code4rena

Kakarot

Smart Contract
Security Assessment

Audit dates: Sep 27 — Oct 25, 2024

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit outlined in this document, C4 conducted an analysis of the Kakarot smart contract system written in Cairo. The audit took place between September 27 — October 25, 2024.

Following the C4 audit, Zenith team RadiantLabs (3docSec and EV_om) reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit report.

Wardens

23 Wardens contributed reports to Kakarot:

1. RadiantLabs (3docSec and EV_om)
2. g_x_w
3. 0xastronatey
4. gumgumzum
5. muellerberndt
6. oakcobalt
7. Bauchibred
8. ahmedaghadi
9. fyamf
10. minglei-wang-3570
11. Emmanuel
12. Davymutinda77
13. AshutoshSB
14. 20centclub (tpiliposian, Skylice and M3talDrag0n)
15. superpozycja
16. zhaojie
17. ulas
18. 1AutumnLeaf777
19. wasm_it (0xrex and rustguy)

This audit was judged by LSDan.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 17 unique vulnerabilities. Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity and 11 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 13 reports detailing issues with a risk rating of LOW severity or non-critical.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Kakarot repository, and is composed of 63 smart contracts written in the Cairo programming language and includes 15398 lines of Cairo code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (6)

[H-01] Unauthorized contracts can bypass precompile authorization via `delegatecall` in Kakarot zkEVM

Submitted by 0xastronatey, also found by RadiantLabs

In the Kakarot zkEVM implementation `exec_precompile`, unauthorized contracts can bypass the precompile authorization mechanism using the `delegatecall`. This allows unauthorized contracts to execute privileged precompile functions intended only for authorized (whitelisted) contracts, compromising the security and integrity of the system.

Vulnerability Details:

Let's take a look at the `exec_precompile` function with the unessential part omitted for brevity:

```
func exec_precompile{
      syscall_ptr: felt*,
      pedersen_ptr: HashBuiltin*,
      range_check_ptr,
      bitwise_ptr: BitwiseBuiltin*,
  }(
      precompile_address: felt,
      input_len: felt,
      input: felt*,
      caller_code_address: felt,
      caller_address: felt,
  ) -> (output_len: felt, output: felt*, gas_used: felt, reverted: felt) {

      //SNIP...
      // Authorization for Kakarot precompiles
      let is_kakarot_precompile =
PrecompilesHelpers.is_kakarot_precompile(precompile_address);
      if is_kakarot_precompile != 0 {
          // Check if caller is whitelisted
          let is_whitelisted =
KakarotPrecompiles.is_caller_whitelisted(caller_code_address);
          if is_whitelisted == FALSE {
              jmp unauthorized_call;
          }
          // Proceed with precompile execution
          // ...
      } else {
          jmp unauthorized_call;
      }
  }
```

The issue arises because the authorization mechanism relies solely on `caller_code_address`, which refers to the code being executed, and does not consider `caller_address`; the actual address of the contract making the call. This approach is insufficient and can be exploited using `delegatecall`.

If an EVM contract A `delegatecall`'s EVM contract B, and B is `allowlisted`, then the call to the Cairo precompile succeeds because the `allowlist` is based on the code address (of B) and not on the execution address (of A).

This means that an unauthorized contract (Contract A) can exploit this behavior to execute privileged operations by delegatecalling a whitelisted contract (Contract B) and passing the authorization check incorrectly.

As such, it is an issue for other whitelisted smart contracts allowing arbitrary calls to external smart contracts. For example, this is the case of some smart contracts allowing callbacks. In those situations, a malicious user could make a DEX execute a call to the malicious smart contract that would be able to access the precompiles pretending to be the DEX.

Impact

This issue could allow unauthorized contracts to perform operations that should be restricted to whitelisted contracts like in the case of DEX that supports callbacks and has been added to the `allowistlist`. Moreover, all contracts within the `allowistlist` must be written with the assumption that they can also be `delegatecalled`; i.e., they must not make any assumptions about their storage or their own EVM address (`address(this)`).

Prior to my discussion with the sponsor in a private thread about the implications of this attack surface, the sponsor intended to eventually remove the whitelist mechanism, not fully considering the dangers of delegatecalls to precompiles, which would have been a bad idea.

With the team's plan to eventually remove the `allowlist`, prior to knowing about this attack surface, it gives any contract the opportunity to exploit this on KakarotEVM; which, as we've seen, can be catastrophic; especially in the case of MoonBeam's `msg.sender` impersonation disclosure by pwning.eth.

Recommendation

In the interim, the obvious mitigation is to ensure that all whitelisted contracts are written with the assumption that they can also be `delegatecalled`. And more importantly, the whitelist mechanism should never be removed, as this would open a floodgate of vulnerabilities to the protocol.

A permanent solution to this issue would be to disable `Delegatecalls` for custom precompiles.

Assessed type

`call/delegatecall`

ClementWalter (Kakarot) confirmed and commented:

Severity: High

Kakarot mitigated:

This PR fixes the issue raised during the audit about eventual honeypots that could lure a user and manipulate starknet tokens without explicit approvals:

1. Remove dependence on `the code_address for` whitelisting, which were reported dangerous.
2. Disable ability of having nested a delegatecall to `0x75001` at protocol level. (no longer possible to do User A `-call->` contract C `-delegatecall->DualVmToken -delegatecall->0x75001` as a byproduct of 1.
3. Added `noDelegateCall` modifier to L2KakarotMessaging and DualVMToken for extra security.
4. Added associated tests under Security.

Status: Mitigation confirmed.

[H-02] Prover can cheat in `felt_to_bytes_little` due to value underflow

Submitted by muellerberndt, also found by superpozycja and RadiantLabs

The function `felt_to_bytes_little()` in `bytes.cairo` converts a felt into an array of bytes.

The prover can cheat by returning a near arbitrary string that does not correspond to the input felt, whereby, the spoofed output bytes and `bytes_len, bytes must fulfill some specific conditions (but, if carefully crafted, can contain almost arbitrary sequences of bytes).

This issue affects the function `felt_to_bytes_little()` as well as other functions that depend on it:

```
    - felt_to_bytes()
      - uint256_to_bytes_little()
      - uint256_to_bytes()
      - uint256_to_bytes32()
      - bigint_to_bytes_array()
```

Those functions are used throughout the code, notably in `get_create_address()` and `get_create2_address()`, which an attacker could exploit to deploy L2 smart contracts from a spoofed sender address (e.g., to steal funds from wallets that use account abstraction).

Details

First, note the following loop:

```
        body:
          let range_check_ptr = [ap - 3];
          let value = [ap - 2];
          let bytes_len = [ap - 1];
          let bytes = cast([fp - 4], felt*);
          let output = bytes + bytes_len;
          let base = 2 ** 8;
          let bound = base;

          %{
              memory[ids.output] = res = (int(ids.value) % PRIME) % ids.base
              assert res < ids.bound, f'split_int(): Limb {res} is out of
range.'
          %}
          let byte = [output];
          with_attr error_message("felt_to_bytes_little: byte value is too
big") {
              assert_nn_le(byte, bound - 1);
          }
          tempvar value = (value - byte) / base;

          tempvar range_check_ptr = range_check_ptr;
          tempvar value = value;
          tempvar bytes_len = bytes_len + 1;

          jmp body if value != 0;
```

We observe that:

1. Value can underflow if the byte returned in the last iteration is greater than the value remaining in the felt. For example, if the remaining value is 1 but the prover/hint returns 2, value will underflow into `STARKNET_PRIME - 1`. Consequently, the loop will continue running as `value != 0`.
2. In the following iterations of the loop, the prover can return arbitrary values, they just need to ensure that value eventually becomes 0. They can use as many iterations as required, but it is important that `bytes_len` ends up at a specific value (see 3).
3. Finally, at the end of the function, there is a check that `bytes_len` is the minimal one possible to represent the value. The lower bound and upper bound for this check is read from `pow256_table` at the offset `bytes_len`. The malicious prover must ensure that the value at this memory address (somewhere into the code segment) is lower than `initial_value`. Any nearby position where the Cairo bytecode contains a zero works.

Proof of Concept

To pass the bounds check, we need a code offset that contains the value `0`. Fortunately, from a malicious prover's perspective, there are many zero-value locations in the code segment. When we dump the memory segment with a hint we find multiple zeroes:

```
    # Offset:content

      46: 5210805298050138111
      47: 5192296858534827628530496329220096
      48: 6
      49: 5189976364521848832
      50: 5193354047062507520
      51: 5189976364521848832
      52: 5193354004112834560
      53: 5191102242953854976
      54: 5198983563776458752
      55: 290341444919459839
      56: 5210805478438109184
      57: 5191102225773985792
      58: 2
      59: 0
      60: 2
      61: 0
      62: 51
      63: 1
      64: 0
      65: 0
      66: 0
      67: 116
      68: 2
      69: 0
      70: 0
      71: 0
      72: 186
      73: 0
      74: 0
      75: 0
      76: 4
```

When creating the proof of concept I had some problems calculating the offsets reliably, but I got it to work with a few brute-force tries. The following test worked reliably for me:

```
class TestFeltToBytesLittle:

      def test_felt_to_bytes_exploit(self, cairo_program, cairo_run):

          hint = f"if ids.bytes_len < 135:\n  memory[ids.output] = 2\n"\
              f"elif ids.bytes_len == 135:\n  ids.base == 2 ** 8\n
memory[ids.output] = res = (int(ids.value) % PRIME) % ids.base\n"\
              "else:  memory[ids.output] = res = (int(ids.value) % PRIME) %
ids.base\n  "

          with (
              patch_hint(
                  cairo_program,
                  "memory[ids.output] = res = (int(ids.value) % PRIME) %
ids.base\nassert res < ids.bound, f'split_int(): Limb {res} is out of
range.'",
                  hint,
              ),
          ):

              output = cairo_run("test__felt_to_bytes_little", n=1)
              expected = int.to_bytes(1, 1, byteorder="little")

              assert(bytes(output) == expected)
```

Running the test shows the prover's manipulated output vs. the expected output:

```
    FAILED
tests/src/berndt/test_berndt.py::TestFeltToBytesLittle::test_felt_to_bytes_exp
loit - AssertionError: assert b'\x02\x02\x0...8\xb8\xb8\x04' == b'\x01'

        At index 0 diff: b'\x02' != b'\x01'

        Full diff:
        - b'\x01'
        + (b'\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02'
        +  b'\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02'
        +  b'\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02'
        +  b'\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02'
        +  b'\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02'
        +  b'\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02'
        +  b'\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02'
        +  b'\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02\x02'
        +
b'\x02\x02\x02\x02\x02\x02\x02\xe6\x0f@@@@@\xd4~4\x84\xc4RQ\xa1\xf6\xcb'
        +  b'\xac\xad\x99\x95\x15\x96\xd6\xc6S\xba\xb8\xb8\xb8\xb8\x04')
```

A skilled attacker who invests enough time can almost arbitrary output strings, as long as they make sure that `value` reaches zero at some point and `pow256_table + bytes_len] == 0`.

P.S., whether the same offset works may depend on the Cairo compiler version. I used the following test to find a valid offset:

```
        def test_should_return_bytes(self, cairo_program, cairo_run):

              stop = 134

              while 1:
                  print("Trying stop = " + str(stop))

                  hint = f"if ids.bytes_len < {stop}:\n  memory[ids.output] =
2\n"\
                      f"elif ids.bytes_len == {stop}:\n  ids.base == 2 ** 8\n
memory[ids.output] = res = (int(ids.value) % PRIME) % ids.base\n"\
                      "else:  memory[ids.output] = res = (int(ids.value) %
PRIME) % ids.base\n  "

                  with (
                      patch_hint(
                          cairo_program,
                          "memory[ids.output] = res = (int(ids.value) % PRIME)
% ids.base\nassert res < ids.bound, f'split_int(): Limb {res} is out of
range.'",
                          hint,
                      ),
                  ):
                      with cairo_error(message="bytes_len is not the minimal
possible"):
                          output = cairo_run("test__felt_to_bytes_little",
n=1)
                          expected = int.to_bytes(1, 1, byteorder="little")

                      stop += 1
```

Recommended Mitigation Steps

The easiest fix is to do range checks on `value` to prevent underflows.

Assessed type

Math

ClementWalter (Kakarot) confirmed and commented:

Severity: High

Kakarot mitigated:

This PR fixes `felt_to_bytes_little` loop stop condition. The loop will stop at 31 bytes.

Status: Mitigation confirmed.

[H-03] Missing constraint in `default_dict_copy`

Submitted by g_x_w

https://github.com/kkrt-labs/kakarot/blob/7411a5520e8a00be6f5243a50c160e66ad285563/src/kakarot/account.cairo#L83

https://github.com/kkrt-labs/kakarot/blob/7411a5520e8a00be6f5243a50c160e66ad285563/src/utils/dict.cairo#L83

Vulnerability details

In CairoZero, the correct usage of dict objects created via `default_dict_new` must be paired with a call to `default_dict_finalize` to ensure the integrity and prevent malicious prover's manipulation of its contents. However, this constraint is missing in the handling of `transient_storage`, `storage` and `valid_jumpdests`, leading to severe vulnerabilities when executing smart contracts.

Description of the Issue

According to CairoZero's documentation (link to `default_dict`), a proper workflow involving `default_dict_new` includes a finalization step using `default_dict_finalize`. This ensures the correct initialization of dictionary elements and prevents malicious provers from manipulating dictionary values through hints. Specifically, `default_dict_finalize` enforces the constraint that the initial value of the first element's `prev_value` in the dictionary must equal to the `default_value`.

However, in the case of `transient_storage`, `storage` and `valid_jumpdests`, this crucial constraint is missing. I will illustrate this issue using `transient_storage`. First, `transient_storage` is initialized in `Account.init()` as follows:

```
let (transient_storage_start) = default_dict_new(0);
```

However, there is no subsequent call to `default_dict_finalize(transient_storage_start, transient_storage, 0)` to finalize the storage. Instead, the function `default_dict_copy()` is called on `transient_storage` multiple times during a transaction through the `Account.copy()` function:

```
let (transient_storage_start, transient_storage) = default_dict_copy(
      self.transient_storage_start, self.transient_storage
  );
```

This copy operation starts by calling `dict_squash` on the original `transient_storage`:

```
func default_dict_copy{range_check_ptr}(start: DictAccess*, end: DictAccess*)
-> (
      DictAccess*, DictAccess*
```

```
  ) {
      alloc_locals;
      let (squashed_start, squashed_end) = dict_squash(start, end);
      local range_check_ptr = range_check_ptr;
      let dict_len = squashed_end - squashed_start;

      local default_value;
      if (dict_len == 0) {
          assert default_value = 0;
      } else {
          assert default_value = squashed_start.prev_value;
      }

      let (local new_start) = default_dict_new(default_value);
      ...
```

`dict_squash` itself does not assert the `prev_value` of the first element in the dictionary. As a result, the subsequent copied `transient_storage`'s initial value is also under the malicious prover's control.

```
    let (local new_start) = default_dict_new(default_value);
```

If we look at the source code of `default_dict_finalize`, we will notice an extra constraint in the `default_dict_finalize_inner` function:

```
func default_dict_finalize{range_check_ptr}(
      dict_accesses_start: DictAccess*, dict_accesses_end: DictAccess*,
default_value: felt
  ) -> (squashed_dict_start: DictAccess*, squashed_dict_end: DictAccess*) {
      alloc_locals;
      let (local squashed_dict_start, local squashed_dict_end) = dict_squash(
          dict_accesses_start, dict_accesses_end
      );
      local range_check_ptr = range_check_ptr;

      default_dict_finalize_inner(
          dict_accesses_start=squashed_dict_start,
          n_accesses=(squashed_dict_end - squashed_dict_start) /
DictAccess.SIZE,
          default_value=default_value,
      );
      return (squashed_dict_start=squashed_dict_start,
squashed_dict_end=squashed_dict_end);
  }

  func default_dict_finalize_inner(
      dict_accesses_start: DictAccess*, n_accesses: felt, default_value: felt
  ) {
      ...
      assert dict_accesses_start.prev_value = default_value;
      ...
  }
```

As shown above, the additional check besides `dict_squash` is:

```
assert dict_accesses_start.prev_value = default_value;
```

This constraint ensures that any uninitialized read from the dictionary returns the correct default value (`0` in this case). However, in `default_dict_copy`, this constraint is absent; meaning the `prev_value` for the first dictionary entry of `transient_storage` is not guaranteed to match the expected default value.

Impact

A malicious prover could manipulate the value read from `transient_storage`, `storage` and `valid_jumpdests`. Specifically, they could fabricate a proof where uninitialized keys in the dictionary return values other than the intended default (`0`). This could lead to unintended or unauthorized access to funds, manipulation of contract state, or other security breaches depending on the logic in the upper-level EVM contract.

Assessed type

Invalid Validation

ClementWalter (Kakarot) confirmed and commented:

Severity: High

ClementWalter (Kakarot) commented:

After another round of review from Zellic, it appears that this is invalid (went to fast in the validation of the finding) as the `default_value` is asserted below in the loop

https://github.com/kkrt-labs/kakarot/commit/474951c7babbd57f06d074aa15ba7a2f1911af21#diff-18450849ea8d97868875c0b5eeb0934a496dbf959c60a5bcf9273ff5f2bbc298L117

1. We pick as default value the first `prev_value`.
2. We consider that it's the default value for the whole dict.
3. We squash and copy the squashed dict into a new `default_dict` with the given `default_value`.
4. In the loop, we assert that the `squashed.prev_value == default_value`.

EV_om (Zenith) commented:

@ClementWalter, we picked a long time at this finding and were under the impression that it was valid. It would be great to get your take on the below POC which we believe would work:

- Construct a transaction the attempts to copy a dictionary with no subsequent reads from the same key.
- Modify all `.prev_values` so the dictionary can be successfully squashed (the prover has the ability to do this).
- `default_value` of the new dict is set to our modified `prev_value`.
- The check on L117 always passes.
- In the next call stack, reads will return the modified value.

[ClementWalter (Kakarot) commented]():

I don't get the first part:

- Construct a transaction the attempts to copy a dictionary with no subsequent reads from the same key
- Modify all `.prev_values` so the dictionary can be successfully squashed (the prover has the ability to do this)

What dict squashing does is that it enforces the fact that the sequence of `DictAccess*` (filled by the prover at their will) is actually a valid sequence of dict read and write.

So the `dict_squash` part is fine and after it, it's guaranteed that the dict was correctly used, but not yet that it was a default dict with a given value.

Then we pick the first prev as `default_value` and we copy the squash dict into a new dict, asserting in the mean time that all the `prev_values` are actually equal to the one picked.

Eventually, it means that

1. The `DictAccess*` was a legit dict.
2. All the `prev_values` were equal to the same value.

Maybe you can try to craft a failing test case?

[EV_om (Zenith) commented]():

@ClementWalter, a full POC would take a bit of time, but maybe @3docSec has something lying around.

I don't get the first part

- Construct a transaction the attempts to copy a dictionary with no subsequent reads from the same key.

You're right, this may not be necessary. Squashing prevents modifying the `prev_value` of an arbitrary entry if there are writes after a read (since it could be constrained to a previously written value), but it should still be possible to modify the default `prev_value`.

Otherwise, as the finding claims:

Specifically, `default_dict_finalize` enforces the constraint that the initial value of the first element's `prev_value` in the dictionary must equal to the `default_value`.

This was never enforced in the code in scope. So we did have that:

1. The `DictAccess*` was a valid sequence of dict read and write.
2. All the prev_values were equal to the same value.

But it was not verified that:

3. All the `prev_values` were equal to the original dictionary's default value, as they should be after squashing.

[3docSec (Zenith) commented]:

Just adding that if we look at the code:

```
  if (dict_len == 0) {
        assert default_value = 0;
    } else {
        assert default_value = squashed_start.prev_value;
    }
```

In the `else` branch, `squashed_start.prev_value` is the free bird that is never validated: if it was asserted to be `0` in any place, the code would've been safe.

[ClementWalter (Kakarot) commented]:

There is a list of`DictAccess*`, filled initially at the prover's will (prover can put any values in these slots in `dict_read` and `dict_write`.

Now squashing enforces that this "random" list is actually a valid list of `DictAccess*`, properly logging read and write of a regular dict (precisely that prev value is always sound with current value).

The "default" part of the dict is ignored in the initial `squash_dict` where the prover could still have populated the dict with any initial values (possibly all different).

From this squash dict, the code pick the first `prev_value` (whatever it is is up to the prover).

Then, it enforces that all the prev*keys are actually of this same prev*value, making it actually a `default_dict(prev_value)`.

At this point the `prev_value` is up to the prover, but it's guaranteed that it's still a `default_dict`.

Note that this function is `default_dict_*copy*_`, so the purpose is only to copy a dict with the same default value, not to finalize it. At other places in the code we do use `default_dict_finalize(0)` to actually enforce the default values are 0.

Now, the issue may still be relevant if at some places, we don't call this `default_dict_finalize`, which actually happens (!) for the account's storage and `valid_jumpdest`.

So I guess that the issue is valid, not because the method itself, but because of missing `default_dict_finalize` for the account's dict.

Kakarot mitigated:

PR fix: `default_dict_copy` finalize with default value `0`.

Status: Mitigation confirmed.

[H-04] Non-finalized dictionary in RIPEMD160 allows forging of output

Submitted by RadiantLabs

When `ripemd160.finish()` does not enter the `if (next_block == FALSE)` condition at L456, the dictionary `x` initialized at the beginning of the function is not finalized. Instead, `x` is reassigned to reference a new dictionary at L470:

```
File: ripemd160.cairo
  456:     if (next_block == FALSE) {
                          ...
  462:         default_dict_finalize(start, x, 0);
  463:         let (res, rsize) = compress(buf, bufsize, arr_x, 16);
  464:         return (res=res, rsize=rsize);
  465:     }
  466:     let (local arr_x: felt*) = alloc();
  467:     dict_to_array{dict_ptr=x}(arr_x, 16);
  468:     let (buf, bufsize) = compress(buf, bufsize, arr_x, 16);
  469:     // reset dict to all 0.
  470:     let (x) = default_dict_new(0);
```

Because the old dictionary is never finalized, it is possible to insert incorrect values for read operations on the old dictionary, which allows proving an incorrect output for any input of `size > 55` (56 with the fix to our separate vulnerability on this value).

Read operations on the old dictionary are performed in `ripemd160::absorb_data`:

```
File: ripemd160.cairo
  148: func absorb_data{range_check_ptr, bitwise_ptr: BitwiseBuiltin*,
dict_ptr: DictAccess*}(
  149:     data: felt*, len: felt, index: felt
  150: ) {
  151:     alloc_locals;
  152:     if (index - len == 0) {
  153:         return ();
  154:     }
  155: 
  156:     let (index_4, _) = unsigned_div_rem(index, 4);
  157:     let (index_and_3) = uint32_and(index, 3);
  158:     let (factor) = uint32_mul(8, index_and_3);
  159:     let (factor) = pow2(factor);
  160:     let (tmp) = uint32_mul([data], factor);
  161:     let (old_val) = dict_read{dict_ptr=dict_ptr}(index_4);
  162:     let (val) = uint32_xor(old_val, tmp);
  163:     dict_write{dict_ptr=dict_ptr}(index_4, val);
  164:
  165:     absorb_data{dict_ptr=dict_ptr}(data + 1, len, index + 1);
  166:     return ();
  167: }
```

Proof of Concept

The below test case can be added to `test_ripemd160.py` to demonstrate the vulnerability:

```
async def test_ripemd160_output_can_be_forged(self, cairo_program, cairo_run):
        msg_bytes = bytes([0x00] * 57)
        with (
                patch_hint(
                        cairo_program,
                        "vm_enter_scope()",
                        "try:\n"
                        "    dict_tracker =
__dict_manager.get_tracker(ids.dict_ptr)\n"
                        "    dict_tracker.data[ids.index_4] = 1\n"
                        "except Exception: pass\n"
                        "vm_enter_scope()"
                )
        ):
                precompile_hash = cairo_run("test__ripemd160",
msg=list(msg_bytes))

                # Hash with RIPEMD-160 to compare with precompile result
                ripemd160_crypto = RIPEMD160.new()
                ripemd160_crypto.update(msg_bytes)
                expected_hash = ripemd160_crypto.hexdigest()

                assert expected_hash.rjust(64, "0") !=
bytes(precompile_hash).hex()
```

The test requires adding the following “noop” hints below [`ripemd160.cairo#L160`] so we can insert the malicious hint:

```
    %{ vm_enter_scope() %}
      %{ vm_exit_scope() %}
```

The patched hint modifies the in-memory dictionary of the prover, which results in the `dict_read` operation in the following line returning an incorrect fixed value. Because a `dict_read` call is really an append operation, and the new values aren't checked against the previous ones if the dictionary is not squashed, this results in a forged output that can be proven to be correct.

Recommended Mitigation Steps

Call `default_dict_finalize(start, x, 0);` before `let (x) = default_dict_new(0);`.

ClementWalter (Kakarot) confirmed and commented:

Severity: High

Kakarot mitigated:

PR fix: finalize dictionary in `RIPEMD160`. Finalizes the `dict` and reassign to start after resetting the `dict`.

Status: Mitigation confirmed.

[H-05] `RIPEMD-160` precompile yields wrong hashes for large set of inputs due to off-by-one error

Submitted by RadiantLabs, also found by gumgumzum

The `RIPEMD-160` digest integrates the input data with one additional block including the message length.

If we look at the Cairo code that achieves this:

```
File: ripemd160.cairo
    let next_block = is_nn_le(55, len);
    if (next_block == FALSE) {
        dict_write{dict_ptr=x}(14, val);
        dict_write{dict_ptr=x}(15, val_15);

        let (local arr_x: felt*) = alloc();
        dict_to_array{dict_ptr=x}(arr_x, 16);
        default_dict_finalize(start, x, 0);
        let (res, rsize) = compress(buf, bufsize, arr_x, 16);
        return (res=res, rsize=rsize);
    }
    let (local arr_x: felt*) = alloc();
    dict_to_array{dict_ptr=x}(arr_x, 16);
    let (buf, bufsize) = compress(buf, bufsize, arr_x, 16);
```

… we see that the `if (next_block == false)` executes one block of code in case `len >= 55`, and the other otherwise.

If we compare this check with the [equivalent implementation in Go (used by Geth)](), for example:

```
        if tc%64 < 56 {
                d.Write(tmp[0 : 56-tc%64])
        } else {
                d.Write(tmp[0 : 64+56-tc%64])
        }
```

We see that this time, the selection is made with `len < 56` as discriminator; if we imagine swapping the `if` and the `else` blocks of the Go implementation, the condition becomes `len >= 56`.

So for the edge case of `len == 55`, Kakarot and Geth will take different actions, which, as shown in the PoC, results in differing hashes. `len` here represents the length of the input modulo 64, hence the `RIPEMD-160` precompile will yield wrong outputs for inputs of length `55 + k*64`.

Because this precompile is a cryptographic hash function, it can be used in many ways by EVM applications, including access control on funds.

Proof of Concept

To prove the point, the following test can be added to the `test_ripemd160.py` unit test suite:

```
    async def test_ripemd160_on_55_length_input(self, cairo_run):
          msg_bytes =
bytes("abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq", "ascii")
          precompile_hash = cairo_run("test__ripemd160", msg=list(msg_bytes))

          # Hash with RIPEMD-160 to compare with precompile result
          ripemd160_crypto = RIPEMD160.new()
          ripemd160_crypto.update(msg_bytes)
          expected_hash = ripemd160_crypto.hexdigest()

          assert expected_hash.rjust(64, "0") == bytes(precompile_hash).hex()
```

… which yields the following output:

```
    >       assert expected_hash.rjust(64, "0") ==
bytes(precompile_hash).hex()
      E       AssertionError: assert '000000000000...65c8fd54fbafd' ==
'000000000000...169718d24abaf'
      E
      E         -
000000000000000000000000578b4715e07320701e0715dc640169718d24abaf
      E         +
000000000000000000000000df3166bed54ffac595c474af28465c8fd54fbafd

      tests/src/kakarot/precompiles/test_ripemd160.py:19: AssertionError
```

Recommended Mitigation Steps

Change the check at L455 to `is_nn_le(56, len)`.

ClementWalter (Kakarot) confirmed and commented:

Severity: High

Kakarot mitigated:

This PR fixes off by one error `ripemd-160`.

Status: Mitigation confirmed.

[H-06] Three valid signatures for the same message

Submitted by fyamf, also found by 0xastronatey, Davymutinda77, and AshutoshSB

On Ethereum, it's possible to create **two** different valid signatures for the same message (known as ECDSA Signature Malleability). However, on Kakarot, you can create **three** different valid signatures for the same message. This happens because, on Ethereum, if the value of `s` is higher than the maximum valid range, it returns `address(0)`. On Kakarot, in this case, it returns a valid address.

Proof of Concept

If `r` and `v` are in the valid range, and `s` is between `1` and `secp256k1n`, the `ecrecover` function behaves the same on both Ethereum and Kakarot.

- On Ethereum, if the value of `s` is higher than `secp256k1n`, it returns `address(0)`.
- On Kakarot, if the value of `s` is higher than `secp256k1n`, it returns an address equivalent to `s + secp256k1n`.

Here's an example for better clarity. The `ecrecover` function for the following inputs will return the same result `0x992FEaf0E360bffaeAcf57c28D2a3F5059Fe0982`:

- On Ethereum:
    - input1: `r = 1, v = 27, s = 2`
    - input2: `r = 1, v = 28, s = secp256k1n - 2 = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD036413F`
- On Kakarot:

- input1: `r = 1, v = 27, s = 2`
- input2: `r = 1, v = 28, s = secp256k1n - 2 = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD036413F`
- input3: `r = 1, v = 27, s = secp256k1n + 2 = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364143`

While inputs 1 and 2 behave as expected, input 3 highlights the problem. With input 3, Ethereum's `ecrecover` returns `address(0)`, but Kakarot returns the same result as input1 and input2.

This shows that a malleability attack on Kakarot can be done in two ways, whereas on Ethereum, it can only be done in one.

In simple terms, on Ethereum, a malleability attack can be done by changing the `v` from `27` to `28` (or vice versa) and swapping `s` with `secp256k1n - s`:

- `r, s, v`
- `r, secp256k1n - s, v = {27 -> 28, 28 -> 27}`

On Kakarot, the malleability attack can be done in two ways: either by changing `s` to `secp256k1n + s` (as long as it's not larger than `2**256 - 1`) or using the same method as Ethereum by changing the `v` and swapping `s` with `secp256k1n - s`:

- `r, s, v`
- `r, secp256k1n - s, v = {27 -> 28, 28 -> 27}`

OR:

- `r, s, v`
- `r, secp256k1n + s, v`

The value of `s` needs to be limited in the `ec_recover::run` function to make sure it's within the valid range.

```
    func run{
          syscall_ptr: felt*,
          pedersen_ptr: HashBuiltin*,
          range_check_ptr,
          bitwise_ptr: BitwiseBuiltin*,
      }(_address: felt, input_len: felt, input: felt*) -> (
          output_len: felt, output: felt*, gas_used: felt, reverted: felt
      ) {
          alloc_locals;

          let (input_padded) = alloc();
          slice(input_padded, input_len, input, 0, 4 * 32);

          let v_uint256 = Helpers.bytes32_to_uint256(input_padded + 32);
          let v = Helpers.uint256_to_felt(v_uint256);
          if ((v - 27) * (v - 28) != 0) {
              let (output) = alloc();
              return (0, output, GAS_COST_EC_RECOVER, 0);
          }

          let msg_hash = Helpers.bytes_to_uint256(32, input_padded);
          let r = Helpers.bytes_to_uint256(32, input_padded + 32 * 2);
          let s = Helpers.bytes_to_uint256(32, input_padded + 32 * 3);

          // v - 27, see recover_public_key comment
          let (helpers_class) = Kakarot_cairo1_helpers_class_hash.read();
          let (success, recovered_address) =
ICairo1Helpers.library_call_recover_eth_address(
              class_hash=helpers_class, msg_hash=msg_hash, r=r, s=s,
y_parity=v - 27
          );

          if (success == 0) {
              let (output) = alloc();
              return (0, output, GAS_COST_EC_RECOVER, 0);
          }

          let (output) = alloc();
          memset(output, 0, 12);
          Helpers.split_word(recovered_address, 20, output + 12);

          return (32, output, GAS_COST_EC_RECOVER, 0);
      }
```

https://github.com/kkrt-labs/kakarot/blob/7411a5520e8a00be6f5243a50c160e66ad285563/src/kakarot/precompiles/ec_recover.cairo#L40

PoC

In the following test, three different combinations of `(r, s, v)` are provided as parameters to the `ecrecover` function, and the result for all of them is the same.

To run the end-to-end tests correctly, the `make test-end-to-end14` command should be used, as defined in the `Makefile`.

```
    test-end-to-end14: deploy
        uv run pytest tests/end_to_end/Simple/test_ecrecover_wrong.py -m "not
slow" -s --seed 42
```

```
import pytest
  import pytest_asyncio

  from kakarot_scripts.utils.kakarot import deploy


  @pytest_asyncio.fixture(scope="package")
  async def target():
      return await deploy(
          "Simple",
          "TestEcrecoverWrong",
      )

  # 0xfffFFFFFFFFFFFFFFFFFFFFFFFFFFFFebaaedce6af48a03bbfd25e8cd0364140

  @pytest.mark.asyncio(scope="package")
  class Test:


      class TestEcrecoverWrong1:
          async def test_ecrecover_wrong1(self, target):
              # r = 1
              # s =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364143 =
secp256k1n + 2
              # v = 27
              addr = await target.call1(1,
115792089237316195423570985008687907852837564279074904382605163141518161494339
, 27)
              print("s is secp256k1n + 2, v is 27: " + str(addr))


      class TestEcrecoverWrong2:
          async def test_ecrecover_wrong2(self, target):
              # r = 1
              # s = 2
              # v = 27
              addr = await target.call1(1, 2, 27)
              print("s is 2, v is 27: " + str(addr))

      class TestEcrecoverWrong3:
          async def test_ecrecover_wrong3(self, target):
              # r = 1
              # s =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD036413F =
secp256k1n - 2
```

```
              # v = 28
              addr = await target.call1(1,
115792089237316195423570985008687907852837564279074904382605163141518161494335
, 28)
              print("s is secp256k1n - 2, v is 28: " + str(addr))
```

```
pragma solidity ^0.8.0;

  contract TestEcrecoverWrong {
      function call1(uint256 _r, uint256 _s, uint256 _v) public view returns
(address) {
          return ecrecover(bytes32(uint256(0)), uint8(_v), bytes32(_r),
bytes32(_s));
      }
  }
```

The output log is:

```
INFO:kakarot_scripts.utils.kakarot:⏳ Deploying TestEcrecoverWrong
  INFO:kakarot_scripts.utils.kakarot:✅ TestEcrecoverWrong deployed at:
0x9fE46736679d2D9a65F0992F2272dE9f3c7fa6e0
  s is secp256k1n + 2, v is 27: 0x992FEaf0E360bffaeAcf57c28D2a3F5059Fe0982
  .s is 2, v is 27: 0x992FEaf0E360bffaeAcf57c28D2a3F5059Fe0982
  .s is secp256k1n - 2, v is 28: 0x992FEaf0E360bffaeAcf57c28D2a3F5059Fe0982
```

Recommended Mitigation Steps

The following modifications are needed to constraint the valid range of s:

```
    //.......
      from starkware.cairo.common.uint256 import Uint256,
uint256_reverse_endian
  +   from starkware.cairo.common.math_cmp import is_not_zero, is_nn
      from starkware.cairo.common.cairo_secp.bigint import bigint_to_uint256
      //........

      func run{
          syscall_ptr: felt*,
          pedersen_ptr: HashBuiltin*,
          range_check_ptr,
          bitwise_ptr: BitwiseBuiltin*,
      }(_address: felt, input_len: felt, input: felt*) -> (
          output_len: felt, output: felt*, gas_used: felt, reverted: felt
      ) {
          alloc_locals;

          let (input_padded) = alloc();
          slice(input_padded, input_len, input, 0, 4 * 32);

          let v_uint256 = Helpers.bytes32_to_uint256(input_padded + 32);
          let v = Helpers.uint256_to_felt(v_uint256);

          if ((v - 27) * (v - 28) != 0) {
              let (output) = alloc();
              return (0, output, GAS_COST_EC_RECOVER, 0);
          }

          let msg_hash = Helpers.bytes_to_uint256(32, input_padded);
          let r = Helpers.bytes_to_uint256(32, input_padded + 32 * 2);
          let s = Helpers.bytes_to_uint256(32, input_padded + 32 * 3);

  +       let s_high = s.high;
  +       let s_low = s.low;
  +
  +       if (is_not_zero(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF - s_high) ==
FALSE) {
  +           let (output) = alloc();
  +           return (0, output, GAS_COST_EC_RECOVER, 0);
  +       }
  +
  +       let is_s_high_not_big =
is_not_zero(0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFE - s_high);
  +       let is_s_low_big = is_nn(s_low -
0xBAAEDCE6AF48A03BBFD25E8CD0364141);
  +       let is_s_not_valid = (1 - is_s_high_not_big) * is_s_low_big;
  +       if ( is_s_not_valid != FALSE){
  +           let (output) = alloc();
  +           return (0, output, GAS_COST_EC_RECOVER, 0);
  +       }
  +
          // v - 27, see recover_public_key comment
          let (helpers_class) = Kakarot_cairo1_helpers_class_hash.read();
          let (success, recovered_address) =
ICairo1Helpers.library_call_recover_eth_address(
              class_hash=helpers_class, msg_hash=msg_hash, r=r, s=s,
y_parity=v - 27
          );

          if (success == 0) {
              let (output) = alloc();
              return (0, output, GAS_COST_EC_RECOVER, 0);
          }
```

```
        let (output) = alloc();
        memset(output, 0, 12);
        Helpers.split_word(recovered_address, 20, output + 12);

        return (32, output, GAS_COST_EC_RECOVER, 0);
    }
```

https://github.com/kkrt-labs/kakarot/blob/7411a5520e8a00be6f5243a50c160e66ad285563/src/kakarot/precompiles/ec_recover.cairo#L40

Assessed type

Context

ClementWalter (Kakarot) confirmed and commented:

Severity: High

Kakarot mitigated:

This PR fixes validation of ecrecover `r` and `s` values and the library call `mock` to throw error if an invalid values reaches this point.

Status: Partially mitigated. Full details in report from RadiantLabs included in the Mitigation Review section below.

Medium Risk Findings (11)

[M-01] `RIPEMD160` precompile crashes with a Cairo exception for some input lengths

Submitted by muellerberndt, also found by RadiantLabs, gumgumzum, and ahmedaghadi

Calling the `RIPEMD160` precompile with certain input lengths results in a Cairo exception. As a result, some L2 smart contracts that use this precompile cannot be executed.

The bug seems to occur in line 477 of the precompile. The relevant code:

```
    func finish{range_check_ptr, bitwise_ptr: BitwiseBuiltin*}(
        buf: felt*, bufsize: felt, data: felt*, dsize: felt, mswlen: felt
    ) -> (res: felt*, rsize: felt) {
        alloc_locals;
        let (x) = default_dict_new(0);
        tempvar start = x;

        (...)
        let (local arr_x: felt*) = alloc();
```

```
          dict_to_array{dict_ptr=x}(arr_x, 16);
          let (buf, bufsize) = compress(buf, bufsize, arr_x, 16);
          // reset dict to all 0.
          let (x) = default_dict_new(0);

          dict_write{dict_ptr=x}(14, val);
          dict_write{dict_ptr=x}(15, val_15);

          let (local arr_x: felt*) = alloc();
          dict_to_array{dict_ptr=x}(arr_x, 16);
          default_dict_finalize(start, x, 0);
          let (res, rsize) = compress(buf, bufsize, arr_x, 16);
          return (res=res, rsize=rsize);
```

The lower part of the code is reached for some input lengths. Note that `x` is redefined with the line:

```
        let (x) = default_dict_new(0);
```

However, `start` still points to the first element of the original dict `x` initialized at the start of the function. Consequently, squashing the dict with `default_dict_finalize(start, x, 0)` will fail because start and x point to different segments.

Proof of Concept

The issue can be verified by running the following Solidity contract as an `e2e-test`:

```
    contract RIPEMDTest {

          function computeRipemd160(bytes memory data) public view returns
(bytes20) {
              bytes20 result;
              assembly {
                  // Load the free memory pointer
                  let ptr := mload(0x40)

                  // Get the length of the input data
                  let len := mload(data)

                  // Call the RIPEMD-160 precompile (address 0x03)
                  let success := staticcall(
                      gas(),           // Forward all available gas
                      0x03,            // Address of RIPEMD-160 precompile
                      add(data, 32),   // Input data starts after the length
field (32 bytes)
                      len,             // Length of the input data
                      ptr,             // Output will be written to ptr
                      32              // The precompile writes 32 bytes (even
though RIPEMD-160 outputs 20 bytes)
                  )

                  // Check if the call was successful
                  if iszero(success) {
                      revert(0, 0)
                  }

                  // Read the first 20 bytes of the output (RIPEMD-160 outputs
20 bytes)
                  result := mload(ptr)
              }
              return result;
          }

          function hashInputLength20() external view returns (bytes20) {
              bytes memory inputData = new bytes(20);

              for (uint256 i = 0; i < 20; i++) {
                  inputData[i] = 0x61;
              }

              return computeRipemd160(inputData);
          }

          function hashInputLength32() external view returns (bytes20) {
              bytes memory inputData = new bytes(55);
              for (uint256 i = 0; i < 32; i++) {
                  inputData[i] = 0x61;
              }
              return computeRipemd160(inputData);
          }
      }
```

While calling `hashInputLength20()` returns the correct result, `hashInputLength32()` will result in a failed test:

```
    FAILED
tests/end_to_end/Berndt/test_berndt.py::TestBerndt::Test1::test_berndt -
starknet_py.net.client_errors.ClientError: Client failed with code 40.
Message: Contract error. Data: {'revert_error': "Error at
pc=0:1034:\nOperation failed: 714:54 - 711:0, can't subtract two relocatable
values with different segment indexes\nCairo traceback (most recent call
last):\nUnknown location (pc=0:23354)\nUnknown location (pc=0:23354)\nUnknown
location (pc=0:23354)\nUnknown location (pc=0:23354)\nUnknown location
(pc=0:23354)\nUnknown location (pc=0:23354)\nUnknown location
(pc=0:23354)\nUnknown location (pc=0:23354)\nUnknown location
(pc=0:23354)\nUnknown location (pc=0:23354)\nUnknown location
(pc=0:23354)\nUnknown location (pc=0:23354)\nUnknown location
(pc=0:23354)\nUnknown location (pc=0:23352)\nUnknown location
(pc=0:22085)\nUnknown location (pc=0:21859)\nUnknown location
(pc=0:18006)\nUnknown location (pc=0:20890)\nUnknown location
(pc=0:1172)\nUnknown location (pc=0:1158)\n"}
```

In the ordinary EVM, the same function returns the result:

```
    bytes20: 0x0000000000000000000000004a6747d1c9fe21fc
```

Additionally, the following Cairo test reproduces the issue:

```
    import pytest
      from Crypto.Hash import RIPEMD160
      from hypothesis import given, settings
      from hypothesis.strategies import binary


      @pytest.mark.asyncio
      @pytest.mark.slow
      class TestRIPEMD160:
          @given(msg_bytes=binary(min_size=56, max_size=63))
          @settings(max_examples=3)
          async def test_ripemd160_should_return_correct_hash(self, cairo_run,
msg_bytes):
              precompile_hash = cairo_run("test__ripemd160",
msg=list(msg_bytes))

              # Hash with RIPEMD-160 to compare with precompile result
              ripemd160_crypto = RIPEMD160.new()
              ripemd160_crypto.update(msg_bytes)
              expected_hash = ripemd160_crypto.hexdigest()

              assert expected_hash.rjust(64, "0") ==
bytes(precompile_hash).hex()
```

This will crash with the exception:

```
    E           Exception:
/Users/bernhardmueller/Library/Caches/pypoetry/virtualenvs/kakarot-UuVS5Smv-
py3.10/lib/python3.10/site-
packages/starkware/cairo/common/squash_dict.cairo:29:5: Error at pc=0:1428:
      E           Can only subtract two relocatable values of the same segment
(16 != 13).
```

Recommended Mitigation Steps

Set the pointer `start` to the correct value. It is also very important to finalize the previously initialized `dict x` as the prover can cheat otherwise.

Double-check the invocation path of the precompile `exec_precompile()`, and add end-to-end tests for all precompiles to make sure that they work as expected when called from L2.

Assessed type

Error

[ClementWalter (Kakarot) confirmed and commented]:

Severity: Medium
Ok, but assets are not directly at risk considering that the tx will revert.

[LSDan (judge) decreased severity to Medium]

Kakarot mitigated:

[PR] fix: finalize dictionary in `RIPEMD160`.

Status: Mitigation confirmed.

[M-02] Address aliasing is wrongfully applied even to EOAs

Submitted by Bauchibred, also found by minglei-wang-3570, ulas, and RadiantLabs

Kakarot inherits the address aliasing logic from Optimism as hinted in `AddressAliasHelper.sol`:

```
// from https://github.com/ethereum-
optimism/optimism/blob/a080bd23666513269ff241f1b7bc3bce74b6ad15/packages/contr
acts-bedrock/src/vendor/AddressAliasHelper.sol

  pragma solidity ^0.8.0;

  library AddressAliasHelper {
      ..snip
  }
```

Now this is a key property of the Optimism bridge, which is that **all contract addresses** are aliased. This is done in order to avoid a contract on L1 to be able to send messages as the same address on L2, because often these contracts will have different owners.

To go into more details about why & how this is used, please review here.

Address aliasing is an important security feature that impacts the behavior of transactions sent from L1 to L2 by smart contracts. Make sure to read this section carefully if you are working with cross-chain transactions. Note that the CrossChainMessenger contracts will handle address aliasing internally on your behalf.

When transactions are sent from L1 to L2 by an Externally Owned Account, the address of the sender of the transaction on L2 will be set to the address of the sender of the transaction on L1. However, the address of the sender of a transaction on L2 will be different if the transaction was triggered by a smart contract on L1.

Because of the behavior of the CREATE opcode, it is possible to create a contract on both L1 and on L2 that share the same address but have different bytecode. Even though these contracts share the same address, they are fundamentally two different smart contracts and cannot be treated as the same contract. As a result, the sender of a transaction sent from L1 to L2 by a smart contract cannot be the address of the smart contract on L1 or the smart contract on L1 could act as if it were the smart contract on L2 (because the two contracts share the same address).

To prevent this sort of impersonation, the sender of a transaction is slightly modified when a transaction is sent from L1 to L2 by a smart contract. Instead of appearing to be sent from the actual L1 contract address, the L2 transaction appears to be sent from an "aliased" version of the L1 contract address. This aliased address is a constant offset from the actual L1 contract address such that the aliased address will never conflict with any other address on L2 and the original L1 address can easily be recovered from the aliased address.

This change in sender address is only applied to L2 transactions sent by L1 smart contracts. In all other cases, the transaction sender address is set according to the same rules used by Ethereum.

TRANSACTION SOURCE	SENDER ADDRESS
L2 user (Externally Owned Account)	The user's address (same as in Ethereum)
L1 user (Externally Owned Account)	The user's address (same as in Ethereum)
L1 contract (using OptimismPortal.depositTransaction)	L1_contract_address + 0x1111000000000000000000000000000000001111

That's to say we expect this aliasing logic to:

- Only be applied to contracts.

However, the issue is that Kakarot applies this aliasing logic not only on contracts but even EOAs, see here:

```
    function sendMessageToL2(address to, uint248 value, bytes calldata data)
external payable {
          uint256 totalLength = data.length + 4;
          uint256[] memory convertedData = new uint256[](totalLength);
          convertedData[0] =
uint256(uint160(AddressAliasHelper.applyL1ToL2Alias(msg.sender)));
          convertedData[1] = uint256(uint160(to));
          convertedData[2] = uint256(value);
          convertedData[3] = data.length;
          for (uint256 i = 4; i < totalLength; ++i) {
              convertedData[i] = uint256(uint8(data[i - 4]));
          }


          starknetMessaging.sendMessageToL2{value: msg.value}(kakarotAddress,
HANDLE_L1_MESSAGE_SELECTOR, convertedData);
      }
```

Evidently, the classic contract check of `msg.sender != tx.origin` is missing which means even when the caller is an EOA it's still get aliased.

Impact

Broken functionality for aliasing senders considering even EOAs get aliased instead of it just being restricted to smart contracts; i.e., if developers implement access control checks on the sender of the transactions, say a deposit tx for `e.`, they would have to believe that the address of the contract from the L1 is not aliased when it is an EOA. However, in Kakarot's case, it is. Therefore, it would result in unintentional bugs where the access control will be implemented incorrectly and these transactions will always fail.

Recommended Mitigation Steps

Apply an if `msg.sender != tx.origin` check instead and in the case where this ends up being true then alias the address otherwise let it be.

Assessed type

Context

ClementWalter (Kakarot) confirmed and commented:

Severity: Low
There is no security risk of any kind. But this will be fixed.

LSDan (judge) commented:

I think this fits as a medium given that functionality is unexpectedly impaired/blocked.

obatirou (Kakarot) commented:

Following https://docs.code4rena.com/awarding/judging-criteria/severity-categorization

Low risk (e.g. assets are not at risk: state handling, function incorrect as to spec, issues with comments)

the function of the protocol or its availability could be impacted.

This is related to “incorrect as to spec” but not “impacted function” to us.

LSDan (judge) commented:

Noted, but I still think Medium is appropriate here.

Kakarot mitigated:

PR 1623 and lib PR 12 removed messaging.

Status: Mitigation confirmed.

[M-03] No way to cancel `l1 -< l2` messages

Submitted by 1AutumnLeaf777, also found by Bauchibred, wasm_it, RadiantLabs, and oakcobalt

There is no api to allow cancellation of `l1->l2` messages. In the event of an issue in the Kakarot contracts, this will result in the fee being permanently lost since the user has no ability to reclaim the funds.

https://github.com/kkrt-labs/kakarot/blob/7411a5520e8a00be6f5243a50c160e66ad285563/solidity_contracts/src/L1L2Messaging/L1KakarotMessaging.sol#L26-L61

As we can see there are only functions to either send the message from `l1 -> l2` or consume an `l2` message. There is no `cancelL1toL2Message` present.

Recommended Mitigation Steps

Introduce the following API to let users cancel their messages after waiting the time limit so that they can reclaim funds.

https://docs.starknet.io/architecture-and-concepts/network-architecture/messaging-mechanism/#l2-l1_message_cancellation

Assessed type

Context

ClementWalter (Kakarot) confirmed and commented:

Severity: Low
Only fees would be lost as it is not a bridge. No other funds at risk. But cancellation will be implemented.

[LSDan (judge) commented](#):

Fee loss is still a value leak/loss of funds. Medium is appropriate.

[obatirou (Kakarot) commented](#):

As per the doc, see https://docs.code4rena.com/awarding/judging-criteria/severity-categorization#loss-of-fees-as-low

loss of fees is a LOW.

[LSDan (judge) commented](#):

```
Loss of fees should be regarded as an impact similar to any other loss of
capital
```
Context: I helped write this rule as one of the 3 SC judges that agreed on it.

```
Loss of unmatured yield or yield in motion shall be capped to medium severity.
```
The intent is that loss of fees is not higher than medium, even if they are substantial yield rewards.

```
Loss of dust amounts are QA
```
Only loss of dust is specifically stated as low.

```
Loss of real amounts depends on specific conditions and likelihood
considerations.
```
The condition above is very likely, making this a valid medium.

Kakarot mitigated:

[PR 1623](#) and [lib PR 12](#) removed messaging.

Status: Mitigation confirmed.

[M-04] `decode_legacy_tx` allows validation of signatures with `chain_id` that are larger than felt, and overflows

Submitted by [Emmanuel](#), also found by [20centclub](#)

In `decode_legacy_tx`, `data_len` for `chain_id` is not constrained to be `<31`, which allows validation of signatures with `chain_id` that are larger than felt, and overflows.

Proof of Concept

`chain_id` is pulled from `items\[6].data` via the `bytes_to_felt` helper function.

Looking at `bytes_to_felt` function, we will observe that the return value will overflow if `len>31`, because current would have been shifted `2^(8\*32)` times, which overflows the max felt value of `2^252`:

```
        func bytes_to_felt(len: felt, ptr: felt*) -> felt {
              if (len == 0) {
                  return 0;
              }
              tempvar current = 0;

              // len, ptr, ?, ?, current
              // ?, ? are intermediate steps created by the compiler to unfold
the
              // complex expression.
              loop:
              let len = [ap - 5];
              let ptr = cast([ap - 4], felt*);
              let current = [ap - 1];

              tempvar len = len - 1;
              tempvar ptr = ptr + 1;
              tempvar current = current * 256 + [ptr - 1];//@audit-info
overflow if len>31

              static_assert len == [ap - 5];
              static_assert ptr == [ap - 4];
              static_assert current == [ap - 1];
              jmp loop if len != 0;

              return current;
          }
```

Since `items\[6].data_len` is not constrained to be `<=31`, absurd values of `chain_id`(containing `>31 data_len`) can be passed, which would overflow, and pass the signature validation.

Recommended Mitigation Steps

Assert that `items\[6].data_len <= 31`:

```
    assertassert_nn(31 - items[6].data_len);
```

Assessed type

Under/Overflow

ClementWalter (Kakarot) confirmed and commented:

Severity: Medium

Kakarot mitigated:

[This PR](#) decodes legacy chain id overflow. Assert the chain id does not overflow for legacy post `eip 155` tx.

Status: Mitigation confirmed.

[M-05] `ExponentiationImpl::pow()` returns `0` for `0^0`

*Submitted by **RadiantLabs**, also found by **Bauchibred***

The `ExponentiationImpl::pow()` function in `math.cairo` incorrectly returns `0` when computing `0^0`, instead of the mathematically accepted value of 1. This breaks a fundamental mathematical convention that is relied upon in many mathematical contexts, including polynomial evaluation, Taylor series, and combinatorial calculations.

The issue occurs because the function first checks if the base is zero and returns zero if true, without considering the special case where the exponent is also zero. This early return means that `0^0` evaluates to `0` instead of 1:

```
fn pow(self: T, mut exponent: T) -> T {
        let zero = Zero::zero();
        if self.is_zero() {
                return zero;
        }
        ...
```

The mathematical definition of `0^0 = 1` is not arbitrary. It is the natural definition that makes many mathematical formulas and theorems work correctly. For example, this definition is necessary for:

- The binomial theorem to work correctly when `x=0`.
- Power series expansions to be valid at `x=0`.
- Combinatorial formulas involving empty sets.
- Preserving continuity in certain mathematical limits.

This function is not currently being used to compute `0^0` in the code in scope. However, given the critical nature of the function and fundamental incorrectness of its output, the expectation of this issue causing vulnerabilities in [future code](#) is fulfilled.

Impact

- Mathematical operations that rely on the standard convention of `0^0 = 1` will produce incorrect results.
- Future code that reaches this case in core Kakarot contracts, protocols built on top of Kakarot's codebase or borrowing from it will experience material errors when

processing edge cases

Recommended Mitigation Steps

Add a check for the `0^0` case before checking if the base is zero:

```
fn pow(self: T, mut exponent: T) -> T {
      // Handle 0^0 case first
      if self.is_zero() && exponent.is_zero() {
          return One::one();
      }

      // Rest of the existing function...
      if self.is_zero() {
          return Zero::zero();
      }
      // ...
  }
```

This change preserves the mathematically correct behavior while maintaining all other functionality of the power function.

ClementWalter (Kakarot) confirmed and commented:

Severity: Medium

Kakarot mitigated:

PR 1579 and ssj PR 1022 fixes pow in SSJ.

Status: Mitigation confirmed.

[M-06] Reentrancy check in `account_contract` can be easily circumvented

Submitted by RadiantLabs, also found by muellerberndt and zhaojie

The `account_contract` has a reentrancy check in `execute_starknet_call` to prevent calls made from the Kakarot EVM to Starknet to re-enter the Kakarot EVM.

The check is implemented by checking that `execute_starknet_call` calls Kakarot's address only with the `get_starknet_address` getter, which is indeed harmless:

```
File: account_contract.cairo
  332: @external
  333: func execute_starknet_call{syscall_ptr: felt*, pedersen_ptr:
HashBuiltin*, range_check_ptr}(
  334:     to: felt, function_selector: felt, calldata_len: felt, calldata:
felt*
  335: ) -> (retdata_len: felt, retdata: felt*, success: felt) {
    Ownable.assert_only_owner();
    let (kakarot_address) = Ownable.owner();
    let is_get_starknet_address = Helpers.is_zero(
        GET_STARKNET_ADDRESS_SELECTOR - function_selector
    );
    let is_kakarot = Helpers.is_zero(kakarot_address - to);
    tempvar is_forbidden = is_kakarot * (1 - is_get_starknet_address);
    if (is_forbidden != FALSE) {
        let (error_len, error) = Errors.kakarotReentrancy();
        return (error_len, error, FALSE);
    }
    let (retdata_len, retdata) = call_contract(to, function_selector, calldata_len, calldata);
    return (retdata_len, retdata, TRUE);
}

```

However, this check leaves another possibility open, that is that the `account_contract` could call itself or another `account_contract` with a signed transaction to re-enter the Kakarot EVM, which is extremely vulnerable to reentrancy because of its extensive use of cached data.

While an exploit via this path can have critical impact on the integrity of the EVM, its likelihood is extremely low because `execute_starknet_call` is accessible only via whitelisted contracts.

Proof of Concept

Because Kakarot stores the effects of the execution at the very end of it via the `Starknet.commit`, it openly does not follow the check-effect-interaction pattern, so the integrity of the EVM is at risk of reentrancy via submissions of signed transactions within the `call cairo` precompile.

Recommended Mitigation Steps

Consider removing the reentrancy check in `account_contract`, and adding a reentrancy guard on the `Kakarot.eth_call` function.

[ClementWalter (Kakarot) confirmed and commented](#):

Severity: Medium
Re-entrancy is actually possible should the attacker be in possession of the signed tx before this tx is actually executed, and can front run it to store it is starknet before it's processed. It's possible, though seems very unlikely. We'll fix the re-entrancy check to put in in the Kakarot contract level.

[obatirou (Kakarot) commented](#):

On second thoughts because of the whitelisting it is impossible to exploit. The only way to access the `execute_starknet_call` is through the `cairo_precompile` but this precompile is whitelisted. At the specified commit for C4, only the DualVMToken contract is whitelisted which cannot lead to the exploit.

We will still do the mitigation with a re-entrancy check at Kakarot contract level.

LSDan (judge) commented:

I'm not willing to jump to "whitelisting makes this impossible to exploit" and still think medium applies. Reentrancy is no joke and a hand wavy hypothetical is enough in this case.

ahmedaghadi (warden) commented:

@LSDan - As attack is only possible through "whitelisting", doesn't it comes under Centralization Risks?

LSDan (judge) commented:

Not in this case. The potential for damage is enough for me to keep it in place as a medium.

Kakarot mitigated:

In PR 1582 the reentrancy check is moved at Kakarot level in `eth_call`.

Status: Mitigation confirmed.

[M-07] Account contract does not gracefully handle panics in called contracts

Submitted by RadiantLabs, also found by oakcobalt

The `DualVmToken` is a utility EVM contract allowing Kakarot EVM accounts to operate Starknet ERC-20s via classic `balanceOf()`, `transfer()` etc. operations.

Calls to the `DualVmToken` contract are routed via `CairoLib` -> `kakarot_precompiles.cairo` -> `account_contract.cairo`, which finally makes a call to the `call_contract()` Cairo 0 syscall to invoke the appropriate Starknet ERC-20 contract.

If we look at how the `execute_starknet_call()` function in `account_countract.cairo` is implemented:

```
File: account_contract.cairo
  332: @external
  333: func execute_starknet_call{syscall_ptr: felt*, pedersen_ptr:
HashBuiltin*, range_check_ptr}(
  334:     to: felt, function_selector: felt, calldata_len: felt, calldata:
felt*
  335: ) -> (retdata_len: felt, retdata: felt*, success: felt) {
  336:     Ownable.assert_only_owner();
```

```
 337:     let (kakarot_address) = Ownable.owner();
 338:     let is_get_starknet_address = Helpers.is_zero(
 339:         GET_STARKNET_ADDRESS_SELECTOR - function_selector
 340:     );
 341:     let is_kakarot = Helpers.is_zero(kakarot_address - to);
 342:     tempvar is_forbidden = is_kakarot * (1 - is_get_starknet_address);
 343:     if (is_forbidden != FALSE) {
 344:         let (error_len, error) = Errors.kakarotReentrancy();
 345:         return (error_len, error, FALSE);
 346:     }
 347:     let (retdata_len, retdata) = call_contract(to, function_selector,
calldata_len, calldata);
 348:     return (retdata_len, retdata, TRUE);
```

We see that the function admits graceful failures (by returning a `success: felt` value that can be `FALSE`), which are then properly handled in the calling precompile. However, the actual call to the target contract (at L347) does not return a `success` boolean, and instead uses the `call_contract` syscall which panics on failures.

This means that any call that panics in the called contract can cause a revert at RPC level.

To illustrate how this is not a hypothetical scenario, but a very likely one that can be achieved also with the DualVmToken contract that is in scope and consequently planned to be whitelisted to make this call, we make a simple example:

an EVM contract uses DualVmToken to transfer more tokens than what it has in its balance

In this scenario, standard OpenZeppelin ERC-20 implementations panic, bubbling up the error up to the Kakarot RPC level.

This also means that most of the ERC-20 behaviours Kakarot intends to support will lead to reverts at the RPC level if encountered. The following table shows the effect of each behaviour if encountered in a call to a `DualVmToken`:

FEATURE	STATUS
Missing return values	Supported
Upgradeability	-
Pausability	RPC-level reverts
Revert on approval to zero address	RPC-level reverts
Revert on zero value approvals	RPC-level reverts
Revert on zero value transfers	RPC-level reverts

FEATURE	STATUS
Revert on transfer to the zero address	RPC-level reverts
Revert on large approvals and/or transfers	RPC-level reverts
Doesn't revert on failure	Unsupported, see separate finding
Blocklists	RPC-level reverts

Impact

Any contract can use the above "over-transfer" call to cause RPC-level crashes in the Kakarot EVM, regardless of how defensively they were called. Protective measures generally considered safe in the EVM space like `ExcessivelySafeCall` (which is used in LayerZero cross-chain applications to prevent channel DoSes that can permanently freeze in-flight tokens), can be bypassed by causing a RPC-level cairo revert.

More generally, any transaction containing a call to a reverting `DualVmToken` at any depth will revert. This means contracts whose logic requires such a call to succeed will be temporarily (if the call eventually succeeds) or permanently bricked.

Proof of Concept

The RPC-level panic can be proved by simply editing the `test_should_transfer` in `test_dual_vm_token.py` end-to-end test to transfer `balance_owner_before + amount` instead of `amount` at L110.

Recommended Mitigation Steps

The new `AccountContract` Cairo 1 implementation uses the Cairo 1 `call_contract` syscall, which offers a recoverable error interface through a `nopanic` signature returning a Result. If it is not possible to deploy the new version, the Cairo 1 syscall can be used from the Cairo 0 `account_contract` via a library call as is done with other functionality.

ClementWalter (Kakarot) confirmed and commented:

Severity: Informative
This is a known limitation of starknet. It is not a problem as relayer can decide to not relay the tx.

LSDan (judge) decreased severity to Medium and commented:

I find it hard to look at this as anything other than an impact to expected functionality/availability of functionality, making the severity medium. As shown above, this is likely to be encountered often with very standard functionality, much of which is included in

OZ EVM implementations of common token patterns. The potential for novel uses of this dynamic to cause hypothetical exploits or denial of service attacks doesn't seem like something that should be ignored or written off to a "limitation of starknet".
obatirou (Kakarot) commented:

To summarize:

1. There is no way to handle a failed syscall in starknet currently. The whole tx panics and stops.
2. DualVM token are starknet token under the hood.
3. As such, patterns such as the `excessively safe` call mentioned by @EV_om in #48 are actually not working as expected.
4. A possible mitigation for the `DualVMToken` especially is to reimplement all the validation logic in solidity and perform the cairo calls only if it will surely succeed.

Let's keep it Medium.

Kakarot mitigated:

This PR uses a call to the Cairo1Helpers class to access the "new" `call_contract` syscall, that will in the future have the ability to gracefully handle failed contract calls.

Status: Mitigation confirmed.

[M-08] `DualVmToken` can be abused to cause RPC-level reverts by revoking native token approval to Kakarot

Submitted by RadiantLabs, also found by minglei-wang-3570

The `DualVmToken` is a utility EVM contract allowing Kakarot EVM accounts to operate Starknet ERC-20s via classic `balanceOf()`, `transfer()`, etc., operations.

These classic operations are offered in two flavors, one that accepts Starknet addresses (`uint256`) and one that accepts EVM addresses (`address`). In case an EVM address is provided, the corresponding Starknet address is looked up through Kakarot first.

If we look at the `approve(uint256, ...)` function in `DualVmToken`:

```
File: DualVmToken.sol
    /// @dev Approve a starknet address for a specific amount
    /// @param spender The starknet address to approve
    /// @param amount The amount of tokens to approve
    /// @return True if the approval was successful
    function approve(uint256 spender, uint256 amount) external returns (bool) {
        _approve(spender, amount);
        emit Approval(msg.sender, spender, amount);
        return true;
214:     }
215:
216:     function _approve(uint256 spender, uint256 amount) private {
217:         if (spender >= STARKNET_FIELD_PRIME) {
218:             revert InvalidStarknetAddress();
219:         }
220:         // Split amount in [low, high]
221:         uint128 amountLow = uint128(amount);
222:         uint128 amountHigh = uint128(amount >> 128);
223:         uint256[] memory approveCallData = new uint256[](3);
224:         approveCallData[0] = spender;
225:         approveCallData[1] = uint256(amountLow);
226:         approveCallData[2] = uint256(amountHigh);
227:
228:         starknetToken.delegatecallCairo("approve", approveCallData);
229:     }
```

We can see that no check whatsoever is done on the `spender` input, except for felt overflow at L217.

This means that the provided `address` could be any Starknet account, including a contract that is not an EVM account, and most importantly including the Kakarot contract.

This is particularly relevant because native token transfers in the Kakarot EVM work under the assumption that Starknet account contracts have infinite approval granted to the Kakarot contract, as we can see from the `account_contract` initialization:

```
File: library.cairo
  083:     func initialize{
  084:         syscall_ptr: felt*,
  085:         pedersen_ptr: HashBuiltin*,
  086:         range_check_ptr,
  087:         bitwise_ptr: BitwiseBuiltin*,
  088:     }(evm_address: felt) {
  ---
  101:         IERC20.approve(native_token_address, kakarot_address,
infinite);
```

By removing this approval and attempting to move native tokens, an EVM contract account can jeopardize EVM state finalization (that is where native token transfers are settled on the Starknet ERC20) and cause RPC-level crashes in the Kakarot EVM, regardless of how defensively they were called. Protective measures generally considered safe in the EVM space like `ExcessivelySafeCall` (which is used in LayerZero cross-chain applications to prevent channel DoSes that can permanently freeze in-flight tokens), can be bypassed by causing a RPC-level Cairo revert.

Proof of Concept

The following contract can trigger an RPC-level revert whenever called with any `calldata`, assuming that `target` is initialized with the `DualVmToken` associated to Kakarot's native ERC20.

```
contract DualVmTokenHack {
      DualVmToken target;

      constructor(DualVmToken _target) {
          target = _target;
      }

      fallback {
          uint256 kakarot = DualVmToken.kakarot();
          target.approve(kakarot, 0);
          address(0xdeadbeef).transfer(1 wei);
      }
  }
```

Recommended Mitigation Steps

Consider adding the following check to the `DualVmToken.approve` function:

```
    require(spender != kakarot);
```

[ClementWalter (Kakarot) confirmed and commented]:

Severity: Low
This is a low issue as there is no clear external attack path. A user would need to decide to disabling its own account by making an approval to kakarot with `0`.

[LSDan (judge) decreased severity to Medium and commented]:

Availability and expected functionality are impacted but there are no funds at risk as far as I can see. This can be used for hypothetical griefing attacks.

[EV_om (warden) commented]:

@LSDan - just to explain our High severity assessment for this finding, as well as [#40], [#49], [#52] and [#53], which share the root cause of RPC-level reverts that is very specific to this codebase.

The scenario we had in mind (which we could admittedly have made a better job of explaining) was that in which a contract requires a call to be made to reach a subsequent state. For example: a bridge contract that requires messages to be processed sequentially. It is ok for the call not to succeed, but it must be attempted.

In this situation, the call reverting at the RPC level leads to the transaction never succeeding, and hence, all funds in the contract being locked irreversibly.

This pattern is quite widely used, such as in LayerZero's `_blockingLzReceive`, and is not safe on Kakarot, with the highest impact being the permanent freezing of funds.
LSDan (judge) commented:

Ruling stands. You're relying on handwavy hypotheticals and external factors. Medium is correct.

Kakarot mitigated:

This PR checks kakarot approval `dualVMToken`.

Status: Mitigation confirmed.

[M-09] Jump in creation code leads to reverting of the starknet transaction

Submitted by ahmedaghadi, also found by gumgumzum

If the creation code consists of `JUMP` or `JUMPI` opcode for `offset` less than `creation code length`, then the starknet transaction reverts.

The `evm.cairo::jump` function is called whenever the `JUMP` or `JUMPI` opcode is executed. This function is as follows:

```
// @notice Update the program counter.
  // @dev The program counter is updated to a given value. This is only ever
called by JUMP or JUMPI.
  // @param self The pointer to the execution context.
  // @param new_pc_offset The value to update the program counter by.
  // @return model.EVM* The pointer to the updated execution context.
  func jump{syscall_ptr: felt*, pedersen_ptr: HashBuiltin*, range_check_ptr,
state: model.State*}(
      self: model.EVM*, new_pc_offset: felt
  ) -> model.EVM* {
  ->  let out_of_range = is_nn(new_pc_offset - self.message.bytecode_len);
      if (out_of_range != FALSE) {
          let (revert_reason_len, revert_reason) =
Errors.invalidJumpDestError();
          let evm = EVM.stop(self, revert_reason_len, revert_reason,
Errors.EXCEPTIONAL_HALT);
          return evm;
      }

      let valid_jumpdests = self.message.valid_jumpdests;
      with valid_jumpdests {
```

```
->      let is_valid_jumpdest = Internals.is_valid_jumpdest(
            self.message.code_address, new_pc_offset
        );
    }

    // ...
}
```

It can be seen that, it checks for `out_of_range` which is `new_pc_offset - self.message.bytecode_len`. For the creation code, technically `bytecode_length` should be `0` but here, it will consist the length of the creation code. So, if the `new_pc_offset` is less than the `creation code length`, then this check would pass and then it checks for `is_valid_jumpdest` by calling [`evm.cairo::is_valid_jumpdest`](), which is as follows:

```
func is_valid_jumpdest{
      syscall_ptr: felt*,
      pedersen_ptr: HashBuiltin*,
      range_check_ptr,
      valid_jumpdests: DictAccess*,
      state: model.State*,
  }(code_address: model.Address*, index: felt) -> felt {
      alloc_locals;
      let (is_cached) = dict_read{dict_ptr=valid_jumpdests}(index);
      if (is_cached != 0) {
          return TRUE;
      }

      // If the account was created in the same transaction,
      // a cache miss is an invalid jumpdest as all valid jumpdests were
cached on deployment.
      let code_account = State.get_account(code_address.evm);
      if (code_account.created != 0) {
          return FALSE;
      }

  ->  let (is_valid) = IAccount.is_valid_jumpdest(code_address.starknet,
index);
      dict_write{dict_ptr=valid_jumpdests}(index, is_valid);

      return is_valid;
  }
```

Here, `let (is_valid) = IAccount.is_valid_jumpdest(code_address.starknet, index);` would revert the whole starknet transaction as it will call zero address which isn't `IAccount` contract. So any contract using `CREATE` or `CREATE2` opcode will be vulnerable to this issue or if a contract makes a callback to another contract by making sure to only send limited gas and handle the revert case properly, the other contract can use `CREATE` or `CREATE2` opcode to make the transaction revert forcefully.

Proof of Concept

Consider the following code:

```
#define macro MAIN() = takes(0) returns(0) {
      0x00
      jump
  }
```

Save the above code in a file name `creation_code.huff`. You can refer [Installing Huff](), if you don't have huff installed.

The above code will jump to the offset `0` which is less than the creation code length (i.e. `2`) and can be converted to bytecode using the following command:

```
huffc creation_code.huff --bytecode
```

The output will be:

```
    60028060093d393df35f56
```

The runtime code, after removing the first few bytes which are responsible for deployment:

```
    5f56
```

Also, consider the following code:

```
#define macro MAIN() = takes(0) returns(0) {
      0x5f 0x00 mstore8 // store `PUSH0` opcode at memory offset 0
      0x56 0x01 mstore8 // store `JUMP` opcode at memory offset 1

      0x02 // size
      0x00 // offset
      0x00 // value
      create
  }
```

Save the above code in a file name `create.huff`. This will jump to the offset `0` which is less than the creation code length (i.e. `2`) and can be converted to bytecode using the following command:

```
huffc create.huff --bytecode
```

The output will be:

```
    600e8060093d393df3605f5f53605660015360025f5ff0
```

The runtime code, after removing the first few bytes which are responsible for deployment:

```
    605f5f53605660015360025f5ff0
```

The `create.huff` would call `CREATE` opcode with creation code as `0x5f56` (`creation_code.huff`).

Now to to run the test, add this file named `test_kakarot_jump.py` in [kakarot/tests/end*to*end]() folder:

```
import pytest
  import pytest_asyncio
  from starknet_py.contract import Contract

  from kakarot_scripts.utils.starknet import (
      get_contract,
  )
  from tests.utils.helpers import (
      generate_random_evm_address,
      hex_string_to_bytes_array,
  )


  @pytest.fixture(scope="session")
  def evm(deployer):
      """
      Return a cached EVM contract.
      """
      return get_contract("EVM", provider=deployer)


  @pytest_asyncio.fixture(scope="session")
  async def origin(evm, max_fee):
      """
      Return a random EVM account to be used as origin.
      """
      evm_address = int(generate_random_evm_address(), 16)
      await evm.functions["deploy_account"].invoke_v1(evm_address,
max_fee=max_fee)
      return evm_address


  @pytest.mark.asyncio(scope="session")
  class TestKakarot:
      async def test_execute_jump(
          self, evm: Contract, origin
      ):
          params = {
                  "value": 0,
                  "code": "605f5f53605660015360025f5ff0", // create.huff
                  "calldata": "",
                  "stack":
"0000000000000000000000000000000000000000000000000000000000000000",
                  "memory": "",
                  "return_data": "",
                  "success": 1,
          }
          result = await evm.functions["evm_call"].call(
              origin=origin,
              value=int(params["value"]),
              bytecode=hex_string_to_bytes_array(params["code"]),
              calldata=hex_string_to_bytes_array(params["calldata"]),
              access_list=[],
          )
          print("result", result)
          assert result.success == params["success"]
```

You can run the test by:

```
# In one terminal
  cd kakarot
  make run-nodes

  # In another terminal
  cd kakarot
  uv run deploy
  uv run pytest -s
"./tests/end_to_end/test_kakarot_jump.py::TestKakarot::test_execute_jump"
```

This test will throw error as follows:

```
    starknet_py.net.client_errors.ClientError: Client failed with code 40.
Message: Contract error. Data: {'revert_error': 'Error at pc=0:37:\nGot an
exception while executing a hint: Requested contract address
0x0000000000000000000000000000000000000000000000000000000000000000 is not
deployed.\nCairo traceback (most recent call last):\nUnknown location
(pc=0:23874)\nUnknown location (pc=0:23829)\nUnknown location
(pc=0:23287)\nUnknown location (pc=0:22784)\nUnknown location
(pc=0:22784)\nUnknown location (pc=0:22784)\nUnknown location
(pc=0:22784)\nUnknown location (pc=0:22784)\nUnknown location
(pc=0:22784)\nUnknown location (pc=0:22784)\nUnknown location
(pc=0:22784)\nUnknown location (pc=0:22784)\nUnknown location
(pc=0:22784)\nUnknown location (pc=0:22784)\nUnknown location
(pc=0:22782)\nUnknown location (pc=0:21980)\nUnknown location
(pc=0:11433)\nUnknown location (pc=0:8999)\nUnknown location
(pc=0:9120)\nUnknown location (pc=0:5232)\n'}
```

You can see the error `Requested contract address 0x0000000000000000000000000000000000000000000000000000000000000000 is not deployed`.

Tools Used

Python test suite

Recommended Mitigation Steps

For `JUMP` and `JUMPI` opcodes, there should be a check whether the current environment is running `creation` code or `runtime` code. If it's `creation` code, then it should revert with invalid jump destination error.

Assessed type

Error

[ClementWalter (Kakarot) confirmed and commented]():

Severity: Medium
Ok, but assets are not directly at risks considering the tx will revert.x

[LSDan (judge) decreased severity to Medium]():

Kakarot mitigated:

[This PR]() sets `code_address` of the deployment code to the created address.

Status: Mitigation confirmed.

[[M-10] Incorrect `totalsupply` value will be returned due to erroneous return data decode implementation]()

Submitted by [oakcobalt](), also found by [RadiantLabs]() and [gumgumzum]()

The returned value of `totalSupply()` in a starknet ERC20 contract is expected to fit in uint256, which is expressed in `(uint128, uint128)` with the first uint128 representing the lower 128bits.

The issue is current implementation of `DualVMToken::totalSupply` incorrectly decodes `returnData` as uint256 instead of `(uint128, uint128)`. Because the first uint128 is the lower

128bits of a uint256 number. This means, `totalSupply` will return an incorrect value because it only reads the lower 128bits.

```
//kakarot/solidity_contracts/lib/kakarot-lib/src/CairoLib.sol
      function totalSupply() external view returns (uint256) {
          bytes memory returnData =
starknetToken.staticcallCairo("total_supply");
  |>      return abi.decode(returnData, (uint256));
      }
```

https://github.com/kkrt-labs/kakarot/blob/7411a5520e8a00be6f5243a50c160e66ad285563/solidity_contracts/src/CairoPrecompiles/DualVmToken.sol#L82

Impacts

`totalSupply` will return incorrect value. Due to `totalSupply` is critical in many defi or accounting based logic, this potentially leads to fund loss and errorneas accounting in any user application that uses DualVMToken.sol. Depending on the context of the application that calls DualVMToken.sol, the fund loss could be critical.

Recommended Mitigation Steps

In `totalSupply()`, change the decode following `_balanceOf`'s implementation:

```
...
          (uint128 valueLow, uint128 valueHigh) = abi.decode(
              returnData,
              (uint128, uint128)
          );
          return uint256(valueLow) + (uint256(valueHigh) << 128);
```

Assessed type

Error

ClementWalter (Kakarot) confirmed and commented:

Severity: Low

LSDan (judge) commented:

Medium is appropriate here. Receiving an incorrect value for `TotalSupply` can cause untold amounts of hand wavy hypothetical misery.

ClementWalter (Kakarot) commented:

Ok for Medium.

2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

There might be paths where `totalSupply` can be used to trigger actions and eventually drain funds.
Kakarot mitigated:

This PR fixes an issue where `totalSupply` was skipping the high-part of the u256.

- Correctly deserialize return value.
- Update test to launch a token with populated (low, high) parts of the u256 struct.

Status: Mitigation confirmed.

[M-11] `handle_l1_message` may unfairly revert `l2` tx with sufficient `l1` sender balance, due to vulnerable fee charge implementation

Submitted by oakcobalt, also found by muellerberndt and RadiantLabs

`handle_l1_message` can only be invoked by starknet os(`@l1_handler`) and is not a regular user invoked transaction from `eth_send_raw_unsigned_tx` flow.

`kakarot::handle_l1_message` -> [library:[handle_l1_message](https://github.com/kkrt-labs/kakarot/blob/7411a5520e8a00be6f5243a50c160e66ad285563/src/kakarot/library.cairo#L421) -> `Interpreter::execute`)

`handle_l1_message` hardcodes EVM gaslimit(2100000000) and gasprice(1) for every `L1->L2` message regardless of the complexity of the actual `l2` tx. In `interpreter::execute`, `L1sender`'s cached balance will be subtracted with the max_fee (2100000000 x 1) first before performing ETH transfer or running EVM.

Case: L1 sender performs minimal operations on L2:

For an L1 sender who only transfers some ETH to a L2 address or perform simple opcodes, the actual gas cost (`required_gas`) may be very close to the intrinsic gas cost(21000). This means the `actual_fee` `l1sender` is required to pay is around `21000 x 1`, which is far less than the calculated `max_fee 2100000000 x 1`.

In this case, `interpreter::execute` will first subtract `max_fee`(`2100000000`) from `L1sender`'s cached balance (`Account.set_balance(sender, &new_balance)`). Note that this cached balance subtraction is done before ETH value transfer and `run(evm)`, which means any subsequent logic will be using `L1sender`'s `new_balance` (e.g., `X - 2100000000`).

```
//src/kakarot/interpreter.cairo
      func execute{
  ...
      }(
          env: model.Environment*,
          address: model.Address*,
          is_deploy_tx: felt,
          bytecode_len: felt,
          bytecode: felt*,
          calldata_len: felt,
          calldata: felt*,
          value: Uint256*,
          gas_limit: felt,
          access_list_len: felt,
          access_list: felt*,
      ) -> (model.EVM*, model.Stack*, model.Memory*, model.State*, felt, felt)
{
  ...
  |>      let max_fee = gas_limit * env.gas_price;
          let (fee_high, fee_low) = split_felt(max_fee);
          let max_fee_u256 = Uint256(low=fee_low, high=fee_high);

          with state {
              let sender = State.get_account(env.origin);
              //@audit L1->L2 flow: L1 sender's cached is first subtracted
with max_fee based on hardcoded values, regardless of the actual fee required
based on L2tx's complexity
  |>          let (local new_balance) = uint256_sub([sender.balance],
max_fee_u256);
              let sender = Account.set_balance(sender, &new_balance);
  ...
              let transfer = model.Transfer(sender.address, address, [value]);
              let success = State.add_transfer(transfer);
  ...
          if (success == 0) {
              let (revert_reason_len, revert_reason) = Errors.balanceError();
              tempvar evm = EVM.stop(evm, revert_reason_len, revert_reason,
Errors.EXCEPTIONAL_HALT);
          } else {
              tempvar evm = evm;
          }

          with stack, memory, state {
              let evm = run(evm);
          }

          let required_gas = gas_limit - evm.gas_left;
  ...
          let actual_fee = total_gas_used * env.gas_price;
          let (fee_high, fee_low) = split_felt(actual_fee);
          let actual_fee_u256 = Uint256(low=fee_low, high=fee_high);
          let transfer = model.Transfer(sender.address, coinbase.address,
actual_fee_u256);

          with state {
              State.add_transfer(transfer);
              State.finalize();
          }

          return (evm, stack, memory, state, total_gas_used, required_gas);
```

https://github.com/kkrt-labs/kakarot/blob/7411a5520e8a00be6f5243a50c160e66ad285563/src/kakarot/interpreter.cairo#L950-L951

POC

Suppose `L1sender`'s current balance is (`2100000000 + 21000`). And `L1sender` wants to transfer `2100000000` to an `L2` address. Suppose the `required_gas` is only `21000` so that `L1sender`'s current balance is sufficient.

1. In `interpreter::execute`, `max_fee`(`2100000000`) is first subtracted from `L1sender`'s cached balance. `new_balance = 2100`.
2. ETH `value`(`2100000000`) transfer logic is performed based on the cached new balance. (`let success = State.add_transfer(transfer)`) Because in the cached state `2100 - 2100000000 < 0`, `success -> 0`.
3. `success == 0` logic is executed, EVM tx reverts with an `EXCEPTIONAL_HALT`.
4. Due to EVM reverted with `exceptional_halt`, total gas is consumed, `max fee 2100000000` is charged.

The above shows that despite `L1sender` having enough balance to perform the `L2` tx, `L1sender`'s tx is reverted with `max fee 2100000000` charged. `L1sender` only has `21000` left after the tx and no transfer is done.

Impact

`L1sender`'s `L2` tx can be unfairly reverted when they have enough balance to cover transfer value and the actual fee.

Because `handle_l1_message` is not invoked by RPC endpoint users, `L1sender` doesn't have a chance to specify `max_fee` value. In such cases, `L1sender` might only reasonably expect to have the EVM gas cost sufficient based on the complexity of the `L2` tx, which can be much lower than `max_fee`. Although `max_fee` is marginal in ETH value, `L1sender`'s `L2` tx can still fail due to the sequence of fee deduction in the cached state. The failure is not the `L1sender`'s fault.

Recommended Mitigation Steps

Because `handle_l1_message` is different from `eth_send_raw_unsigned_tx` in its invocation and `max_fee` setting, consider allowing `L1sender` to input `gas_limit` from `L1`, and decode the user input `gas_limit` in `handle_l1_message` to compute `max_fee`. Or, refactor the control flow to skip max fee deduction when called from `handle_l1_message`.

[ClementWalter (Kakarot) confirmed and commented]:

Severity: Medium
Ok, the gas price for `handle_l1_message` should be `0`.

Kakarot mitigated:

[PR 1584] fix: `handle_l1_message` gas price is set to `0`; superseded by [PR 1623] fix: remove messaging and [PR 12] fix: remove messaging

Status: Mitigation error. Full details in reports from RadiantLabs included in the [Mitigation Review] section below.

Low Risk and Non-Critical Issues

For this audit, 13 reports were submitted by wardens detailing low risk and non-critical issues. The [report highlighted below] by **Radiant Labs** was marked as best from the judge and is included for completeness.

The following wardens also submitted reports: [Rhaydden], [20centclub], [Bauchibred], [NexusAudits], [Koolex], [kirill_taran], [1AutumnLeaf777], [Emmanuel], [gumgumzum], [oakcobalt], [fyamf], [ahmedaghadi].

Please see the [Mitigation Review] section below for a complete list of all Low Risk and Non-Critical issues identified during the audit, and confirmed as mitigated.

[01] Missing boundary check in EVM transaction decoding

With the `account_contract.cairo`'s `execute_from_outside` function, callers submit a `calldata` blob of `calldata_len` length, out of which the transaction data is extracted by selecting `[call_array].data_len` bytes starting from position `[call_array].data_offset`.

Within the logic that makes this extraction, there is no check that `[call_array].data_offset + [call_array].data_len` fits within `calldata_len`, effectively allowing for the submission of transactions that pass or fail validation depending on what's allocated out of the boundaries of the input data.

Consider adding a boundary check to enforce `[call_array].data_offset + [call_array].data_len < calldata_len`.

Similar lack of validation for non-critical variables can be found in the actual length of the `signature` array vs `signature_len`, the actual length of the `call_array` vs `call_array_len`, as well as in `eth_transactions::parse_access_list()`, where the parsing of `address_item` misses a check that `address_item.data_len = 20`, that the `access_list.data_len = 2` (spec in the python snippet here).

Kakarot mitigated:

PR 1633: Check `call_array` within bounds `calldata`, `stack_size_diff`, remove setter native token and raise for invalid `y_parity` values

Status: Mitigation confirmed.

[02] Several of the `stack_size_diff` values defined in `constants.cairo` are off

The `stack_size_diff` constants define, for each opcode, what the expected delta between the stack size before and after the operation is. These values are used by the interpreter to predict EVM stack overflows.

The following values are incorrect:

- `ADDMOD` is `-1`, should be `-2`
- `MULMOD` is `-1`, should be `-2`
- `NOT` is `-1`, should be `0`
- `CALLDATACOPY` is `0`, should be `-3`
- `CODECOPY` is `0`, should be `-3`
- `RETURNDATACOPY` is `0`, should be `-3`

Kakarot mitigated:

PR 1633: Check `call_array` within bounds `calldata`, `stack_size_diff`, remove setter native token and raise for invalid `y_parity` values

Status: Mitigation confirmed.

[03] Inconsistent interfaces used for ERC-20 casing

On Starknet, there are two main ways of defining ERC-20 functions, `camelCase` and `snake_case`. At the time of writing, `camelCase` is being deprecated for `snake_case`, with many tokens supporting both for maximum compatibility.

There is, however, an inconsistency between Kakarot's native token handling, which uses `camelCase` `balanceOf` and `transferFrom` calls, and the `DualVmToken` contract that instead uses `snake_case` only.

For maximum compatibility, it is recommended to update both Kakarot and `DualVmToken` to support both and accept additional configuration selecting which interface to use.

Kakarot mitigated:

PR 1633: Check `call_array` within bounds `calldata`, `stack_size_diff`, remove setter native token and raise for invalid `y_parity` values

Status: Mitigation confirmed.

[04] After Kakarot's native token is changed, an approval from `account_contract` instances can't be re-triggered

When an `account_contract` instance is created and initialized, it approves the Kakarot's native token to Kakarot, in order to allow it to settle native transfers happening within the EVM:

```
File: library.cairo
    func initialize{
        syscall_ptr: felt*,
        pedersen_ptr: HashBuiltin*,
        range_check_ptr,
        bitwise_ptr: BitwiseBuiltin*,
    }(evm_address: felt) {
  ---
        let infinite = Uint256(Constants.UINT128_MAX,
Constants.UINT128_MAX);
        IERC20.approve(native_token_address, kakarot_address,
infinite);
```

However, Kakarot has a setter that can be called by admins to update the native token used within the EVM:

```
File: kakarot.cairo
  @external
  func set_native_token{syscall_ptr: felt*, pedersen_ptr: HashBuiltin*,
range_check_ptr}(
      native_token_address: felt
  ) {
      Ownable.assert_only_owner();
      return Kakarot.set_native_token(native_token_address);
  }
```

After this is called, there is no way to re-trigger approval from `account_contract` that were previously initialized.

As this functionality is likely not intended to ever be used after Kakarot has been deployed, consider implementing an initializer pattern.

Alternatively, consider adding the necessary methods in Kakarot and `account_contract` to allow the Kakarot contract to force-approve spending of the native token from the account contracts to itself.

Kakarot mitigated:

PR 1633: Check `call_array` within bounds `calldata`, `stack_size_diff`, remove setter native token and raise for invalid `y_parity` values

Status: Mitigation confirmed.

[05] Some invalid values for the `v` field in Ethereum signatures are accepted

Within the `account_contract`'s `execute_from_outside` function, the validation on `v` is unnecessarily loose: whenever the correct `y_parity` value is `1`, values like `2` or `3` will still be accepted. This is because the `verify_eth_signature_uint256` function calls the `recover_eth_address` Cairo 1 library function to verify the signature, which parses `y_parity` as a boolean.

ClementWalter (Kakarot) confirmed

Kakarot mitigated:

This PR checks:

- Missing boundary check in EVM transaction decoding.
  - Add a check to ensure `[call_array].data_offset + [call_array].data_len <= calldata_len`.
- Several of the `stack_size_diff` values defined in `constants.cairo` are off.
  - Put the right values.
- After Kakarot's native token is changed, an approval from `account_contract` instances can't be re-triggered.
  - Remove the setter, this will not change.
- Some invalid values for the `v` field in Ethereum signatures are accepted.
  - Add a check to ensure `y_parity` is `0` or `1`.

Status: Mitigation confirmed.

Mitigation Review

Introduction

Following the C4 audit, Zenith team RadiantLabs (3docSec and EV_om) reviewed the mitigations for all identified issues within a private repository.

Mitigation Review Scope and Summary

During the Mitigation review, the Zenith team determined that 5 issues from the original audit were either partially mitigated or had a mitigation error. The table below provides details regarding the status of each in-scope vulnerability from the original audit, followed by full details on the vulnerabilities that were not fully mitigated.

MITIGATION	PURPOSE	ORIGINAL ISSUE	STATUS
PR 1565	`ecrecover` validation	#13	🔴 Partially Mitigated
PR 1566	Codehash commitment	#10	🔴 Mitigation Error
PR 1567	Cases with Cairo `precompiles` and `delegatecalls`	#38, #124	🟢 Mitigation Confirmed
PR 1569	DualVMToken `totalSupply` fixes	#32	🟢 Mitigation Confirmed
PR 1573	`felt_to_bytes_little` loop stop condition	#118	🟢 Mitigation Confirmed
PR 1575	Fix off by one error `ripemd-160`	#50	🟢 Mitigation Confirmed
PR 1577	Finalize dictionary in `RIPEMD160`	#54 , #120	🟢 Mitigation Confirmed
PR 1579	SSJ pow	#65	🟢 Mitigation Confirmed

MITIGATION	PURPOSE	ORIGINAL ISSUE	STATUS
PR 1581	Decode legacy chain id overflow	#69	🟢 Mitigation Confirmed
PR 1582	Reentrancy check is moved at Kakarot level in `eth_call`	#64	🟢 Mitigation Confirmed
PR 1584	`handle_l1_message` gas price set to `0`	#29	🔴 Mitigation Error
PR 1587	Set `code_address` of the deployment code to the created address	#44	🟢 Mitigation Confirmed
PR 1592	Use `default_dict_finalize` with `0`	#91	🟢 Mitigation Confirmed
PR 1593	Last 10 blockhash are `0`	#5	🔴 Mitigation Error
PR 1603	Remove useless computation in `uint256_fast_exp`	#2	🟢 Mitigation Confirmed
PR 1604	High offset memory expansion computations	#4	🟢 Mitigation Confirmed
PR 1606	Update base fee starting from next block only	#28	🟢 Mitigation Confirmed

MITIGATION	PURPOSE	ORIGINAL ISSUE	STATUS
PR 1611	Remove dead code in `exec_create`	#70	🟢 Mitigation Confirmed
PR 1612	Return uncapped `return_data_len` from `execute_from_outside`	#24	🟢 Mitigation Confirmed
PR 1616	Add checks on Starknet return values for DualVMToken	#51, #42	🔴 Mitigation Error
PR 1618	Selfdestruct send to burn address	#83	🟢 Mitigation Confirmed
PR 1619	Memory expansion early return condition	#101 , #71	🟢 Mitigation Confirmed
PR 1622	Check kakarot approval `dualVMToken`	#48 , #116	🟢 Mitigation Confirmed
PR 1623	Remove messaging	#111, #105	🟢 Mitigation Confirmed
PR 1625	Use Cairo1Helpers class for `call_contract` for future-proofness	#49	🟢 Mitigation Confirmed
PR 1626	Underconstrained hint in `initialize_jumpdests`	#26 , #67	🟢 Mitigation

MITIGATION	PURPOSE	ORIGINAL ISSUE	STATUS
			Confirmed
PR 1627	OOB read in `parse_storage_keys` on malformed inputs	#81	🟢 Mitigation Confirmed
PR 1633	Check `call_array` within bounds `calldata`, `stack_size_diff`, remove setter native token and raise for invalid `y_parity` values	#66	🟢 Mitigation Confirmed
PR 1635	Validate `s <= N//2` when recovering tx sender	#98, #125	🟢 Mitigation Confirmed
PR 1636	Adds carry check add transfer and remove unnecessary `memcpy`	#128	🟢 Mitigation Confirmed
PR 1637	Remove unecessary `charge_gas` and prefund precompiles	#46	🟢 Mitigation Confirmed
lib PR 12	Remove messaging	Part of PR 1623 for issues #111 and #105	🟢 Mitigation Confirmed
lib PR 13	Check `data.length` and `pendingWordLen`	#77, #86, #99 and #100	🟢 Mitigation Confirmed
ssj PR 1022	SSJ pow	Required for PR 1625	🟢 Mitigation Confirmed

MITIGATION	PURPOSE	ORIGINAL ISSUE	STATUS
ssj PR 1023	Add `new_call_contract_syscall` to Cairo1Helpers class	Required for PR 1625	🟢 Mitigation Confirmed
PR 1621	warmup all precompiles	#58	🟢 Mitigation Confirmed

[PR 1565] Stricter validation on `v`, `r`, `s`

Context

Several findings were reported concerning overly permissive validation of signatures on Kakarot:

- Lack of validation of `s`'s range within ecrecover leads to some invalid signatures passing, reported in Issue #13.
- Lack of validation of `r`'s valid range within `ecrecover` leads to RPC-level reverts, reported in Issue #9.
- The finding above also mentions that an ECDSA signature must meet the constraint `0 < s < secp256k1n / 2 + 1` to be valid. This is only the case for transaction signatures, and not for `ecrecover`.
- Invalid values for `v` accepted as valid in `verify_eth_signature_uint256` for Kakarot transactions, which we reported in QA-05.

Description

The mitigation in PR 1565 addresses both findings, but signature validation in `verify_eth_signature_uint256` remains overly permissive. We believe this is unlikely to pose a security risk, but it represents a deviation from spec.

Recommendation

We recommend strictly constraining values of `r`, `s` and `v` to their respective ranges in `verify_eth_signature_uint256` to ensure the range of accepted signatures on Kakarot matches that of Ethereum.

Client

We did not see it, as several issues were marked as duplicate, but were not really. It should be fixed with PR 1635 and PR 1633.

Zenith

Confirmed fixes.

[PR 1566] `Extcodehash` is fixed only partially

Context

It was reported in Issue #10 that `extcodehash` incorrectly returned `0` instead of `keccak256("")` in some cases. The fix applied by the team was slightly different from the one recommended in the original finding.

Description

It is still possible that `extcodehash` returns `0` for EVM accounts with non-zero balance, if these accounts were deployed with a `kakarot.deploy_externally_owned_account` call.

In this case:

- Manually calling `deploy_externally_owned_account` on an EVM address will create the account with the stored hash at 0.
- Transferring native tokens to it via the Starknet ERC20 (may also work via EVM transfer but not sure) will not change the code hash.
- When calling `extcodehash` on this address, because the contract is found as deployed already, its stored code hash is returned, which is 0.

Recommendation

We recommend adding the `0` or `keccak256("")` selection in `extcodehash` instead of `Starknet._commit_account`, so also the case of accounts deployed outside a Starknet commit are properly handled.

Client

Initially fixed by PR 1644; superceded by PR 1654.

Zenith

The PR looks good to us.

EV_om (Zenith) commented:

To clarify: the recommendation is to never store `keccak256("")` as codehash. Instead, have `extcodehash` return either `0` or `keccak256("")` depending on the account's state, and do not store a variable that can change dynamically based on external conditions.

[PR 1584] New L1 EVM gas limit allows exceeding block gas limit

Context

Issue #29's group addressed excessively restrictive gas/gas price allocations for EVM transactions initiated from L1.

Description

With the applied fix, `2^{64}-1` is the new gas limit for EVM transactions originated from L1. While this is not necessarily a problem as Starknet computation limits still apply, it introduces a way around the block gas limit enforced on transactions sent on the Kakarot RPC:

```
File: eth_rpc.cairo
        let (block_gas_limit) = Kakarot.get_block_gas_limit();
        with_attr error_message("Transaction gas_limit > Block gas_limit")
{
            assert_nn(block_gas_limit - tx.gas_limit);
        }

```

... with `block_gas_limit` being configurable.

Recommendation

We recommend giving L1 transactions `Kakarot.get_block_gas_limit()` gas instead of `2^{64}-1`.

Client

Addressed with #1623.

Zenith

Related to deleted code.

[PR 1593] Off-by-one error in blockhash fix

Context

Issue #10 finding reported a cairo-level crash due to the possibility of looking up block hashes of recent Starknet blocks. The team addressed the issue by applying the fix recommended in the original report.

Description

Quoting the Starknet docs we have:

`Block number out of range` -> `block_number` is greater than `current_block - 10`.

From this, we see that the first value that is out of range and generates an error on Starknet is `current_block - 9`.

The fixing code checks this range as follows:

```
File: block_information.cairo
        let in_range = is_in_range(
            block_number.low, lower_bound, evm.message.env.block_number - 10
        );

        if (in_range == FALSE) {
            Stack.push_uint256(Uint256(0, 0));
            return ();
        }
```

Now if we look at the `is_in_range` Cairo function, we have that the range check is done with the `upper` value being exclusive, not inclusive, so `upper` has to be the first value out of range:

```
File: math_cmp.cairo
// Returns 1 if value is in the range [lower, upper).
// Returns 0 otherwise.
// Assumptions:
//   upper - lower <= RANGE_CHECK_BOUND.
@known_ap_change
func is_in_range{range_check_ptr}(value, lower, upper) -> felt {
```

There is, therefore, an off-by-one error and the fix applied is overdone by one block.

Recommendation

The correct fix would in our opinion would be:

```
        let in_range = is_in_range(
              block_number.low, lower_bound, evm.message.env.block_number - 9
          );
```

This is further confirmed by the fact that the newly introduced test checks `block_number - 9` up to `block_number`, so `block_number - 10` should be considered in range (and looked up) and not out of range (and shorted to 0).

Client

Fixed by PR 1643.

Zenith

Confirmed as fixed.

[PR 1616] Checking return value of ERC20 calls breaks compatibility with tokens that lack one

Context:

PR 1616 introduced verification of the return value of ERC20 operations in `DualVmToken.sol`. This enables the contract to safely support ERC-20s that don't revert on failure and return `false` instead.

Description

The fix breaks another ERC-20 behaviour that was originally intended to be supported (Missing return values).

Recommendation

We assume this change was intentional and this behaviour is no longer supported by the DualVmToken contract. If it needs to be, then the length of the return data must be checked before decoding it.

We also recommend modifying the revert condition here to ensure consistency between errors thrown by `transfer()` and `transferFrom()`.

Client

Acknowledged. For all `dualVMtoken` token deployed, the underlaying token will be checked before deployment to ensure there are compatible.

Zenith

Roger the acknowledged and watch out plan, seems very reasonable.

Disclosures