# Semaphore Security Audit

By Kyle Charbonnet, Security Engineer @ EF's Privacy & Scaling Explorations Team

May, 2022

# Table of Contents

# Overview

## Executive Summary

The Semaphore code base was audited with a focus on both the smart contracts and the Circom circuits. The contracts followed solidity best practices with great documentation and NatSpec comments. Two major bugs were found, one for a Semaphore contract and another for an important dependency contract. Both of these have been fixed. There were no identified issues or concerns with the Circom circuits. Overall, the Semaphore codebase is simple, well written, and, in my opinion, ready for production.

## Background

Semaphore is a zero-knowledge protocol that acts as a generic privacy layer for Ethereum dApps. It allows users to prove they are members of a group without revealing their identity. They are also able to endorse an arbitrary string when they prove their membership. This functionality enables a wide range of use cases, from private voting to whistleblowing.

The two main components of Semaphore are the smart contracts and the Circom circuits. The smart contracts allow users to interact with the protocol on-chain. The circuits are used to generate zero-knowledge proofs that a user is part of a group without revealing their identity. The proof can then be verified through a Verifier smart contract on-chain.

Semaphore implements the protocol logic in SemaphoreCore.sol, SemaphoreGroups.sol, and Semaphore.sol. These contracts are meant to be extended by more detailed protocol contracts that need the Semaphore utility. For example, the Semaphore codebase also provides a private voting contract and a whistleblowing contract. These are just two examples of the applications that can be built on Semaphore.

## Coverage

Github Repo: https://github.com/semaphore-protocol/semaphore
Commit Hash: 339d3ac7da9dd22eeda46f418d7c48976724b1ca
Documentation: https://semaphore.appliedzkp.org/

All contracts in semaphore/contracts and all circuits in semaphore/circuits were reviewed. The following were focused on during the review of the smart contracts (but not limited to):

- Proper access control for handling group settings and membership
- Identity revealing information kept off-chain and private when proving membership
- Inability to successfully cast a vote or whistleblow when a proof is invalid
- Nullifier hashes were properly set when required
- Values were within range of the snark field when required
- Malicious members cannot compromise the privacy of others or halt the protocol

The following were focused on during the review of the circuits:

- The identity nullifier, identity trapdoor, merkle tree indices, and merkle tree siblings are private inputs
- The circuit cannot be rendered broken by invalid public inputs
- All variables are uniquely determined - there are no missing constraints
- All component inputs are restricted to their expected format

# Techniques Used

The following techniques were used for this audit - Static Analysis via Slither, Fuzzing Campaign via Echidna, Weak Verification via Ecne, and manual review. Manual review played the largest role in reviewing this codebase, but the other tools help rule out a wide range of potential bugs.

## Static Analysis - Slither

Slither is a tool provided by TrailOfBits. It was used on all of the smart contracts. There were many low level findings that were reviewed but determined not to be an issue. There were also 2 medium findings and 0 high findings.

| Finding | Priority | Action Needed |
|---|---|---|
| SemaphoreVoting.createPoll(uint256,address,uint8).poll (contracts/extensions/SemaphoreVoting.sol#51) is a local variable never initialized | Medium | No - default values are correctly assumed |
| Verifier.verifyProof(uint256[2][2], uint256[2],uint256[4]).proof (all verifiers in contracts/verifiers/) is a local variable never initialized | Medium | No - all 3 components are written to in the following 3 lines |

## Fuzzing Campaign - Echidna

Echidna is a tool provided by TrailOfBits. It was used to fuzz the SemaphoreGroups.sol properties. The results give more confidence that Semaphore groups respond properly to their state changing functions.

| Property | Result |
|---|---|
| A group's zero value cannot be changed once set | Passed |
| A group's depth value cannot be changed once set | Passed |
| A group's root always changes after addMember is called with any leaf but the 0 leaf | Passed |

| | |
|---|---|
| A group's 'number of leaves' value always increases by 1 after addMember is called | Passed |
| A group's root always changes after removeMember is called | Passed |
| A group's 'number of leaves' value never changes | Passed |

### Weak Verification of Circuits - Ecne

Ecne, a new tool from 0xPARC used for weakly verifying Circom circuits, was used on all circuits in Semaphore. The tool found that all variables in the circuits are uniquely determined, and therefore the circuits are weakly verified. This gives confidence that given a set of inputs, there is only one unique output to the circuit. Therefore, an attacker cannot use a given set of inputs to generate two different proofs with different outputs. More info on weak verification and Ecne can be found here: https://0xparc.org/blog/ecne

| **Ecne Findings** | |
|---|---|
| Uniquely Determined Variables | 25416 |
| Total Variables | 25416 |
| Sound Constraints | True |

## General Analysis

| **Category** | **Evaluation** |
|---|---|
| Access Control | **Strong**. Access is limited as intended, mainly to group admins. |
| Launch Risk | **Strong**. Semaphore does not manage assets. Additionally many dApps built on Semaphore will deploy their own Semaphore contract. They should consider launch controls if necessary. |
| Code Quality | **Strong**. Code follows best practices for solidity and circom. No unnecessary use of assembly. No confusing variable/function names. Good use of interfaces and inheritance. |
| Decentralization | **Weak.** Group admins have full control over adding/removing members and casting votes. They can censor any vote in the case of SemaphoreVoting. |
| Events | **Strong**. Events are emitted after every important function call such as casting votes and adding members. Relevant details are emitted. |

| | |
|---|---|
| Dummy Proof | **Moderate**. Contract function names are clear in their intent and hard to misuse. However, some 0 address checks are missing. |
| Complexity | **Strong**. Short and simple contracts and circuits. |
| Testing | **Moderate**. Strong and well written unit tests. However the circuits are not well tested. This is due to a lack of Circom testing tools at the moment. |
| Documentation | **Strong**. NatSpec comments for all functions and good documentation on the website. |
| Cryptography | **Not Thoroughly Reviewed** |
| ZK Circuits | **Strong**. Circuits are simple and weakly verified. |

# Findings

## Major

### [M1] Missing ZK snark scalar field check on 0 leaf in dependency

**Location**
https://github.com/privacy-scaling-explorations/zk-kit/blob/cc21bb125fb594772e7e46111fdba05845a06355/packages/incremental-merkle-tree.sol/contracts/IncrementalBinaryTree.sol#L29

**Description**
During initialization of the IncrementalBinaryTree.sol, the user can enter any value for the uint256 zero field. This becomes an issue if the user uses a value greater than the SNARK_SCALAR_FIELD. A zero leaf can be inside an array of proofSiblings when proving existence of a leaf which may cause an issue when the IncrementalBinaryTree.verify function is called during the IncrementalBinaryTree.remove function. The verify function requires that all proofSiblings are less than the SNARK_SCALAR_FIELD. So, if the 'zero' leaf is greater than the SNARK_SCALAR_FIELD, this verify function will unintentionally fail.

**Suggested Solution**
Add a require(zero < SNARK_SCALAR_FIELD, "...") statement in the IncrementalBinaryTree.init() function.

### [M2] Missing checks to ensure zk proof inputs are less than SNARK_SCALAR_FIELD

**Location**

https://github.com/semaphore-protocol/semaphore/blob/5186a940ff495ff163bd5779631a716d0bf96507/contracts/base/SemaphoreGroups.sol#L27

**Description**
The SemaphoreGroup.sol contract uses the "groupId" as the external nullifier. Therefore, the groupId will be provided as a public input when calling Verifier.verifyProof. When a user creates a new poll via SemaphoreVoting.createPoll, the user can enter any uint256 value as the poll id, which will then be the group Id. The issue arises when the user inputs a value for pollId that is greater than the SNARK_SCALAR_FIELD constant. Then, the call to verify any proofs for this poll will fail because the Verifier checks that *input[i]* < *snark_scalar_field* for each public input. Since both the SemaphoreVoting.sol and SemaphoreWhistleblowing.sol contracts use the groupId as external nullifiers, this issue is present in both.

**Suggested Solution**
Add a constraint in the SemaphoreGroup.sol contract to check that the groupId is less than the SNARK_SCALAR_FIELD. This will protect any extensions that use the group id as an external nullifier.

# Warnings

## [W1] Root history can be overridden by a different group

**Location**
https://github.com/semaphore-protocol/semaphore/blob/fd3cc6f7db46aec83c0f807a8155dcd66561a0db/contracts/base/SemaphoreGroups.sol#L21

**Description**
The root history field variable in the SemaphoreGroups.sol contract is meant to store a mapping of every root to its corresponding group id. Since it is possible that the same users (with the same identity commitments) are added as members in the same order, the root history for the first group will be overridden. Instead all of those roots will map to the new group. This can be an issue for any applications that assume the root history for a group is unique and never overridden.

**Suggested Solution**
Allow dApps to implement this feature themselves if needed and remove from contract.

## [W2] Users must trust the verification key stored in the Verifier contracts

**Location**
N/A

**Description**

The verification key stored in the Verification contract can store a key that is meant for a much less strict circuit. All it takes is one person to catch the incorrect verification keys, but that requires good background knowledge on how to do that. Perhaps an easy to use script that verifies that the verification key stored in the Verifier contracts matches the circuits will help users to fully trust the application.

**Suggested Solution**
Add a simple script that any user can quickly run to verify that the verifying keys match the circuits.

### [W3] Missing address 0 check

**Location**
https://github.com/semaphore-protocol/semaphore/blob/ee7aad1dd72c5b473c0012ca0f9c9d0ce5710125/contracts/Semaphore.sol#L40,
https://github.com/semaphore-protocol/semaphore/blob/ee7aad1dd72c5b473c0012ca0f9c9d0ce5710125/contracts/Semaphore.sol#L54,
https://github.com/semaphore-protocol/semaphore/blob/ee7aad1dd72c5b473c0012ca0f9c9d0ce5710125/contracts/extensions/SemaphoreVoting.sol#L42,
https://github.com/semaphore-protocol/semaphore/blob/ee7aad1dd72c5b473c0012ca0f9c9d0ce5710125/contracts/extensions/SemaphoreWhistleblowing.sol#L44

**Description**
When creating a group or updating the admin, users can input the 0 address to be the admin. This will lock the group and render it useless, wasting gas for the user. Inputting the 0 address can happen when a user doesn't initialize an address variable and the default value is used. This is a dummy proof warning.

**Suggested Solution**
Add zero address checks at the beginning of the listed functions.

## Fix Log

| Issue | Severity | Status |
|---|---|---|
| [M1] | Major | Fixed. https://github.com/privacy-scaling-explorations/zk-kit/issues/23 |
| [M2] | Major | Fixed. https://github.com/semaphore-protocol/semaphore/issues/90 |
| [W1] | Warning | Fixed. https://github.com/semaphore-protocol/semaphore/commit/c6667d98b8c72a424afd118da6cbde03a4945690 |

| | | |
|---|---|---|
| [W2] | Warning | Not handled. This is a common issue for a lot of zk projects and so this issue may be fixed for many projects at once with a single tool that allows users to quickly verify a verifying key. |
| [W3] | Warning | Not handled. |

## Vulnerability Classifications

| **Severity Categories** | |
|---|---|
| **Severity** | **Description** |
| Recommendation | Information not relevant to security, but may be helpful for efficiency, costs, etc. |
| Warning | The issue does not pose an immediate security threat, but may be a lack of following best practices or more easily lead to the future introductions of bugs. |
| Minor | The code does not work as intended. Impact to the system and users is minimal if present at all. |
| Major | The issue can lead to **moderate** financial, reputation, availability, or privacy damage. Or the issue can lead to substantial damage under extreme and unlikely circumstances. |
| Critical | The issue can lead to **substantial** financial, reputation, availability, or privacy damage. |