Semaphore PSE audit

By Mridul, Yufei Li, Kyle Charbonnet

March 2024

- Semaphore PSE audit
  - Overview
    - Background
    - Scope
      - Semaphore
      - Zk-kit
  - Findings
    - Critical Severity
      - 1. Missing admin access check
      - 2. Constrain babyjubjub secret scalar to be < `l`
      - 3. Private key can be out of range
    - High Severity
      - 1. `createGroup()` can be frontrun
      - 2. `inv()` returns garbage for `0n`
      - 3. `pow()` doesn't work for negative exponents
    - Medium severity
      - 1. Incorrect merkle proof length check
      - 2. Latest deleted leaf can be updated
    - Low severity
      - 1. Two-step admin update
      - 2. `F1Field` functions assume the inputs are in range
      - 3. f1-field tests are for composite order
    - Gas Optimizations
      - 1. `onlyExistingGroup(groupId) onlyGroupAdmin(groupId)` can be reduced to `onlyGroupAdmin(groupId)`
      - 2. Multiple `onlyExistingGroup(groupId)` checks
      - 3. Return `merkleTreeRoot` from internal functions like `_addMember()`
      - 4. Cache Merkle tree parameters
      - 5. `_insertMany()` can be further gas optimized
      - 6. Use `if` instead of `while` to increment `depth` in `InternalLeanIMT._insert()`
      - 7. `unchecked` for loop counter no longer required
      - 8. Solidity optimizer is turned off
    - Informational
      - 1. Define a const variable for `MAX_DEPTH`
      - 2. `binary-merkle-root` verifies zero root for `depth > MAX_DEPTH`
      - 3. Suggestions for variable renaming

- 4. Add visual explanation of LeanIMT
- 5. `_hash()` can shift by fewer bits
- 5. Use noble cryptography libraies

Overview

Background

Semaphore is a protocol, designed to be a simple and generic privacy layer for Ethereum DApps. Using zero knowledge, Ethereum users can prove their membership of a group and send signals such as votes or endorsements without revealing their original identity.

The three components of Semaphore are the smart contracts, core typescript, and the Circom circuits. The smart contracts manage creating and managing groups, and anonymous signaling. Circuit is used to prove that membership of a group and attaches the signaling to a particular scope. Typescript code enables smooth interaction with these systems.

Scope

Semaphore

Github repo: https://github.com/semaphore-protocol/semaphore/tree/v4.0.0-beta.1

Commit Hash: 8eb19e83fda62644872b2fcfbd85011d3b2c21e2

Files: Smart contracts, circuits, Typescript.

Zk-kit

Github repo: https://github.com/privacy-scaling-explorations/zk-kit/tree/imt.sol-v2.0.0-beta.8

Commit Hash: 215dfb30ba548918181419df5598d0a652901b7c

Files: InternalLeanIMT.sol, binary-merkle-root.circom, and related Typescript files.

Findings

Critical Severity

1. Missing admin access check

Context: SemaphoreGroups.sol#L83

Description: The `_addMember()` function is missing `onlyGroupAdmin(groupId)` check. Without it anyone can add members to the group:

Recommendation: Add `onlyGroupAdmin` modifier.

Implemted fix: PR 702.

2. Constrain babyjubjub secret scalar to be < `l`

Context: semaphore.circom#L44

Description: Following Geometry's bug disclosure (Circom_bug.pdf), it's known that babyjubjub secret key has to be constrained to be `< l`. `l` is as defined https://eips.ethereum.org/EIPS/eip-2494 and the same as `r` defined in Geometry's bug report. circomlib's BabyPbk() circuit (as of at the time of writing this issue) enforces the secret scalar to fit in 253 bits, where as `l` is of 251 bits.

Recommendation: Add a `LessThan(251)([secret, l]) === 1` constraint to make sure the secret scalar is `< l`.

Note: An implicit assumption on `LessThan(n)` inputs is that they fit in `n` bits. So, the correct fix is to first ensure that `secret` fits in 251 bits using `Num2Bits(251)` template. However, we can skip this check here. Reasoning: https://hackmd.io/@blockdev/Bkj0Qp8x0.

Implemented fix PR 743.

3. Private key can be out of range

Context: eddsa-poseidon.ts#L56

Description: Following Geometry's bug disclosure (Circom_bug.pdf), it's known that babyjubjub secret key has to be constrained to be `< l`. `l` is as defined https://eips.ethereum.org/EIPS/eip-2494 and the same as `r` defined in Geometry's bug report. zk-kit generates the private key such that it's outside this range (eddsa-poseidon.ts#L56):

```
return scalar.shiftRight(leBufferToBigInt(hash), BigInt(3))
```

This value is of 253 bit size, and we need this to fit in 251 bits.

Recommendation: To maintain backward compatibility, we cannot change the number of shifted bits. Update the return value to:

```diff
-return scalar.shiftRight(leBufferToBigInt(hash), BigInt(3))
+return scalar.shiftRight(leBufferToBigInt(hash), BigInt(3)) % subOrder
```

Implemented fix PR 257.

High Severity

1. `createGroup()` can be frontrun

Context: Semaphore.sol#L27, Semaphore.sol#L34

Description: A malicious actor can monitor calls of `createGroup(groupId, admin)` and frontrun it with `createGroup(groupId, admin1)`. This makes the attacker admin of the group, and reverts the genuine call.

Recommendation: Remove `groupId` from `createGroup()` argument. Create a storage variable `groupId` which maintains the next group ID to be assigned. `createGroup()` then assigns the new group this group ID and increments `groupId`, thus preventing frontrunning.

Implemented fix: PR 703.

2. `inv()` returns garbage for `0n`

Context: f1-field.ts#L88

Description: `inv(a: bigint)` is supposed to throw error for 0 (f1-field.ts#L83):

```
* If the input value is zero, which has no inverse, an error is thrown.
```

But it returns `0`. That's because `newr` is initialized to 0 and it nevers enters the loop(f1-field.ts#L88-L105):

```
inv(a: bigint): bigint {
    let t = this.zero
    let r = this._order
    let newt = this.one
    let newr = a % this._order

    while (newr) {
        const q = r / newr
        ;[t, newt] = [newt, t - q * newt]
        ;[r, newr] = [newr, r - q * newr]
    }

    if (t < this.zero) {
        t += this._order
    }

    return t
}
```

For reference, here's how it's implemented in ffjavascript(f1field_native.js#L132):

```
if (!a) throw new Error("Division by zero");
```

Recommendation: Throw error on `0` input.

Implemented fix: PR 255.

3. `pow()` doesn't work for negative exponents

Context: f1-field.ts#L199.

Description: `pow()` fn comment says it handles negative exponent but it doesn't. Negative exponents need special handling, so currently `pow()` runs in an infinite loop.

Recommendation: Following this logic for the fix:

$$a^{-e} = (a^{-1})^e \mod p$$

`pow(a, -e) == pow(inv(a), e)`.

So take the inverse of `a` if `e` is negative.

Implemented fix: PR 249.

Medium severity

1. Incorrect merkle proof length check

Context: InternalLeanIMT.sol#L221-L223

Description: Updating LeanIMT of size 1 (tree with 1 leaf inserted), `_update()` function reverts because of incorrect length check on merkle proof:

```
} else if (siblingNodes.length == 0) {
    revert WrongSiblingNodes();
}
```

This is because size 1 tree doesn't need additional data to calculate the root since the leaf is the root itself.

Recommendation: Remove this check.

Implemented fix: PR 220.

2. Latest deleted leaf can be updated

Context: InternalLeanIMT.sol#L211

Description: `_update(...)` updates `oldLeaf` to `newLeaf`, and sets `self.leaves[newLeaf]` to `self.leaves[oldLeaf]`. If `newLeaf` is `0`, it is now identified as part of the tree. So it becomes possible to update the `0` leaf since this check now passes:

```
} else if (!_has(self, oldLeaf)) {
    revert LeafDoesNotExist();
}
```

According to zk-kit team, updating deleted leaves is not an expected behavior.

Recommendation: Update `self.leaves[newLeaf]` only when `newLeaf != 0`. InternalLeanIMT.sol#L271:

```
+if (newLeaf != 0) {
    self.leaves[newLeaf] = self.leaves[oldLeaf];
+}
```

Now, `_has()` alwayas returns `false` for a `0` leaf.

Implemented fix: PR 229.

Low severity

1. Two-step admin update

Context: SemaphoreGroups.sol#L58

Description: The best practice for updating privileged roles is through a two-step process where the current admin first proposes a new address to toke the role, and then the new address accepts it via a transaction. This avoids mistakes from admin and avoids passing the role to a non-existent address.

Recommendation: Consider adding a two-step admin transfer similar to OpenZeppelin's Ownable2Step.sol.

Implemented fix: PR 718.

2. `F1Field` functions assume the inputs are in range

Context: f1-field.ts#L68

Description: Functions like `sub()` and `add()` will return values outside the field range if the inputs are outside the range:

For example, `add(_order, _order)` returns `_order`.

```
sub(a: bigint, b: bigint): bigint {
    return a >= b ? a - b : this._order - b + a
}

add(a: bigint, b: bigint): bigint {
    const res = a + b

    return res >= this._order ? res - this._order : res
}
```

Recommendation: Check that the inputs are in range, or provide clarifying comments.

Implemented fix: Added clarifying comments with PR 248.

3. f1-field tests are for composite order

Context: f1-field.test.ts#L7

Description: Describe the bug f1-field.test.ts uses 12 as the field order:

```
field = new F1Field(BigInt(12))
```

For a non-prime order, some elements may not have an inverse, f1-field.test.ts#L75:

```
expect(field.inv(a)).toBe(BigInt(1))
```

Recommendation: Change the order to a prime number.

Implemented fix: PR 252.

Gas Optimizations

1. `onlyExistingGroup(groupId) onlyGroupAdmin(groupId)` can be reduced to `onlyGroupAdmin(groupId)`

Context: SemaphoreGroups.sol#L61

Description: `onlyExistingGroup(groupId) onlyGroupAdmin(groupId)` checks that a group exists and the `msg.sender` is the group admin. Just checking for `onlyGroupAdmin(groupId)` is sufficient for this case as a group exists only when the group has a non-zero admin. Thus, if we ensure `msg.sender` is the admin, we implicitly ensure that the admin is non-zero.

Recommendation: Replace `onlyExistingGroup(groupId) onlyGroupAdmin(groupId)` with `onlyGroupAdmin(groupId)`.

Implemented fix: PR 702.

2. Multiple `onlyExistingGroup(groupId)` checks

Context: `validateProof()`

Description: `validateProof()` checks for existing group which then calls `verifyProof()` which agains ensures group's existence.

Recommendation: Remove `onlyExistingGroup(groupId)` modifier on `validateProof()`

Implemented fix: PR 702.

3. Return `merkleTreeRoot` from internal functions like `_addMember()`

Context: SemaphoreGroups.sol#L70, SemaphoreGroups.sol#L83

Description: In functions like `addMember()`, we currently have:

```
_addMember(groupId, identityCommitment);

uint256 merkleTreeRoot = getMerkleTreeRoot(groupId);
```

`merkleTreeRoot` can instead be returned from `_addMember` to save gas.

Recommendation: Return `merkleTreeRoot` from `_addMember()` and `_addMembers()`.

Implemented fix: PR 692.

4. Cache Merkle tree parameters

Context: InternalLeanIMT.sol#L93, InternalLeanIMT.sol#L49-L53

Description: `self.size` and `depth` can be cached to save storage reads in `_insert()` and `_insertMany()`.

`_insert()`:

```
while (2 ** self.depth < self.size + 1) {
    self.depth += 1;
}

uint256 index = self.size;
```

`_insertMany()`:

```
self.leaves[leaves[i]] = self.size + 1 + i;
```

Recommendation: Cache multiple storage reads to save gas.

Implemented fix: PR 221.

5. `_insertMany()` can be further gas optimized

Context: InternalLeanIMT.sol#L49-L51

Description: On top of PR 221 as a fix for above issue, there is this gas optimization to remove 1 `if` condition in the loop:

Recommendation: InternalLeanIMT.sol#L156-L172 can be updated to:

```
uint256 parentNode;

if ((i + nextLevelStartIndex) * 2 + 1 < currentLevelSize) {
    // Assign the right node if the value exists.
    uint256 rightNode = currentLevel[(i + nextLevelStartIndex) * 2 + 1 -
currentLevelStartIndex];
    // If it has a right child the result will be the hash(leftNode, rightNode)
    parentNode = PoseidonT3.hash([leftNode, rightNode]);
} else {
    // If right child doesn't exist, it will be the leftNode.
    parentNode = leftNode;
}
```

With L2s this gas difference shouldn't matter enough, and this may be hurting readability. So feel free to keep the code as-is.

Implemented fix: Keeping code as-is to prefer readability.

6. Use `if` instead of `while` to increment `depth` in `InternalLeanIMT._insert()`

Context: InternalLeanIMT.sol#L49-L51

Description: A new insertion can increase tree's depth by at most 1. We can be sure that this loop isn't executed twice:

```
while (2 ** self.depth < self.size + 1) {
  self.depth += 1;
}
```

Recommendation: Turn the `while` loop into an `if` condition saving gas.

Implemented fix: PR 219.

7. `unchecked` for loop counter no longer required

Description: Starting 0.8.22, there is no need to use `unchecked` for loop counters: https://soliditylang.org/blog/2023/10/25/solidity-0.8.22-release-announcement/.

Since zk-kit is supposed to be used with older versions, keeping code as-is may be a good idea though.

Recommendation: Consider removing `unchecked` for loop counters keeping in mind that it increases gas for pre 0.8.22 Solidity versions.

Implemented fix: Keeping the code as-is.

8. Solidity optimizer is turned off

Context: hardhat.config.ts#L18

Description: Semaphore is using Solidity with optimizer turned off.

Recommendation: Turn on Solidity optimizer as per Hardhat's documentation:

```
solidity: {
  version: "0.8.23",
  settings: {
    optimizer: {
      enabled: true,
      runs: 200
    }
  }
}
```

Implemented fix: PR 741.

Informational

1. Define a const variable for `MAX_DEPTH`

Context: Semaphore.sol#L139

Description: Since `12` will be updated as max depth is increased, it's better to define it as a const variable. It highlights the use of the constant and there is lower chances of forgetting it in case a modification is needed.

Recommendation: Use a constant variable for max depth.

Implemented fix: Fixed with PR 711.

2. `binary-merkle-root` verifies zero root for `depth > MAX_DEPTH`

Context: binary-merkle-root.circom#L40-L42

Description: For `depth > MAX_DEPTH`, `root` and `isDepth` below is calculated as `0`. Thus, `out = 0` will be always be verified successfully.

```
var isDepth = IsEqual()([depth, MAX_DEPTH]);

out <== root + isDepth * nodes[MAX_DEPTH];
```

Recommendation: Add a warning in comments for integrators.

Implemented fix: PR 211.

3. Suggestions for variable renaming

Context: InternalLeanIMT.sol#L49-L51

Description: Consider renaming these variables to `currentLevelNewNodes`, `numberOfNewNodes`, and `nextLevelNewNodes`: InternalLeanIMT.sol#L101:

```
uint256[] memory currentLevel;
```

InternalLeanIMT.sol#L125-L126:

```
uint256 numberOfNodes = nextLevelSize - nextLevelStartIndex;
uint256[] memory nextLevel = new uint256[](numberOfNodes);
```

Consider `currentLevelSize` which means the total number of nodes in the current level. `currentLevel` means the new nodes in the level. So semantically the naming can confuse. InternalLeanIMT.sol#L114:

```
uint256 currentLevelSize = self.size + leaves.length;
```

Recommendation: Consider renaming variables.

Implemented fix: PR 227.

4. Add visual explanation of LeanIMT

Context: InternalLeanIMT.sol#L49-L51

Description: The Lean incremental Merkle tree is an optimized version of the first version of the ZK-Kit binary Merkle tree. More documentation would help devs better understand how it works.

Recommendation: There's already an HackMD doc that explains it visually by @vplasencia. Adding that document to Github would be helpful for people.

Implemented fix: PR 231.

5. `_hash()` can shift by fewer bits

Context: Semaphore.sol#L192

Description: The goal with right shifting hash is to make it fit into Bn254 curve order. The curve order is of 254 bits ($log_2(p) = 253.59..$). So shifting by 3 bits is fine.

```
return uint256(keccak256(abi.encodePacked(message))) >> 8;
```

Practically, shifting by 8 bits is also secure enough. This issue is for reference and anyone else to chime in with their thoughts.

Recommendation: Shift by 3 bits instead.

Implemented fix: Kept as is since it's practically secure.

5. Use noble cryptography libraies

Description: Noble is a set of libraries providing "Audited & minimal JS implementation" of several cryptographic primitives implemented in zk-kit.

Some links: https://github.com/paulmillr/noble-curves, https://github.com/paulmillr/noble-hashes, https://paulmillr.com/noble/

It can be used to define babyjubjub curve, import poseidon, blake and sha hashes.

Recommendation: Consider using these libraries as they are audited and optimized, and it can fix known and unkown bugs (or devation from standard) in zk-kit: like how eddsa secret key is derived currently.

Note: This is discussed here: zk-kit#238.