zkSync Era Security Audit

: Bug Fixes Review

September 24, 2024

Revision 1.0

ChainLight@Theori

Table of Contents

Executive Summary

Starting on September 17, 2024, ChainLight of Theori reviewed the recent security bug findings and fixes in the zkSync Era circuits. The vulnerabilities were identified by Matter Labs and/or external reporters. The purpose of ChainLight's review was to identify the root cause, to analyze potential variants, and to review and suggest remediations.

Audit Overview

Scope

Name	zkSync Era Security Audit
Target / Version	• Git Repository (matter-labs/zksync-protocol): commit `1bfb6af108c187f8a153afa39bbff0b40bdc4831` • Git Repository (matter-labs/zksync-crypto): commit `542a1a11d57963ace2446c03f2552b74289ec18d` • Git Repository (matter-labs/zksync-crypto-fork): commit `73782316774820f6773a1025c23a47ce6c786c50` • Git Repository (matter-labs/era-zk_evm): commit `f1db9573a645f48e260649eeaaaf93b5c2ed473a`
Application Type	ZK Circuit
Lang. / Platforms	ZK Circuit [Boojum]

Code Revision

N/A

Severity Categories

Severity	Description
Critical	The attack cost is low (not requiring much time or effort to succeed in the actual attack), and the vulnerability causes a high-impact issue. (e.g., Effect on service availability, Attacker taking financial gain)
High	An attacker can succeed in an attack which clearly causes problems in the service's operation. Even when the attack cost is high, the severity of the issue is considered "high" if the impact of the attack is remarkably high.
Medium	An attacker may perform an unintended action in the service, and the action may impact service operation. However, there are some restrictions for the actual attack to succeed.
Low	An attacker can perform an unintended action in the service, but the action does not cause significant impact or the success rate of the attack is remarkably low.
Informational	Any informational findings that do not directly impact the user or the protocol.
Note	Neutral information about the target that is not directly related to the project's safety and security.

Status Categories

Status	Description
Reported	ChainLight reported the issue to the client.
WIP	The client is working on the patch.
Patched	The client fully resolved the issue by patching the root cause.
Mitigated	The client resolved the issue by reducing the risk to an acceptable level by introducing mitigations.
Acknowledged	The client acknowledged the potential risk, but they will resolve it later.
Won't Fix	The client acknowledged the potential risk, but they decided to accept the risk.

Finding Breakdown by Severity

Category	Count	Findings
Critical	0	• N/A
High	4	• `BUGFIX-001` • `BUGFIX-002` • `BUGFIX-003` • `BUGFIX-004`
Medium	0	• N/A
Low	2	• `BUGFIX-007` • `BUGFIX-008`
Informational	2	• `BUGFIX-009` • `BUGFIX-010`
Note	0	• N/A

Findings

Summary

#	ID	Title	Severity	Status
1	BUGFIX-001	SHA256: add range-checks to make gad get sound	High	Patched
2	BUGFIX-002	Boojum/U32: range-check output of sub in div_by_const	High	Patched
3	BUGFIX-003	Boojum/U8: range-check output of sub_no_overflow	High	Patched
4	BUGFIX-004	Poseidon2 BN254 implementation bug	High	Patched
5	BUGFIX-007	ZkSync implementation of the permutation argument is weak	Low	Patched
6	BUGFIX-008	Small unsoundness in the recursive verifier	Low	Won't Fix
7	BUGFIX-009	Wrong parameters in gates.	Informational	Patched
8	BUGFIX-010	Exception-free formulas for mixing points are broken if the point is infinity	Informational	Mitigated

#1 `BUGFIX-001` SHA256: add range-checks to make gadget sound

ID	Summary	Severity
`BUGFIX-001`	The aligned_variables from `split_and_rotate`, used in the SHA256 gadget, have to be u4s. For most cases, this was done by the call to xor. But for the ones stored in `t1_shifted_10`, some variables are not checked, as they are overwritten.	**High**

Description

The `split_and_rotate` function takes in a variable containing a 32-bit value and outputs 8 variables intended to represent the 4-bit chunks of the 32-bits after a constant rotation is applied. However, `split_and_rotate` itself doesn't actually range check the output variables -- it only checks that they combine to the input value via `ReductionGate`s. In almost every case, `split_and_rotate` doing these 4-bit range checks would be redundant, as the outputs are typically passed to `tri_xor_many` which uses a lookup table that also acts as range checks.

This issue stems from one case where some output variables from `split_and_rotate` were never range constrained:

```
let (t1_rotated_10, _, t1_rotated_10_decompose_high) = split_and_rotate(cs
, t1, 10);

let mut t1_shifted_10 = t1_rotated_10;
t1_shifted_10[7] = zero;
t1_shifted_10[6] = zero;
t1_shifted_10[5] = t1_rotated_10_decompose_high;
...
let s1_chunks = tri_xor_many(cs, &t1_rotated_17, &t1_rotated_19, &t1_shift
ed_10);
```

As some of the original variables in `t1_rotated_10` are immediately overwritten, they are never range constrained, which allows the other values in `t1_rotated_10` to be invalid.

After review, we concluded that this was the only case where the `split_and_rotate` result wasn't directly passed into `tri_xor_many`.

Impact

High

These unconstrained variables allow invalid SHA256 hashes to be computed.

Recommendation

In addition to the provided fix, we have some long-term recommendations to reduce the chance of similar issues:

1. Functions should clearly document their assumptions about range constraints. For example, `split_and_rotate` should inform its callers that the output variables must be separately range constrained.
2. The circuit synthesis could dynamically track range constraints that are (allegedly) applied to variables. Specifically, each `Variable` can have a corresponding interval that is tracked. Circuit gates and lookup tables can adjust the interval of each variables they operate on. `Variable`s can also be annotated with their expected/required range constraints, which are verified at the end of circuit synthesis using their corresponding intervals. This scheme would accomplish two goals: (a) enable detection of missing range constraints during circuit synthesis without introducing new constraints, and (b) narrow the scope of code that must be reviewed for range constraint correctness.

References

N/A

Remediation

Patched

The issue was fixed in commit `a548279113178128dde7cf99c5fb20bc16440b0d` of `matter-labs/zksync-crypto-fork`.

The fix uses `tri_xor_many` to range check the three unchecked values before they are overwritten:

```
// These positions need to be range-checked manually to ensure
// [split_and_rotate] is sound.
let _ = tri_xor_many(
```

```
    cs,
    &[t1_rotated_10[7]],
    &[t1_rotated_10[6]],
    &[t1_rotated_10[5]],
);
```

#2 `BUGFIX-002` Boojum/U32: range-check output of sub in div_by_const

ID	Summary	Severity
`BUGFIX-002`	Missing range constraint on remainder check in `div_by constant`.	High

Description

The `UIntXAddGate` is described as follows:

```
// a + b + carry_in = c + 2^N * carry_out,
// `carry_out` is boolean constrainted
// but `c` is NOT. We will use reduction gate to perform decomposition of
`c`, and separate range checks
```

It's worth noting that the gate itself also does not range constrain `a` or `b`.

`UIntXAddGate::perform_subtraction_with_expected_borrow_out(a, b, borrow_in, zerovar, borrow_out)` creates the following gate:

```
Self {
    a: output_variable,
    b,
    carry_in: borrow_in,
    c: a,
    carry_out: borrow_out
}
```

and thus does not itself range constrain the `output_variable`.

In `Uint32::div_by_constant`, the remainder after division should be constrained to be lower than the divisor. To implement that, the code uses `perform_subtraction_with_expected_borrow_out` to check that the output carry flag is true when computing `remainder - constant`:

```
let _ = UIntXAddGate::<32>::perform_subtraction_with_expected_borrow_out(
    cs,
    remainder.variable,
    allocated_constant.variable,
    no_borrow,
    no_borrow,
    boolean_true.variable,
);
```

However, since the `output_variable` is not range constrained and is ignored in `UInt32::div_by_constant`, this fails to constrain the size of the remainder. Specifically, a malicious prover can assign `(remainder - constant + 2**32) mod p` to the `output_variable` to satisfy the gate even if `remainder < constant`.

Impact

High

The issue allows invalid division results to be proven, which impacts many parts of the circuit code. A notable example is determining which memory cell is accessed during a memory read or write:

```
let (cell_idx, unalignment) = offset.div_by_constant(cs, 32);
```

Recommendation

N/A

References

N/A

Remediation

Patched

The issue was fixed in commit `b2afca2d77045f692bf6abe5e1d4baaf871009af` of `matter-labs/zksync-crypto-fork`.

The provided fix uses `UInt32::from_variable_checked` to range constrain the subtraction result.

#3 BUGFIX-003 Boojum/U8: range-check output of sub_no_overflow

ID	Summary	Severity
BUGFIX-003	Brief explanation	High

Description

This issue is similar to BUGFIX-002. In `U8::sub_no_overflow`, a subtraction result from `UIntXAddGate` is used without a range check:

```
let no_borrow = cs.allocate_constant(F::ZERO);
let result_var = UIntXAddGate::<8>::perform_subtraction_no_borrow(
    cs,
    self.variable,
    other.variable,
    no_borrow,
    no_borrow,
);

let result = Self {
    variable: result_var,
    _marker: std::marker::PhantomData,
};
```

Impact

High

`U8::sub_no_overflow` is used in the keccak256 round function, so this issue can allow invalid keccak hashes to be computed.

Recommendation

N/A

References

N/A

Remediation

Patched

The issue was fixed `73782316774820f6773a1025c23a47ce6c786c50` of `matter-labs/zksync-crypto-fork`. The fix adds a range check to the subtraction result:

```
let result = Self::from_variable_checked(cs, result_var);
```

#4 `BUGFIX-004` Poseidon2 BN254 implementation bug

ID	Summary	Severity
`BUGFIX-004`	`Poseidon2Params::defaults` zeros out the wrong round constants.	High

Description

The partial rounds of Poseidon2 should applied in between the first half and second half of the full rounds, as described in Section 6 of https://eprint.iacr.org/2023/323.pdf. The default round constants are generated via a deterministic hashing procedure, and the code was previously modifying the round constants at the beginning (rather than in between the sets of full rounds):

```
let mut round_constants = params.round_constants().to_owned();
for i in 0..params.partial_rounds {
   for j in 1..WIDTH {
       round_constants[i][j] = E::Fr::zero();
    }
}
```

Impact

High

Using the wrong parameters can greatly reduce the security of the Poseidon hash family.

Recommendation

N/A

References

https://eprint.iacr.org/2023/323.pdf

Remediation

Patched

The issue was fixed in commit `542a1a11d57963ace2446c03f2552b74289ec18d` of `matter-labs/zksync-crypto`. The fix changes the offset of the round constants being overwritten:

```
let mut round_constants = params.round_constants().to_owned();
for i in (full_rounds / 2)..(full_rounds / 2) + partial_rounds {
    for j in 1..WIDTH {
        round_constants[i][j] = E::Fr::zero();
    }
}
```

#5 `BUGFIX-007` ZkSync implementation of the permutation argument is weak

ID	Summary	Severity
`BUGFIX-007`	If proving a batch with very large queues, the security of the permutation argument can reach as low as 70 bits.	Low

Description

The soundness of the zkSync Era circuits relies on various queues (e.g. storage writes and RAM access) being properly constrained. This is accomplished by sorting the queue (out of circuit) according to a simple ordering and verifying consistency by iterating the sorted queue. Constraining the order of the sorted queue is easy to do in-circuit, but verifying it contains the same elements as the original queue is done with a *permutation argument*.

The permutation argument constructs two polynomials whose roots are the encodings of the elements of their respective queue, such that the queues are equal if and only if the polynomials are equal. Let $q_o(x) \in \mathbb{F}_p[x]$ be the polynomial for the original queue and $q_s(x) \in \mathbb{F}_p[x]$ be the polynomial for the sorted queue.

Deterministically checking if $q_o(x) = q_s(x)$ is expensive, but it can be approximated by random evaluations. Let $p(x) = q_s(x) - q_o(x)$, so we want to check if $p(x) = 0$. The Schwartz-Zippel Lemma tells us that if $p \neq 0$, $\deg(p) \leq d$, and r is a uniformly random value in $\mathbb{F}_p$, then $\Pr[p(r) = 0] \leq \frac{d}{p}$.

This means we can approximate checking if $q_o(x) = q_s(x)$ by evaluating $p(x)$ on a random point and confirming it is zero. If this random evaluation is performed k times, the probability of a false positive is at most $\left(\frac{d}{p}\right)^k$.

As usual in ZK proofs, the random point selection is replaced with the Fiat-Shamir heuristic, wherein the points are selected by hashing the input values.

zkSync Era's proof system uses $p = 2^{64} - 2^{32} + 1$ and repeats the random evaluation $k = 2$ times. Since the queue polynomials are degree at most 2^{32} (the maximum length of a Boojum

queue), the false-positive probability is roughly $\left(\frac{2^{32}}{2^{64}}\right)^2 = 2^{-64}$. This means an attacker brute forcing an invalid sorted storage queue takes an expected 2^{64} attempts.

This is notably lower than other parts of the system, but is still sufficiently high that an attack is unlikely, especially given the high fixed cost of each attempt. Nonetheless, the permutation argument is a weak link that is easy to strengthen.

Impact

Low

Describe the impact of the bug

Recommendation

There are two primary ways to increase the security of the permutation check.

1. Increase the number of evaluations
2. Perform the evaluation(s) over a field extension

Option 2 has better probability scaling properties. Specifically, the false-positive probability when using a degree-k extension is $\frac{d}{p^k}$, which is a factor of d^{k-1} better than doing k base field evaluations.

However, option 1 is a significantly simpler change to the codebase. An single evaluation ($k = 3$) will increase the security budget by at least 2^{32} without introducing additional complexity.

If option 2 is chosen, we recommend having the field extension evaluation closely audited to ensure the additional complexity does not introduce new vulnerabilities.

A third, supplemental improvement would be to constrain the queue length to be at most 2^{24}, which should be sufficiently large for any batch being proven in practice, but further increase the security budget by 8 bits per evaluation.

References

https://en.wikipedia.org/wiki/Schwartz%E2%80%93Zippel_lemma

Remediation

Patched

Commit `ec69d6f0e13f0cdf67471fb12ec0e098aea973e4` of `matter-labs/zksync-protocol-25-09-2024` increases the number of evaluations to 3, upgrading the worst-case

security level to 96 bits.

#6 BUGFIX-008 Small unsoundness in the recursive verifier

ID	Summary	Severity
BUGFIX-008	The transcript challenge bits can be slightly influenced by a malicious prover.	Low

Description

The recursive verifier uses the Fiat-Shamir heuristic to generate challenges for the proof verification routine. Some of these challenges come in the form of indexes in binary trees, so the challenges are generated in a bit-decomposed form which is interpreted as the path through the tree. To generate these path challenges, `BoolsBuffer::get_bits` is used:

```
    pub fn get_bits<CS: ConstraintSystem<F>, T: CircuitTranscript<F>>(
        ...
    ) -> Vec<Boolean<F>> {
        ...
            // get 1 field element form transcript
            let field_el = transcript.get_challenge(cs);
            let el_bits = field_el.spread_into_bits::<CS, 64>(cs);
            let mut lsb_iterator = el_bits.iter();
        ...
    }
```

Here, `transcript.get_challenge` returns an element in the Goldilocks field ($p = 2^{64} - 2^{32} + 1$), and `spread_into_bits<CS, 64>` returns a 64-bit decomposition of the field element. However, there are $2^{64} - p = 2^{32} - 1$ many elements that have *two* valid 64-bit decompositions in the field, namely for all $0 \leq i < 2^{32-1}$, the integer bit decompositions for i and $p + i$ both satisfy the constraints for `spread_into_bits<CS, 64>`.

This means that if the transcript challenge returns a value in this range, an attacker gets 1 bit of control over the challenge bits, i.e. they can choose which of the two valid decompositions to assign to the witness.

Impact

Low

For a given transcript, the probability that the transcript challenge is vulnerable is roughly 2^{-32} and they are only given 1 bit of control over the path challenge. While even this level of control should be eliminated, it is extremely unlikely to be exploitable in practice.

Recommendation

It is difficult to fix this issue without introducing a nontrivial number of additional constraints. One possibility is to reject the invalid bit decompositions as follows:

1. Split the field element into 32-bit limbs (with range checking)
2. Constrain that if the high limb is `0xffffffff`, then the low limb must be `0`
3. Decompose each 32-bit limb and concatenate the results.

Additionally, `spread_into_bits<CS, N>` could be updated to require $N < log_2(p)$ during circuit synthesis to help catch similar issues in the future.

References

N/A

Remediation

Won't Fix

As this issue is difficult to exploit, it was decided to not fix it immediately.

#7 BUGFIX-009 Wrong parameters in gates.

ID	Summary	Severity
BUGFIX-009	Some gates have their `max_constraint_degree` configured incorrectly.	Informational

Description

In Boojum, each gate type has an associated `GateConstraintEvaluator` which provides some metadata about the gates and implements the gate evaluation at proving time. The `GateConstraintEvaluator::gate_purpose` function returns the metadata information while the `GateConstraintEvaluator::evaluate_once` function implements the actual evaluation. For example, the `BooleanConstaintEvaluator` has the following simple definitions:

```
impl<F: PrimeField> GateConstraintEvaluator<F> for BooleanConstraitEvaluat
or {
    ...
    fn gate_purpose() -> GatePurpose {
        GatePurpose::Evaluatable {
            max_constraint_degree: 2,
            num_quotient_terms: 1,
        }
    }
    ...
    fn evaluate_once<
        P: field::traits::field_like::PrimeFieldLike<Base = F>,
        S: TraceSource<F, P>,
        D: EvaluationDestination<F, P>,
    >(
        &self,
        trace_source: &S,
        destination: &mut D,
        _shared_constants: &Self::RowSharedConstants<P>,
        _global_constants: &Self::GlobalConstants<P>,
        ctx: &mut P::Context,
    ) {
```

```
        let one = P::one(ctx);
        let a = trace_source.get_variable_value(0);
        let mut tmp = one;
        tmp.sub_assign(&a, ctx);

        let mut contribution = a;
        contribution.mul_assign(&tmp, ctx);

        destination.push_evaluation_result(contribution, ctx);
    }
}
```

This gate implements the common boolean constraint `a * (1-a) == 0`, which has degree 2, as reflected by the `max_constraint_degree` in the `GatePurpose`.

Several gates were found to return the incorrect value for `max_constraint_degree`:

- `U32SubConstraintEvaluator` returns 1, but it should be 2 as it contains a boolean constraint for a carry bit within it.
- `U32AddConstraintEvaluator` returns 1, but it should be 2 as it contains a boolean constraint for a carry bit within it.
- `U32TriAddCarryAsChunkConstraintEvaluator` returns 2, but it should be 1 as all terms are products of a global constant and variable.

Impact

Informational

This gate degree information is used during setup to assign selectors, as explained in the documentation in `CSReferenceAssembly::compute_selector_and_constants_placement`:

```
// every gate has a specific degree that it evaluates too,
// and potentially non-trivial selector's path that
// looks like unbalanced tree

//             X
//       sel0   (1-sel0)
//   sel1  (1-sel1)  sel1   (1-sel1)
//    G0     G1       G2    sel2   (1-sel2)
//                          G3       G4
```

```
// and our task is to find placements of gates and selectors
// such that it leads to the minimal value of
// degree(gate) * len(selectors path) from the root
```

The degree in `GatePurpose` is used to search for the assignment of selectors which minimizes the total degree of the resulting constraint, including selectors. The maximum degree of the resulting constraint is then used to determine the `quotient_degree` value used in the polynomial IOP.

Hence, `max_constraint_degree` values which are too large could result in non-optimal selector assignment. Worse yet, values which are too small could result in miscalculation of the total degree of the constraint (including selectors) and hence also the `quotient_degree`, which could have implications for the security of the proof system. However, whether this occurs is dependent on the selector assignment and may require significantly low values as the `quotient_degree` is rounded up to the nearest power of two. Note that the selector assignment is dependent on the set of gate types used, and so will vary circuit-to-circuit.

Recommendation

Correct the `max_constraint_degree` value for all gates identified in the description above.

References

N/A

Remediation

Patched

This issue was fixed in commit `3f77d715b27bebe28fa8f53861324f1d4284c19b` of `matter-labs/ zksync-crypto-fork`.

#8 `BUGFIX-010` Exception-free formulas for mixing points are broken if the point is infinity

ID	Summary	Severity
`BUGFIX-010`	`SWProjectivePoint::add_mixed` expects the argument point is a valid affine point (i.e. not the point at infinity), but is occasionally used with invalid affine points.	Informational

Description

Various circuits (for instance, the `ecrecover` circuit) in zkSync Era must constrain the result of elliptic curve arithmetic over non-native fields. A common operation needed is adding two elliptic curve points, and the preferred algorithm for a ZK-circuit implementation is an exception-free mixed point addition formula, as it uses fewer constraints than other methods. Here, "mixed point" means that one of the points is represented in projective coordinates while the other is represented in affine coordinates. Recall that affine coordinates are unable to represent the additive identity in an elliptic curve group, i.e. the point at infinity.

As a result, using mixed point addition requires knowing that one of the points cannot be the identity element. However, since ZK-circuits are uniform computation models, the circuit cannot "choose" not to use mixed point addition if it encounters a point at infinity. Instead, it must pass a *dummy* input into the mixed point addition and ignore the result via a conditional select.

An example of this approach is in `ecrecover`'s fixed-based point multiplication. This performs a base-256 optimized multiplication of the generator with a scalar. Each iteration uses a lookup-table to lookup an affine point, where the point at infinity is represented by the dummy value (x, y) where $x = y = 0$, and the point is added to an accumulator. Below is the accumulation code:

```
let new_acc = acc.add_mixed(cs, &mut (x, y));
let should_not_update = byte.is_zero(cs);
acc = Selectable::conditionally_select(cs, should_not_update, &acc, &new_a
cc);
```

As you can see, the result of the `add_mixed` is ignored if the lookup byte is zero, directly corresponding to when the `(x, y)` are not a valid affine point.

All other uses of `add_mixed` either operate similarly, or are guaranteed the affine point is valid by some other means.

Impact

Informational

This issue was assessed to have no security impact on the current circuits. However, some mitigations can ensure issues don't arise in future versions of the circuits.

Recommendation

This common pattern of an `add_mixed` followed by a `conditionally_select` could become the default interface for point addition. Specifically, the `add_mixed` function could be rewritten as follows:

```
pub fn add_mixed<CS: ConstraintSystem<F>>(
    &mut self,
    cs: &mut CS,
    other_xy: &mut (NN, NN),
    is_infinity: Boolean<F>
) -> Self {
    let res = self.add_sub_mixed_impl(cs, other_xy, false);
    Selectable::conditionally_select(cs, is_infinity, self, &res)
}
```

Note that the `conditionally_select` could in principle be optimized to not create any constraints if the `is_infinity` flag is a constant value, allowing this interface to have zero overhead for cases where `other_xy` is guaranteed to be valid.

References

Algorithm 8 from https://eprint.iacr.org/2015/1060.pdf

Remediation

Mitigated

There was no current security impact of this code pattern. Commit `3f77d715b27bebe28fa8f53861324f1d4284c19b` of `matter-labs/zksync-crypto-fork` adds documentation to help avoid potentially dangerous use of `add_mixed`.

Revision History

Version	Date	Description
1.0	Sep 24, 2024	Initial version

Theori, Inc. (“We”) is acting solely for the client and is not responsible to any other party. Deliverables are valid for and should be used solely in connection with the purpose for which they were prepared as set out in our engagement agreement. You should not refer to or use our name or advice for any other purpose. The information (where appropriate) has not been verified. No representation or warranty is given as to accuracy, completeness or correctness of information in the Deliverables, any document, or any other information made available. Deliverables are for the internal use of the client and may not be used or relied upon by any person or entity other than the client. Deliverables are confidential and are not to be provided, without our authorization (preferably written), to entities or representatives of entities (including employees) that are not the client, including affiliates or representatives of affiliates of the client.