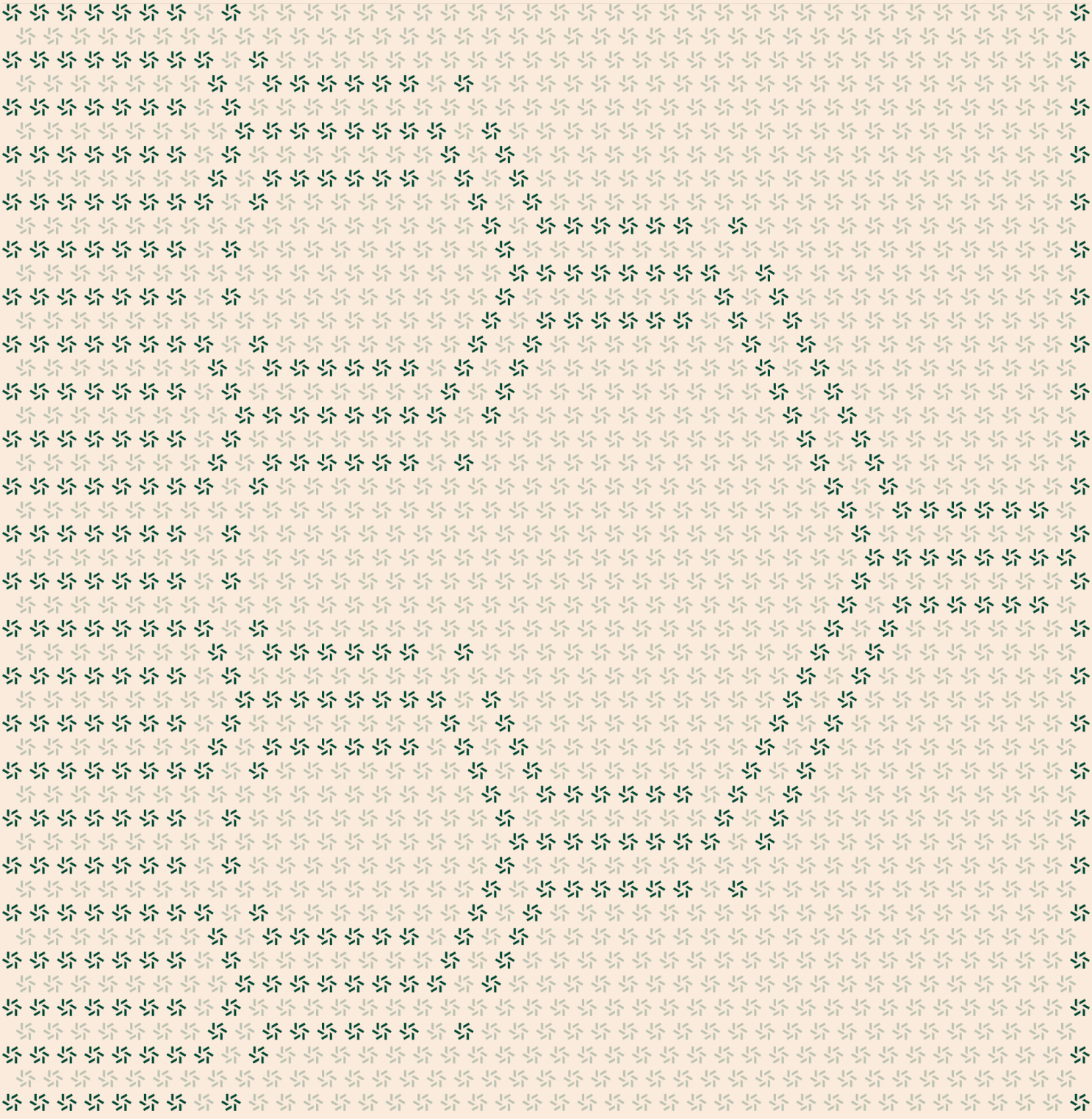
Prepared for
Clement Walter
KKRT Labs

Prepared by
Filippo Cremonese
Jinseo Kim
Zellic

September 6, 2024

Kakarot EVM

zkEVM Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for KKRT Labs from June 20th to August 21st, 2024. During this engagement, Zellic reviewed Kakarot EVM's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the Kakarot architecture sound from a security perspective?
- Does the implementation of Ethereum transaction processing conform to the specification?
- Does the implementation of the EVM opcode semantics conform to the EVM specification?
- Does the implementation of the EVM opcode gas accounting conform to the EVM specification?
- Are there any security issues related to Kakarot extensions to the EVM specification, such as in precompiles?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

We did not attempt to identify all possible issues related to possible mismatches between the EVM gas costs and the Starknet execution (proof and/or validation) cost.

Considering the number of severe findings uncovered during the audit, we note that improper handling of arithmetic overflows, input transaction parsing, and conversion between packed and unpacked formats remain key areas of concern and possible sources of security issues.

1.4. Results

During our assessment on the scoped Kakarot EVM contracts, we discovered 25 findings. Four critical issues were found. Four were of high impact, six were of medium impact, eight were of low impact, and the remaining findings were informational in nature. All issues have been acknowledged and addressed by KKRT Labs, as documented in the remediation section of each finding.

Breakdown of Finding Impacts

Impact Level	Count
Critical	4
High	4
Medium	6
Low	8
Informational	3

2. Introduction

2.1. About Kakarot EVM

KKRT Labs contributed the following description of Kakarot EVM:

> Kakarot is a provable EVM built in Cairo, Starkware's Turing-complete STARK-friendly ZK-DSL.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Underconstrained circuits. The most common type of vulnerability in a ZKP circuit is not adding sufficient constraints to the system. This leads to proofs generated with incorrect witnesses in terms of the specification of the project being accepted by the ZKP verifier. We manually check that the set of constraints satisfies soundness, enough to remove all such possibilities and in some cases provide a proof of the fact.

Overconstrained circuit. While rare, it is possible that a circuit is overconstrained. In this case, appropriately assigning witnesses will become impossible, leading to a vulnerability. To prevent this, we manually check that the constraint system is set up with completeness so that the proofs generated with the correct set of witnesses indeed pass the ZKP verification.

Missing range checks. This is a popular type of an underconstrained-circuit vulnerability. Due to the usage of field arithmetic, overflow checks and range checks serve a huge purpose to build applications that work over the integers. We manually check the code for such missing checks and, in certain cases, provide a proof that the given set of range checks is sufficient to constrain the circuit up to specification.

Cryptography. ZKP technology and their applications are based on various aspects of cryptography. We manually review the cryptography usage of the project and examine the relevant studies and standards for any inconsistencies or vulnerabilities.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices, guidelines, and code quality standards.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3. Scope

The engagement involved a review of the following targets:

Kakarot EVM Contracts

Type	Cairo0
Platform	Starknet
Target	kakarot
Repository	https://github.com/kkrt-labs/kakarot ↗
Version	`464d254fbb6483964fc986a726314db36b778322`

Programs	`Account contract`
	`Account proxy contract`
	`EVM contract`
	`Precompiles`
	`Starknet backend`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of nine person-weeks. The assessment was conducted by two consultants over the course of nine calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Filippo Cremonese
Engineer
fcremo@zellic.io

Jinseo Kim
Engineer
jinseo@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

June 20, 2024	Start of primary review period
August 21, 2024	End of primary review period

3. Detailed Findings

3.1. Sender spoofing to `eth_call` leading to arbitrary Cairo code execution

Target	EVM		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Kakarot provides an `eth_call` entry point, which is intended to execute an EVM message without having a direct effect on the global EVM state. The function does not commit its effects to storage, so it is used both for simulating a transaction (e.g., for gas estimation) as well as for executing a transaction by `eth_send_transaction`, which has the responsibility to commit the state to storage.

When simulating a transaction, the address of the EVM message sender is arbitrary. This would normally be sound since the function does not commit to state and is analogous to the `eth_call` endpoint of the EVM RPC. Note that the `eth_call` function is annotated with `@view`, but this attribute is not enforced at runtime by the Cairo VM.

This, combined with the ability to set an arbitrary EVM caller address, allows to exploit the capability to perform Cairo library calls to gain arbitrary code execution at the Cairo level.

This capability is intended to be used only by a set of allowlisted EVM contracts. The privileged EVM contracts use native Cairo code to implement functionality that would otherwise be too difficult or expensive to implement using EVM code. By spoofing the sender of a message, it is possible to bypass the access control checks performed in `exec_precompile` to authorize a precompile call and use the `LIBRARY_CALL_SOLIDITY_SELECTOR` to perform a Cairo library call to an arbitrary class hash. The code executed using the library call has the same context as the EVM interpreter code.

Impact

This issue allows to achieve arbitrary Cairo code execution. This ultimately leads to a full and permanent compromise of the VM by allowing an attacker to upgrade the EVM contract. The attacker can also perform arbitrary EVM state changes (via invocation of Account contracts) and transfer assets entrusted to the VM and its accounts.

Recommendations

As a short-term mitigation, ensure that the origin of a call cannot be spoofed when calling into `eth_call`, is possible. This may not be a viable solution if calling `eth_call` with an arbitrary EVM sender address is needed on chain.

Long term, we encourage to reevaluate the design choice to allow a set of privileged EVM callers

to invoke Cairo precompiles using the library call. This should not be necessary under normal circumstances, and at the time of this writing, it is not strictly required by any precompile, which can be implemented using a normal call. If library calls are deemed necessary, we recommend to restrict the capability to perform them to a set of EVM contracts separate from the set of EVM contracts allowed to perform regular Cairo calls, in order to minimize the attack surface as much as possible. Ideally, the allowlist should be implemented so that each EVM sender is explicitly allowed to call a specific set of Cairo contracts.

The current documentation docs seem to suggest[1] that developers can request allowlisting of their contracts. This privilege must only be granted to fully trusted contracts.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1281 ↗, by ensuring `eth_call` and `eth_estimate_gas` cannot be used as entry points into Kakarot in an on-chain transaction, which could be used to bypass the view-only restriction.

Additionally, the ability to perform library calls has been removed in PR #1397 ↗.

[1] https://github.com/kkrt-labs/kakarot/blob/v0.8.4/docs/general/cairo_precompiles.md?plain=1#L211

3.2. Underconstrained `load_packed_bytes` leading to arbitrary transaction manipulation

Target	utils.cairo		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `load_packed_bytes` helper function is used to unpack an array of felts into an array of individual bytes.

The function is underconstrained, allowing a malicious prover to change the returned unpacked value arbitrarily.

Impact

This helper is used throughout the codebase, including to unpack the array of felts encoding an incoming Ethereum transaction when verifying its signature and when executing it. Note that since the function allocates a new memory region every time it is called, a malicious prover can manipulate the result of every invocation independently.

A malicious prover can therefore manipulate the result of the unpacking operation only when it is invoked for unpacking a transaction for executing, providing legitimate results while verifying the transaction signature instead. This allows to arbitrarily change the transaction being executed. The impact is critical, as it allows to manipulate any EVM transaction.

Appendix 5.4. ↗ shows a basic proof of concept demonstrating the issue.

Recommendations

Assert that the `value` tempvar is zero at the end of every iteration.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1299 ↗.

3.3. Insufficient length checks leading to transaction signature-verification bypass

Target	accounts/library.cairo		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

Each EVM account (both EOAs and smart contracts) is represented on Kakarot by an instance of a Cairo class hash, deployed at a unique and deterministic address. Transactions are initiated by sending a transaction to the contract representing the EOA account of the sender of the transaction.

The Kakarot account contract implements account abstraction using the `__validate__` and `__execute__` functions. The `__validate__` function is responsible, among other things, to validate the signature of the Ethereum transaction to execute.

However, a lack of length validation of the input data allows an attacker that acts as a malicious prover to bypass the signature check for any account.

During transaction validation and execution, the input data is unpacked using `load_packed_bytes`. The function receives the length of the data to unpack as input and returns a pointer to an array of felts containing the individual unpacked bytes. The felts that follow the returned bytes are unconstrained and can be freely set by the prover. This is true for any input length, including zero, in which case the function returns a pointer to a memory region fully under control of the prover.

Since the length of the packed transaction data is controlled by the attacker, they can cause the result of `load_packed_bytes` to be completely unconstrained. As a result, a malicious attacker can submit a malicious transaction to an EOA account, with a short (zero) packed length. Acting as a prover, they can initialize the memory containing the unpacked transaction bytes to represent a legitimate signed transaction during the execution of `__validate__`, passing the signature check.

Since `__execute__` invokes `load_packed_bytes` again, it obtains a pointer to an independent unconstrained memory region. This memory region can be manipulated by the malicious prover to change the transaction that will actually be executed.

Impact

This issue allows to modify any submitted transaction and bypass signature verification for any EOA for which a valid signature can be obtained.

Recommendations

Short-term, perform strict length checks on any user input, ensuring that no out-of-bound reads are performed.

Long-term and where technically feasible, eliminate repeated parsing/unpacking of the same data.

Remediation

This issue has been acknowledged by KKRT Labs, and fixes were implemented in the following PRs:

- PR #1356
- PR #1383

3.4. Underconstrained `load_bytecode` allowing to manipulate the bytecode of any contract and arbitrary Cairo code execution

Target	accounts/library.cairo::Internal		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `load_bytecode` function is used to load the bytecode of a contract. The function loads the packed bytecode from storage and unpacks it into an array of felts encoding individual bytes.

The unpacking routine does not sufficiently constrain the values it operates on, allowing a malicious prover to arbitrarily manipulate the resulting loaded bytecode.

Impact

This issue allows a malicious prover to change arbitrarily the bytecode of any EVM contract being executed. The change is temporary and only effective during the transaction processed by the malicious prover. However, it is still a critical safety issue, as it allows to compromise any EVM contract. This includes the privileged contracts that are allowed to call Cairo precompiles, leading to arbitrary Cairo code execution due to the ability to library call Cairo code to implement precompiles. See Finding 3.1. ↗ for additional discussion of this threat.

Appendix 5.2. ↗ contains a minimal proof of concept demonstrating the issue.

Recommendations

Ensure `load_bytecode` correctly unpacks the values it operates on. The `value` tempvar must be asserted to be zero at the end of every unpacking loop.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1290 ↗.

3.5. Missing RLP-decoded–subitems length check

Target	eth_transaction.cairo		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The functions tasked with decoding Ethereum transactions are not appropriately checking the number of items contained in lists returned by the RLP-decoding routines.

Take for example the `decode_2930` function:

```
func decode_2930{bitwise_ptr: BitwiseBuiltin*, range_check_ptr}(
    tx_data_len: felt, tx_data: felt*
) -> model.EthTransaction* {
    alloc_locals;

    let (items: RLP.Item*) = alloc();
    RLP.decode(items, tx_data_len - 1, tx_data + 1);
    let sub_items_len = [items].data_len;
    let sub_items = cast([items].data, RLP.Item*);

    let chain_id = Helpers.bytes_to_felt(sub_items[0].data_len,
    sub_items[0].data);
    let nonce = Helpers.bytes_to_felt(sub_items[1].data_len,
    sub_items[1].data);
    let gas_price = Helpers.bytes_to_felt(sub_items[2].data_len,
    sub_items[2].data);
    let gas_limit = Helpers.bytes_to_felt(sub_items[3].data_len,
    sub_items[3].data);
    let destination = Helpers.try_parse_destination_from_bytes(
        sub_items[4].data_len, sub_items[4].data
    );
    let amount = Helpers.bytes_to_uint256(sub_items[5].data_len,
    sub_items[5].data);
    let payload_len = sub_items[6].data_len;
    let payload = sub_items[6].data;

    let (access_list: felt*) = alloc();
    let access_list_len = parse_access_list(
        access_list, sub_items[7].data_len, cast(sub_items[7].data, RLP.Item*)
```

```
    );
    tempvar tx = new model.EthTransaction(...);
    return tx;
}
```

The `sub_items_len` variable is not checked, leading to an out-of-bounds read of unasserted memory.

Impact

Every `RLP.Item` of the type list has its `data` member pointing to a newly allocated memory region. An out-of-bounds read allows the prover to freely choose the memory contents. Since transactions are decoded twice and independently during `__validate__` and `__execute__`, it is possible for a rogue attacker to submit RLP-encoded data that does not encode a full valid transaction but rather one that has a few (or none) of the required RLP subitems.

This issue has similar impact as Finding 3.3. ↗, allowing a malicious prover to change the transaction being executed and the transaction being verified independently. However, the attacker is more limited in that they have to obtain the signature for an invalid transaction from the victim.

Recommendations

Check the `Item.data_len` field of every RLP-decoded item before using it to ensure no out-of-bound usages are made.

Note that all functions decoding RLP data perform insufficient checks, including `decode_legacy_tx`, `decode_2930`, and `decode_1559`.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1307 ↗.

3.6. Incorrectly skipped memory writes in CALLDATACOPY, CODECOPY, and EXTCODECOPY

Target	environmental_information.cairo		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The following functions contain a similar implementation bug, which could incorrectly skip memory writes if the source offset is greater than or equal to 2^{128}:

- `exec_copy`, handling CALLDATACOPY and CODECOPY
- `exec_extcodecopy`, handling EXTCODECOPY

Consider the following snippet from `exec_extcodecopy` (edited for brevity):

```
func exec_extcodecopy(evm: model.EVM*) -> model.EVM* {
    // ...

    // Gas
    // Calling `get_account` subsequently will make the account warm for the
    next interaction
    let is_warm = State.is_account_warm(evm_address);
    tempvar access_gas_cost = is_warm * Gas.WARM_ACCESS + (1 - is_warm) *
    Gas.COLD_ACCOUNT_ACCESS;

    // Any size upper than 2**128 will cause an OOG error, considering the
    maximum gas for a transaction.
    let upper_bytes_bound = size.low + 31;
    let (words, _) = unsigned_div_rem(upper_bytes_bound, 32);
    let copy_gas_cost_low = words * Gas.COPY;
    tempvar copy_gas_cost_high = is_not_zero(size.high) * 2 ** 128;

    let memory_expansion = Gas.memory_expansion_cost_saturated(
        memory.words_len, dest_offset, size
    );

    let evm = EVM.charge_gas(
        evm, access_gas_cost + copy_gas_cost_low + copy_gas_cost_high +
    memory_expansion.cost
    );
```

```
    if (evm.reverted != FALSE) {
        return evm;
    }

    tempvar memory = // ...

    // Offset.high != 0 means that the sliced data is surely 0x00...00
    // And storing 0 in Memory is just doing nothing
    if (offset.high != 0) {
        return evm;
    }

    let (sliced_data: felt*) = alloc();
    let account = State.get_account(evm_address);
    slice(sliced_data, account.code_len, account.code, offset.low, size.low);

    Memory.store_n(size.low, sliced_data, dest_offset.low);

    return evm;
}
```

The highlighted lines contain an incorrect early return condition. Given the following pseudocode,

- `MSTORE` a value at offset 0x100
- `CALLDATACOPY` from offset 0xffff...fff to offset 0x100

the `CALLDATACOPY` should write zeros to the destination offset, but no write will be performed instead, preserving the previously written value.

Impact

We believe the conditions required for this issue to be exploitable (a `CALLDATACOPY`, `CODECOPY`, or `EXTCODECOPY` from a very high offset into previously initialized memory) to be rare in practice. ‘ However, this is still a significant deviation from the EVM specification, potentially exploitable in some edge cases.

Recommendations

Remove the incorrect early return from the function.

This issue was not detected by the official EVM unit tests. Consider submitting additional test cases that cover this edge case.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1286 ↗.

3.7. Missing overflow handling in BYTE opcode implementation

Target	stop_and_math_operations.cairo		
Category	Coding Mistakes	**Severity**	High
Likelihood	Medium	**Impact**	High

Description

The BYTE opcode (taking two stack operands `i` and `x`) is implemented by right-shifting the `x` operand. The amount by which the `x` operand is right-shifted is computed by multiplying `i` by eight.

```
// compute y = (x >> (248 - i * 8)) & 0xFF
let (mul, _) = uint256_mul(popped[0], Uint256(8, 0));
let (right) = uint256_sub(Uint256(248, 0), mul);
let (shift_right) = uint256_shr(popped[1], right);
let (result) = uint256_and(shift_right, Uint256(0xFF, 0));
```

This multiplication can overflow, but the carry bit is ignored. Additionally, the `uint256_sub` call can potentially underflow.

Impact

Due to this oversight, the wrong result is computed if `i` is very large.

This test case (added to tests/src/kakarot/instructions/test_stop_and_math_operations.py) demonstrates the issue:

```
(Opcodes.BYTE,
    [0x8000000000000000000000000000000000000000000000000000000000000000,
    0x112233445566778899000aabbccddeeff112233445566778899000aabbccddeeff],
    0x00),
```

The operation should return zero, but `0x11` is returned instead.

Recommendations

Handle the potential overflow by returning zero or by setting the shift amount to a fixed value.

This issue was not detected by the official EVM unit tests. Consider submitting additional test cases that cover this edge case.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1288 ↗.

3.8. The `split_word` helper is unsound for long lengths

Target	utils.cairo		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The `split_word` helper is used to split a felt value into `len` felts that represent its base-256 encoding.

The function is not sound if `len` is greater than 31, allowing a malicious prover to manipulate the output.

Impact

This issue allows a malicious prover to manipulate the output of the function. The function is used with a vulnerable `len` value of 32 by the functions implementing `ec_recover`, so it is possible for a malicious prover to cause a signature validation to incorrectly reject a valid signature as well as potentially accept an invalid signature.

Appendix 5.1. ↗ shows a proof of concept demonstrating how a malicious prover can manipulate the output of the function.

Recommendations

Ensure that all callers of `split_word` use a `len` of at most 31. Consider adding an assertion inside `split_word` that would directly prevent unsafe lengths from being used.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1305 ↗.

3.9. Gas-calculation overflow in `exec_mcopy`

Target	memory_operations.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The handler for the `mcopy` opcode uses an unsafe addition when computing the amount of gas to charge for the opcode.

In the following lines,

```
// GAS
let memory_expansion = Gas.max_memory_expansion_cost(
    memory.words_len, src, size, dst, size
);

// Any size upper than 2**128 will cause an OOG error, considering the maximum
    gas for a transaction.
let upper_bytes_bound = size.low + 31;
let (words, _) = unsigned_div_rem(upper_bytes_bound, 32);
let copy_gas_cost_low = words * Gas.COPY;
tempvar copy_gas_cost_high = is_not_zero(size.high) * 2 ** 128;

let evm = EVM.charge_gas(
    evm, memory_expansion.cost + copy_gas_cost_low + copy_gas_cost_high
);
```

the addition of `memory_expansion.cost` and `copy_gas_cost_high` can overflow, resulting in the incorrect amount of gas being charged.

Impact

This issue could allow disproportionately expensive operations to be charged an insufficient amount of gas. This is also a deviation from the official EVM specification.

Recommendations

Avoid adding up the individual gas amounts and charge them individually. Alternatively, handle the potential overflow.

Remediation

The gas calculation in the `mcopy` handler is not subject to overflowing anymore due to the changes introduced in PR #1298 ↗.

3.10. Overflow in `max_memory_expansion_cost`

Target	gas.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Given two memory chunks, the `max_memory_expansion_cost` helper function computes the maximum expansion cost based on the maximum offset reached by each chunk.

The function receives Uint256 arguments but operates using felts, failing to handle an overflow in the line highlighted in the following excerpt.

```
func max_memory_expansion_cost{range_check_ptr}(
        words_len: felt, offset_1: Uint256*, size_1: Uint256*, offset_2:
    Uint256*, size_2: Uint256*
    ) -> model.MemoryExpansion {
    alloc_locals;
    let (is_zero_1) = uint256_eq([size_1], Uint256(0, 0));
    let (is_zero_2) = uint256_eq([size_2], Uint256(0, 0));
    let is_zero = is_zero_1 * is_zero_2;
    if (is_zero != FALSE) {
        let expansion = model.MemoryExpansion(cost=0,
    new_words_len=words_len);
        return expansion;
    }
    let max_expansion_is_2 = is_le(offset_1.low + size_1.low, offset_2.low +
    size_2.low);
    let max_expansion = max_expansion_is_2 * (offset_2.low + size_2.low) + (
        1 - max_expansion_is_2
    ) * (offset_1.low + size_1.low);

    let low_expansion = calculate_gas_extend_memory(words_len, max_expansion);
    let expansion_cost = low_expansion.cost + (
        offset_1.high + size_1.high + offset_2.high + size_2.high
    ) * Gas.MEMORY_COST_U128;
    let expansion = model.MemoryExpansion(
        cost=expansion_cost, new_words_len=low_expansion.new_words_len
    );
    return expansion;
}
```

Impact

This helper is used to compute the memory-expansion cost for multiple operations. This issue can result in a wrong (lower-than-intended) amount of gas being charged for a very large memory-expansion operation. It also represents a violation of the official EVM specification.

Recommendations

Detect and appropriately handle the overflow, especially ensuring the amount of gas charged does not roll back to a low value.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1298 ↗.

3.11. Incorrect zero-sized access handling in `max_memory_expansion_cost`

Target	gas.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Given two memory chunks, the `max_memory_expansion_cost` helper function computes the maximum expansion cost based on the maximum offset reached by each chunk.

The function is not correctly handling the case where only one memory access has zero size while the other does not.

```
func max_memory_expansion_cost{range_check_ptr}(
        words_len: felt, offset_1: Uint256*, size_1: Uint256*, offset_2:
    Uint256*, size_2: Uint256*
    ) -> model.MemoryExpansion {
    alloc_locals;
    let (is_zero_1) = uint256_eq([size_1], Uint256(0, 0));
    let (is_zero_2) = uint256_eq([size_2], Uint256(0, 0));
    let is_zero = is_zero_1 * is_zero_2;
    if (is_zero != FALSE) {
        let expansion = model.MemoryExpansion(cost=0,
    new_words_len=words_len);
        return expansion;
    }
    let max_expansion_is_2 = is_le(offset_1.low + size_1.low, offset_2.low +
    size_2.low);
    let max_expansion = max_expansion_is_2 * (offset_2.low + size_2.low) + (
        1 - max_expansion_is_2
    ) * (offset_1.low + size_1.low);

    let low_expansion = calculate_gas_extend_memory(words_len, max_expansion);
    let expansion_cost = low_expansion.cost + (
        offset_1.high + size_1.high + offset_2.high + size_2.high
    ) * Gas.MEMORY_COST_U128;
    let expansion = model.MemoryExpansion(
        cost=expansion_cost, new_words_len=low_expansion.new_words_len
    );
    return expansion;
}
```

Impact

The function can incorrectly return a zero cost if only one of the two memory accesses has zero size. This is a violation of the EVM specification and could allow a smart contract to expand memory to an arbitrarily large size without paying the intended gas cost.

Recommendations

Only return zero if both memory accesses are zero-sized.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1298 ↗.

3.12. Underconstrained `felt_to_bytes_little`

Target	utils/bytes.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The `felt_to_bytes_little` function is used to convert a felt to an array of felts that encode the input as a sequence of little-endian byte-sized values.

The function relies on oracle hints to perform a modulo operation; however, it is missing constraints on the output bytes that assert they are in the 0–255 range.

Impact

The function is indirectly used by several parts of the codebase. The impact of this issue is not entirely clear; however, we believe the issue can have medium to critical severity.

Recommendations

Ensure the output values of the function are constrained between 0 and 255.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1317 ↗.

3.13. Overflow in the `memory_expansion_cost_saturated` function

Target	gas.cairo		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `memory_expansion_cost_saturated` function calculates the cost of memory expansion, given the current memory size and the access offset/size:

```
func memory_expansion_cost_saturated{range_check_ptr}(
    words_len: felt, offset: Uint256, size: Uint256
) -> model.MemoryExpansion {
    let (is_zero) = uint256_eq(size, Uint256(low=0, high=0));
    if (is_zero != FALSE) {
        let expansion = model.MemoryExpansion(cost=0,
    new_words_len=words_len);
        return expansion;
    }

    if (offset.high + size.high != 0) {
        // Hardcoded value of cost(2^128) and size of 2**128 bytes (will
    produce OOG in any case)
        let expansion = model.MemoryExpansion(
            cost=MEMORY_COST_U128,
    new_words_len=0x800000000000000000000000000000000
        );
        return expansion;
    }

    return calculate_gas_extend_memory(words_len, offset.low + size.low);
}
```

If the access size is zero, this function returns the zero cost. If the offset or the size is greater than or equal to 2^{128}, this function returns a hardcoded cost that will always result in an out-of-gas error. If neither are zero, the `calculate_gas_extend_memory` will calculate the exact cost for memory expansion.

```
func calculate_gas_extend_memory{range_check_ptr}(
    words_len: felt, max_offset: felt
```

```
) -> model.MemoryExpansion {
    alloc_locals;
    let memory_expansion = is_le(words_len * 32 - 1, max_offset);
    if (memory_expansion == FALSE) {
        let expansion = model.MemoryExpansion(cost=0,
    new_words_len=words_len);
        return expansion;
    }

    let prev_cost = memory_cost(words_len);
    let (new_words_len, _) = unsigned_div_rem(max_offset + 31, 32);
    let new_cost = memory_cost(new_words_len);

    let expansion_cost = new_cost - prev_cost;
    let expansion = model.MemoryExpansion(cost=expansion_cost,
    new_words_len=new_words_len);
    return expansion;
}
```

If the current memory size, `words_len * 32`, completely contains the accessed memory, this function returns zero. This check is done by the line `is_le(words_len * 32 - 1, max_offset)`.

The following code implements the `is_le` function:

```
// Returns 1 if a >= 0 (or more precisely 0 <= a < RANGE_CHECK_BOUND).
// Returns 0 otherwise.
@known_ap_change
func is_nn{range_check_ptr}(a) -> felt {
    // (...)
}

// Returns 1 if a <= b (or more precisely 0 <= b - a < RANGE_CHECK_BOUND).
// Returns 0 otherwise.
@known_ap_change
func is_le{range_check_ptr}(a, b) -> felt {
    return is_nn(b - a);
}
```

It should be noted that the `is_le` function assumes that the difference of two input parameters are less than 2^{128}, because the value the `is_le` function returns is `0 <= b - a < RANGE_CHECK_BOUND = 2 ** 128`.

Let us assume that the attacker accesses the memory with the offset $2^{128} - 1$ and the size $2^{128} - 1$. The `max_offset` will be $2^{129} - 2$ here, breaking the above assumption because of the line `is_le(words_len * 32 - 1, max_offset)`.

The `is_le` function will return false with these inputs provided. As a result, this memory access will

not be reverted.

Impact

It is possible for contracts to expand memory without paying the intended gas cost. This is a deviation from the EVM standard.

Recommendations

Check for overflows and/or use functions that work over the full felt range to implement gas calculations.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1322 ↗.

3.14. Missing maximum-length checks

Target	Kakarot		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	High	**Impact**	Medium

Description

The helpers `Helpers.bytes_to_felt` and `Helpers.bytes_to_uint256` are used throughout the codebase to convert an array of felts to a single felt or Uint256 value.

The two functions assume that the length of the input array is small enough for the result to fit, respectively, in a felt or Uint256. If the input length is too big, the functions silently overflow.

Multiple usages do not appropriately check the input length and are potentially vulnerable. For example, consider this excerpt from `decode_legacy_transaction`:

```
func decode_legacy_tx{bitwise_ptr: BitwiseBuiltin*, range_check_ptr}(
    tx_data_len: felt, tx_data: felt*
) -> model.EthTransaction* {
    // see https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md
    alloc_locals;
    let (items: RLP.Item*) = alloc();
    RLP.decode(items, tx_data_len, tx_data);

    // the tx is a list of fields, hence first level RLP decoding
    // is a single item, which is indeed the sought list
    assert [items].is_list = TRUE;
    let sub_items_len = [items].data_len;
    let sub_items = cast([items].data, RLP.Item*);

    let nonce = Helpers.bytes_to_felt(sub_items[0].data_len, sub_items[0].
        data);
    let gas_price = Helpers.bytes_to_felt(sub_items[1].data_len, sub_items[1].
        data);
    let gas_limit = Helpers.bytes_to_felt(sub_items[2].data_len, sub_items[2].
        data);
    // ...
}
```

The code is decoding the `nonce`, `gas_price`, and `gas_limit` without asserting a bound on the

`data_len` of the RLP item.

Impact

Among the several potentially vulnerable usages of `bytes_to_felt` and `bytes_to_uint256`, the highest impacts appear to be ones related to transaction parsing. While the issue does not appear to be trivially exploitable to compromise the VM in a significant way, it represents a nonconformity with respect to the EVM specifications, which should be corrected. Given the number of usages, we cannot exclude exploitability of the issue with high confidence.

Recommendations

Ensure the length of the arrays passed to `bytes_to_felt` and `bytes_to_uint256` is at most, respectively, 31 and 32.

We recommend performing this check directly in the functions rather than at the call sites; if an equivalent function lacking checks is desired for efficiency reasons in occasions where the length is guaranteed to be within acceptable bounds, we recommend creating one such function named explicitly as unsafe.

Remediation

This issue has been acknowledged by KKRT Labs, and fixes were implemented in the following PRs:

- PR #1393 ↗
- PR #1383 ↗

3.15. Incorrect `bytes32_to_felt` usage

Target	kakarot_precompiles.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `bytes32_to_felt` helper function can be used to load an array of 32 felts in big-endian order as a single felt value. The function makes two assumptions:

- Every input felt has a value between 0 and 255 (inclusive); there is no check ensuring this.
- The reconstructed value does not overflow a felt; overflows silently wrap around.

There are several usages of `bytes32_to_felt` in kakarot_precompiles.cairo, specifically in the `cairo_precompile` and `cairo_message` functions.

Here are potentially unsound usages in `cairo_precompile`:

- `let starknet_selector = Helpers.bytes32_to_felt(starknet_selector_ptr);`
- `let data_offset = Helpers.bytes32_to_felt(data_offset_ptr);`
- `let data_words_len = Helpers.bytes32_to_felt(data_len_ptr);`

A potentially unsound usage in `cairo_message` is `let target_address = Helpers.bytes32_to_felt(input);`.

Impact

The usage in `cairo_message` is potentially reachable from the EVM, as the target address being decoded from calldata could be an invalid address where the most significant 12 bytes are not null. The usage of `bytes32_to_felt` could overflow and accept an invalid address that should not be accepted.

Recommendations

Different options are available, with the most appropriate solution depending on the specific usage.

If the full 256-bit range is allowed in the input, replace `bytes32_to_felt` with a function that returns a Uint256.

In cases where inputs that overflow a felt are possible but invalid, we suggest to modify `bytes32_to_felt` to assert no overflows occur. Alternatively, the caller could be modified as well.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1339 ↗.

3.16. Missing check for type of decoded RLP items

Target	EVM		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

RLP decoding is required to parse Ethereum transactions submitted for execution. The RLP encoding defines three types of elements: byte strings, lists, and integers.

The RLP decoding routines used by Kakarot do not distinguish between byte strings and integers but do distinguish lists. However, the usages of the routines made while decoding a transaction do not check the type of the decoded elements.

Impact

The issue does not appear to be practically exploitable; however, we are not fully confident in this and highly recommend to remediate this issue to prevent any possible exploit. Additionally, this issue constitutes a violation of the EVM specification, as it leads to accepting invalidly encoded transactions.

Recommendations

We recommend to add checks that ensure the correct type of every decoded RLP item, as follows:

```
  let sub_items_len = [items].data_len;
  let sub_items = cast([items].data, RLP.Item*);

assert sub_items[0].is_list = FALSE;
  let nonce = Helpers.bytes_to_felt(sub_items[0].data_len, sub_items[0].data);
assert sub_items[1].is_list = FALSE;
  let gas_price = Helpers.bytes_to_felt(sub_items[1].data_len,
     sub_items[1].data);
assert sub_items[2].is_list = FALSE;
  let gas_limit = Helpers.bytes_to_felt(sub_items[2].data_len,
     sub_items[2].data);
```

Note that the above is not a complete patch, and assertions are needed for every RLP subitem.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1307 ↗.

3.17. Invalid Uint256 construction in `AccountContract::validate`

Target	AccountContract::validate		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

The `AccountContract::validate` function is constructing Uint256 without ensuring that the individual legs are less than 2^{128}.

```
let (tx_info) = get_tx_info();

// Assert signature field is of length 5: r_low, r_high, s_low, s_high, v
assert tx_info.signature_len = 5;
let r = Uint256(tx_info.signature[0], tx_info.signature[1]);
let s = Uint256(tx_info.signature[2], tx_info.signature[3]);
```

Impact

This issue could allow to represent the same signature with multiple encodings. Bypassing signature verification does not appear to be possible.

Recommendations

Require the individual legs used to construct the Uint256 to be less than 2^{128}.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1331 ↗.

3.18. Overflow in effective gas calculation

Target	accounts/library.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Medium	**Impact**	Low

Description

Kakarot EVM calculates the gas price of a transaction in the following code:

```
// ensure that the user was willing to at least pay the base fee
let enough_fee = is_le(base_fee, tx.max_fee_per_gas);
let max_fee_greater_priority_fee = is_le(tx.max_priority_fee_per_gas,
    tx.max_fee_per_gas);
let max_gas_fee = tx.gas_limit * tx.max_fee_per_gas;
let (max_fee_high, max_fee_low) = split_felt(max_gas_fee);
let (tx_cost, carry) = uint256_add(tx.amount, Uint256(low=max_fee_low,
    high=max_fee_high));
assert carry = 0;
let (is_balance_enough) = uint256_le(tx_cost, balance);

if (enough_fee * max_fee_greater_priority_fee * is_balance_enough *
    tx_gas_fits_in_block == 0) {
    // (...)
} else {
    // priority fee is capped because the base fee is filled first
    let possible_priority_fee = tx.max_fee_per_gas - base_fee;
    let priority_fee_is_max_priority_fee = is_le(
        tx.max_priority_fee_per_gas, possible_priority_fee
    );
    let priority_fee_per_gas = priority_fee_is_max_priority_fee *
        tx.max_priority_fee_per_gas + (1 - priority_fee_is_max_priority_fee) *
        possible_priority_fee;
    // signer pays both the priority fee and the base fee
    let effective_gas_price = priority_fee_per_gas + base_fee;

    let (return_data_len, return_data, success, gas_used)
    = IKakarot.eth_send_transaction(
        // (...)
        gas_price=effective_gas_price,
        // (...)
    );
```

```
}
```

The effective gas price is calculated by adding the priority fee to the base fee. However, it should be noted that no check is performed if this addition overflows. While the code may seem to check this by enforcing the maximum limit for the priority fee, the priority fee can be a negative value because of the way the `is_le` function works.

Impact

An attacker can submit a transaction with the gas price lower than the base fee.

Recommendations

Consider checking that the given priority fee of a transaction is nonnegative.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1329 ↗.

3.19. Incorrect usage of `try_parse_destination_from_bytes`

Target	utils/utils.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `try_parse_destination_from_bytes` function is used to parse an EVM address from a byte array. If the length of the bytes array is not 20, the function returns an empty option.

```
func try_parse_destination_from_bytes(bytes_len: felt, bytes: felt*) ->
    model.Option {
    if (bytes_len != 20) {
        let res = model.Option(is_some=0, value=0);
        return res;
    }
    let address = bytes20_to_felt(bytes);
    let res = model.Option(is_some=1, value=address);
    return res;
}
```

All usages of the helper are made during transaction parsing to decode the destination address. The length and data decoded from the RLP-encoded transaction are passed directly to the function, without checking the data length. The following example is from the `decode_2930` function, with the other instances having a virtually equivalent flaw:

```
func decode_2930{bitwise_ptr: BitwiseBuiltin*, range_check_ptr}(
    tx_data_len: felt, tx_data: felt*
) -> model.EthTransaction* {
    alloc_locals;

    let (items: RLP.Item*) = alloc();
    RLP.decode(items, tx_data_len - 1, tx_data + 1);
    let sub_items_len = [items].data_len;
    let sub_items = cast([items].data, RLP.Item*);

    let chain_id = Helpers.bytes_to_felt(sub_items[0].data_len,
    sub_items[0].data);
    let nonce = Helpers.bytes_to_felt(sub_items[1].data_len,
    sub_items[1].data);
```

```
    let gas_price = Helpers.bytes_to_felt(sub_items[2].data_len,
    sub_items[2].data);
    let gas_limit = Helpers.bytes_to_felt(sub_items[3].data_len,
    sub_items[3].data);
    let destination = Helpers.try_parse_destination_from_bytes(
        sub_items[4].data_len, sub_items[4].data
    );
    // more code omitted ...
}
```

Impact

The behavior of the transaction-decoding functions is incorrect as it accepts a transaction with an invalidly encoded address as if it was sent to no address. Transactions sent to no address are treated as contract deployment transactions.

While this issue appears to be unlikely to be exploitable in practice, it is a violation of the Ethereum specification.

Recommendations

Reject invalidly encoded transactions, including invalidly encoded addresses.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1384 ↗.

3.20. Underconstrained `bytes_used_128`

Target	utils.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The `bytes_used_128` function calculates the number of bytes used by a 128-bit value. The function is implemented as a manually unrolled loop.

```
func bytes_used_128{range_check_ptr}(value: felt) -> felt {
    tempvar base = 256 ** 15;
    let bound = base;
    tempvar max = base - 1;
    let (output) = alloc();

   %{
       memory[ids.output] = res = (int(ids.value) % PRIME) % ids.base
       assert res < ids.bound, f'split_int(): Limb {res} is out of range.'
   %}
   tempvar x = [output];
    [range_check_ptr] = x;
    assert [range_check_ptr + 1] = max - x;

    let range_check_ptr = range_check_ptr + 2;
    if (value - x != 0) {
        return 16;
    }
    tempvar base = base / 256;
    let bound = base;
    tempvar max = base - 1;

    // 15 more manually unrolled iterations omitted...
}
```

As can be observed in the snippet above, the function is underconstrained, and as a result, a malicious prover can pick any value `x` as long as `0<=x<=256**15`. Therefore, the prover can cause the function to return an incorrect length for the number of bytes required to represent a 128-bit number.

Impact

The function appears to be used only by the implementation of the EXP opcode to compute the gas charge. Therefore, it could be abused by a malicious prover to alter the amount of gas charged for the opcode, violating the EVM specification. We consider the impact of this issue to be low.

Appendix 5.3. ↗ shows a basic proof of concept demonstrating the issue.

Recommendations

Assert a lower bound on the value of x, adjusted according to the loop unroll iteration.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1301 ↗.

The fix simplifies the function significantly, maintaining the unrolled loop form but leveraging the library function `unsigned_div_rem` to perform a division (by 256^{15}, 256^{14}, ...) and determining whether the unrolled loop execution should continue or not depending on the quotient of the division.

3.21. Ignored `RLP.decode` return value

Target	EVM		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The `RLP.decode` function is used in three instances to decode Ethereum transactions. The function returns the number of decoded items, but all usages ignore the return value of the function.

Impact

This issue does not appear to be exploitable. However, it is a violation of the official EVM specifications and a potential source of future vulnerabilities.

Appendix 5.5. ↗ contains a proof of concept demonstrating the issue.

Recommendations

Assert that the number of root items decoded from RLP is one when decoding a transaction.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1307 ↗.

3.22. Invalid destination L1 EVM address accepted by `cairo_message`

Target	kakarot_precompiles.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `bytes32_to_felt` function converts the given 32-byte data into a felt value:

```
@known_ap_change
func bytes32_to_felt(val: felt*) -> felt {
    let current = [val] * 256 ** 31;
    let current = current + [val + 1] * 256 ** 30;
    let current = current + [val + 2] * 256 ** 29;
    // (...)
    let current = current + [val + 29] * 256 ** 2;
    let current = current + [val + 30] * 256 ** 1;
    let current = current + [val + 31];
    return current;
}
```

It should be noted that `Helpers.bytes32_to_felt` silently overflows if the input value is greater than the value that can be represented in a felt. However, the `cairo_message` precompile, which exposes the `send_message_to_l1` syscall to the EVM, is parsing the target L1 EVM as follows:

```
// Input is formatted as:
// [to_address: address][data_offset: uint256][data_len: uint256][data:
    bytes[]]

// Load target EVM address
let target_address = Helpers.bytes32_to_felt(input);
```

Because EVM addresses are only 20 bytes long, this code accepts invalid destination addresses.

Impact

The impact of this bug is unclear and could depend on external contracts, including the receiver of the message on the L1 chain. We deem the issue to be unlikely to be exploitable in common circumstances.

Recommendations

Reject invalid destination addresses longer than 20 bytes and ensure that the bytes-to-felt conversion does not overflow.

Note that there are other usages of `bytes32_to_felt` made by the `cairo_precompile` precompile. Some of these other usages are potentially at risk of overflowing silently as well. While the arguments to this function can be considered trusted (since the EVM caller is trusted), we recommend to consider adding additional checks to all `bytes32_to_felt` usages.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1339 ↗.

3.23. Inefficient implementation of the EXTCODEHASH opcode

Target	environmental_information.cairo		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The following code implements the EXTCODEHASH opcode, which should provide the codehash of the given address:

```
func exec_extcodehash{
    syscall_ptr: felt*,
    pedersen_ptr: HashBuiltin*,
    range_check_ptr,
    bitwise_ptr: BitwiseBuiltin*,
    stack: model.Stack*,
    memory: model.Memory*,
    state: model.State*,
}(evm: model.EVM*) -> model.EVM* {
    alloc_locals;

    let (address_uint256) = Stack.pop();
    let evm_address = uint256_to_uint160([address_uint256]);

    // Gas
    // Calling `get_account` subsequently will make the account warm for the
    next interaction
    let is_warm = State.is_account_warm(evm_address);
    tempvar access_gas_cost = is_warm * Gas.WARM_ACCESS + (1 - is_warm) *
        Gas.COLD_ACCOUNT_ACCESS;
    let evm = EVM.charge_gas(evm, access_gas_cost);
    if (evm.reverted != FALSE) {
        return evm;
    }

    let account = State.get_account(evm_address);
    let has_code_or_nonce = Account.has_code_or_nonce(account);
    let account_exists = has_code_or_nonce + account.balance.low +
    account.balance.high;
    // Relevant cases:
```

```
    // https://github.com/ethereum/go-
    ethereum/blob/master/core/vm/instructions.go#L392
    if (account_exists == FALSE) {
        Stack.push_uint128(0);
        return evm;
    }

    let (local dst: felt*) = alloc();
    let (dst_len, last_word, last_word_num_bytes)
    = bytes_to_bytes8_little_endian(
        dst, account.code_len, account.code
    );

    let (implementation) = Kakarot_cairo1_helpers_class_hash.read();
    let (code_hash) = ICairo1Helpers.library_call_keccak(
        class_hash=implementation,
        words_len=dst_len,
        words=dst,
        last_input_word=last_word,
        last_input_num_bytes=last_word_num_bytes,
    );

    Stack.push_uint256(code_hash);

    return evm;
}
```

This function calculates the codehash on demand and pushes it to the stack. However, it should be noted that the `EXTCODEHASH` opcode only costs 100 gas. Because the maximum size of the code is 24,576 bytes and hashing 24,576 bytes with the `KECCAK256` opcode costs around 4,600 gas, the `EXTCODEHASH` opcode unfairly consumes a disproportionate amount of computational resources.

Impact

The implementation of the opcode is not efficient, requiring more resources than necessary to be executed.

Recommendations

Since the codehash of a given account only depends on the associated bytecode, it is possible to compute the codehash only once and cache it. The initial computation could happen at the time of account deployment or lazily the first time `EXTCODEHASH` is executed.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1309 ↗.

3.24. EVM sender not relayed by `cairo_message`

Target	precompiles.cairo		
Category	Coding Mistakes	**Severity**	Low
Likelihood	N/A	**Impact**	Informational

Description

The Kakarot EVM allows smart contracts to send a message to L1 via the `send_message_to_l1` function.

The function does not restrict access to `send_message_to_l1`. Therefore, L1 receivers cannot reliably distinguish the address of the EVM sender, since all messages originating from an EVM smart contract will appear to originate from the same L2 address (the Kakarot EVM), and the message contents are entirely controlled by the EVM smart contract sending the message.

Impact

This issue is reported as Informational as Kakarot does not offer the ability to reliably retrieve the address of the EVM smart contract sending a message from the L2 EVM. However, it is likely that some projects will require to authenticate the source of L2-originated messages but have no straightforward way to do so securely.

Recommendations

We encourage to consider implementing a way to retrieve the EVM address of the sender of a message.

This can be implemented by restricting the ability to invoke the `cairo_message` precompile to a specific EVM relayer contract, which prepends (or appends) the `msg.sender` address to the body of the message.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1399 ↗.

The PR in question implemented allowlisting for calling any Kakarot precompile, including `cairo_message`, and an L2 messaging Solidity contract which will be the only contract in the Kakarot EVM allowed to call `cairo_message`. We note that the allowlist is shared between all Kakarot precompiles, therefore care must be taken when adding any contract to the allowlist. The L2 messaging contract includes the address of its caller in the data sent to the L1, allowing the L1 receiver to reliably

retrieve the EVM address of the L2 sender of the message. We also note that this PR removed some sanity checks on the format of the data passed to `cairo_message`, which are not strictly required as only trusted contracts will be allowed to invoke the precompile.

3.25. Missing constraint leading to arbitrary Cairo code execution

Target	eth_transaction.cairo		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Informational

Description

The `eth_transaction.cairo::decode` function lacks a check on the decoded transaction type.

```
func decode{bitwise_ptr: BitwiseBuiltin*, range_check_ptr}(
    tx_data_len: felt, tx_data: felt*
) -> model.EthTransaction* {
    let tx_type = get_tx_type(tx_data);
    tempvar offset = 1 + 3 * tx_type;

    [ap] = bitwise_ptr, ap++;
    [ap] = range_check_ptr, ap++;
    [ap] = tx_data_len, ap++;
    [ap] = tx_data, ap++;
    jmp rel offset;
    call decode_legacy_tx;
    ret;
    call decode_2930;
    ret;
    call decode_1559;
    ret;
}
```

Impact

This issue was independently discovered and remediated by the KKRT Labs team shortly after we started our engagement. The issue was remediated in a commit that appears earlier in the VCS history than the commit under review, which was finalized after the issue was discovered and remediated. The issue is documented for transparency and reported as Informational.

The transaction type is used to compute an offset that is used to perform a low-level jump. The lack of validation could allow an attacker to redirect execution to a semi-arbitrary code location, with potentially critical consequences, including arbitrary code execution.

Recommendations

The issue was independently found and remediated by the KKRT Labs team by validating the transaction type before indexing the jump table.

Remediation

This issue has been acknowledged by KKRT Labs, and a fix was implemented in PR #1217 ↗.

4. Assessment Results

At the time of our assessment, the reviewed code was not deployed on Starknet mainnet.

During our assessment on the scoped Kakarot EVM contracts, we discovered 25 findings. Four critical issues were found. Four were of high impact, six were of medium impact, eight were of low impact, and the remaining findings were informational in nature. All issues have been acknowledged and addressed by KKRT Labs, as documented in the remediation section of each finding.

4.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.

5. Appendices

5.1. Proof of concept: `split_word`

The following code is a proof of concept for finding 3.8. ↗. It shows how a malicious prover can manipulate the result of `split_word` (in this case, by adding `PRIME`). The modifications to the test suite do not cause a verifier error but do cause the `test_should_return_evm_address_in_bytes32` test to fail because the recovered public key is different from the expected one.

```
diff --git a/src/utils/utils.cairo b/src/utils/utils.cairo
index da451281..96a15aa3 100644
--- a/src/utils/utils.cairo
+++ b/src/utils/utils.cairo
@@ -434,6 +434,19 @@ namespace Helpers {

     // @notice Splits a felt into `len` bytes, big-endian, and outputs to
    `dst`.
     func split_word{range_check_ptr}(value: felt, len: felt, dst: felt*) {
+ %{
+ print(f"Len: {ids.len}")
+ print(f"Value: {hex(ids.value)}")
+
+ global tempval
+
+ if ids.len == 32:
+ tempval = int(ids.value) + PRIME
+ else:
+ tempval = int(ids.value)
+
+ print(f"Tempval: {hex(tempval)}")
+ %}
         if (len == 0) {
             assert value = 0;
             return ();
@@ -443,7 +456,9 @@ namespace Helpers {
         let base = 256;
         let bound = 256;
         %{
- memory[ids.output] = res = (int(ids.value) % PRIME) % ids.base
+ global tempval
+
+ memory[ids.output] = res = tempval % ids.base
             assert res < ids.bound, f'split_int(): Limb {res} is out of
    range.'
         %}
         tempvar low_part = [output];
```

5.2. Proof of concept: `load_bytecode`

The following patch provides a basic proof of concept for Finding 3.4. ↗. The proof of concept alters every byte unpacked by `load_bytecode` by adding 1 to it. This modification causes several tests to fail due to invalid bytecode — but no verifier failures.

```
diff --git a/src/kakarot/accounts/library.cairo
    b/src/kakarot/accounts/library.cairo
index 94abf963..17fabdde 100644
--- a/src/kakarot/accounts/library.cairo
+++ b/src/kakarot/accounts/library.cairo
@@ -692,7 +692,8 @@ namespace Internals {
         // Put byte in output and assert that 0 <= byte < bound
         // See math.split_int
         %{
- memory[ids.output] = res = (int(ids.value) % PRIME) % ids.base
+ memory[ids.output] = res = (int(ids.value) % PRIME + 1) % ids.base
             assert res < ids.bound, f'split_int(): Limb {res} is out of
    range.'
         %}
         tempvar a = [output];
```

5.3. Proof of concept: `bytes_used_128`

The following snippet is a basic proof of concept demonstrating Finding 3.20. ↗.

```
diff --git a/src/utils/utils.cairo b/src/utils/utils.cairo
index da451281..9384374c 100644
--- a/src/utils/utils.cairo
+++ b/src/utils/utils.cairo
@@ -685,7 +685,8 @@ namespace Helpers {
         let (output) = alloc();

         %{
- memory[ids.output] = res = (int(ids.value) % PRIME) % ids.base
+ memory[ids.output] = res = 1
             assert res < ids.bound, f'split_int(): Limb {res} is out of
    range.'
         %}
         tempvar x = [output];
```

The above patch causes no validation failure, but it does cause the test `poetry run pytest tests/src/ -m "not NoCI" --log-cli-level=INFO -n logical -vv -k test_should_return_bytes_used_in_128_word` to fail with the following error because the

result of `bytes_used_128` is incorrect.

```
@given(word=st.integers(min_value=0, max_value=2**128 - 1))
@settings(max_examples=20, deadline=None)
def test_should_return_bytes_used_in_128_word(cairo_run, word):
bytes_length = (word.bit_length() + 7) // 8
output = cairo_run(
"test__bytes_used_128",
word=word,
)
> assert bytes_length == output[0]
E assert 0 == 16
E Falsifying example: test_should_return_bytes_used_in_128_word(
E cairo_run=_factory,
E word=0,
E )
E Explanation:
E These lines were always and only run by failing examples:
E /home/vscode/.cache/pypoetry/virtualenvs/kakarot-9tgfeqj9-
    py3.10/lib/python3.10/site-packages/_pytest/assertion/util.py:265
```

5.4. Proof of concept: `load_packed_bytes`

The following snippet is a basic proof of concept demonstrating Finding 3.2. ↗. The proof of concept substitutes the bytes returned by the function with `0x12`. The modification causes no verifier errors, but it does cause several tests to fail due to the incorrect result returned by `load_packed_bytes`.

```
diff --git a/src/utils/utils.cairo b/src/utils/utils.cairo
index da451281..ce617f07 100644
--- a/src/utils/utils.cairo
+++ b/src/utils/utils.cairo
@@ -1178,7 +1178,8 @@ namespace Helpers {
// Put byte in output and assert that 0 <= byte < bound
// See math.split_int
%{
- memory[ids.output] = res = (int(ids.value) % PRIME) % ids.base
+ memory[ids.output] = res = 0x12
             assert res < ids.bound, f'split_int(): Limb {res} is out of
    range.'
         %}
         tempvar a = [output];
```

5.5. Proof of concept: Ignored `RLP.decode` return value

The following patch provides a basic proof of concept demonstrating Finding 3.21. ↗ by modifying an existing test to append the byte `0x11` to a transaction. The test suite does not have failures after this modification, showing how trailing data is accepted while decoding a transaction.

```
diff --git a/tests/src/utils/test_eth_transaction.py
    b/tests/src/utils/test_eth_transaction.py
index 62d99883..6dbc690c 100644
--- a/tests/src/utils/test_eth_transaction.py
+++ b/tests/src/utils/test_eth_transaction.py
@@ -15,6 +15,13 @@ class TestEthTransaction:
self, cairo_run, transaction
):
encoded_unsigned_tx = rlp_encode_signed_data(transaction)
+            if type(encoded_unsigned_tx) == list:
+                encoded_unsigned_tx.append(0x11)
+            elif type(encoded_unsigned_tx) == bytes:
+                encoded_unsigned_tx = encoded_unsigned_tx + b"\x11"
+            else:
+                pass
+
           decoded_tx = cairo_run(
               "test__decode",
               data=list(encoded_unsigned_tx),
```

If an assertion on the return value of `RLP.decode` calls is added, tests fail because `n_items` is 2, not 1, demonstrating how `RLP.decode` is decoding additional data. The following patch demonstrates this for one of the three `RLP.decode` calls:

```
diff --git a/src/utils/eth_transaction.cairo b/src/utils/eth_transaction.cairo
index a957c4ef..2734a898 100644
--- a/src/utils/eth_transaction.cairo
+++ b/src/utils/eth_transaction.cairo
@@ -81,7 +81,8 @@ namespace EthTransaction {
alloc_locals;

         let (items: RLP.Item*) = alloc();
-        RLP.decode(items, tx_data_len - 1, tx_data + 1);
+        let n_items = RLP.decode(items, tx_data_len - 1, tx_data + 1);
+        assert n_items = 1;
         let sub_items_len = [items].data_len;
         let sub_items = cast([items].data, RLP.Item*);
```