hexens × Ava Labs.

JAN.25

SECURITY REVIEW REPORT FOR AVA LABS

CONTENTS

- Public Key Registration Proof Replay in the Registrar contract
- Missing Event Emission for Auditor PCT Values in Withdraw Function
- Redundant parameter in withdraw function
- Single-step ownership change introduces risks
- Potential DoS Attack on ZK Prover Infrastructure via Negative Scalar Multiplication
- Wrong curve order usage in grindKey function

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

SCOPE

The analyzed resources are located on:

https://github.com/ava-labs/EncryptedERC
commit: bbfc7a1fa4706a3bfd4d895c184ece5dcd0a7e60
https://github.com/ava-labs/ac-eerc-sdk/src/crypto/key.ts
commit: dcb90c689dda4e83f3efb63751271ea5d4fdfc13

The issues described in this report were fixed. Corresponding commits are mentioned in the description.

AUDITING DETAILS

STARTED
20.01.2025

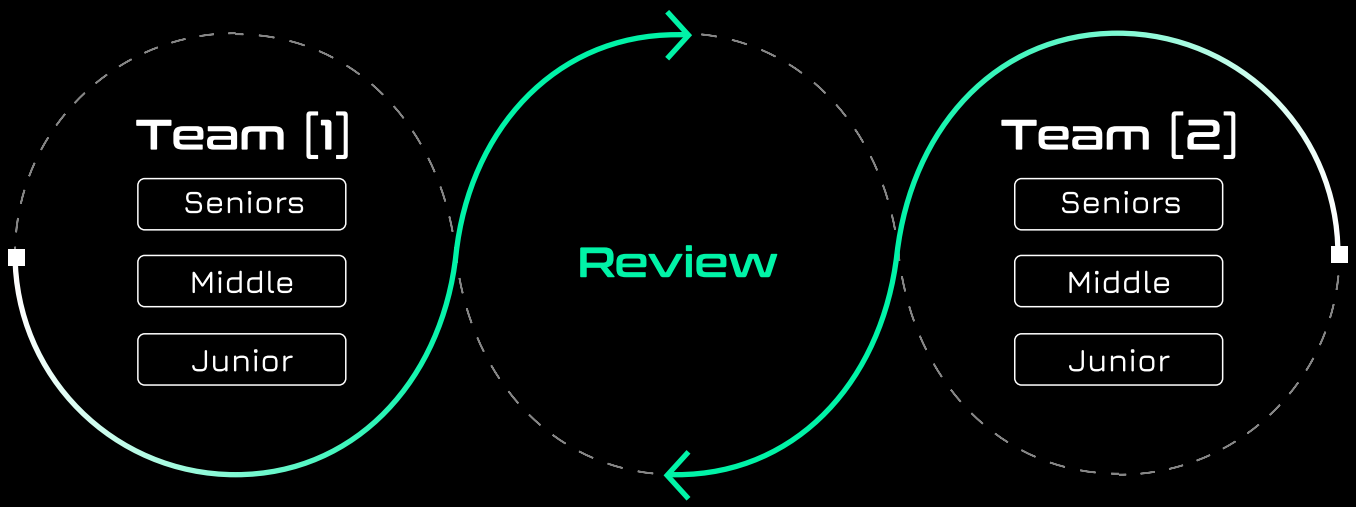
DELIVERED
18.02.2025

Review Led by

HAYK ANDRIASYAN
Senior Security Researcher | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

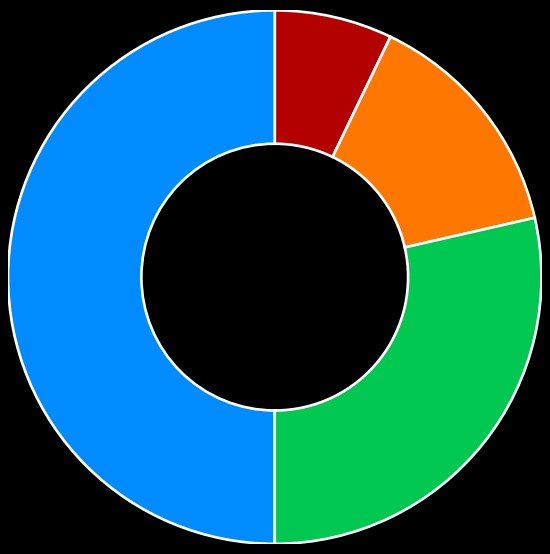
ISSUE SYMBOLIC CODES

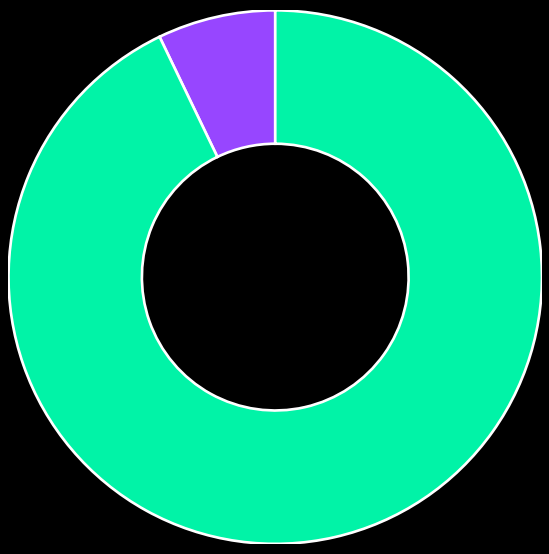
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	0
Medium	2
Low	4
Informational	7

Total: 14





WEAKNESSES

This section contains the list of discovered weaknesses.

ETRP-3

MISSING BABYJUBJUB BASE POINT SUBGROUP ORDER CHECK CONSTRAINTS ON SCALAR NUMBERS

SEVERITY: Critical

REMEDIATION:

Add a constraint in the circuits that checks the scalar values which are being multiplied with the base point are lower than the Base Point subgroup order.

STATUS: Fixed

DESCRIPTION:

The application leverages Gnark circuits to validate the correctness of operations such as Mint, Transfer, Withdraw, and Registration. These circuits are compiled using the BN254 elliptic curve.

```
witness.New(ecc.BN254.ScalarField())
```

The scalar field represents the set of scalars used in operations performed on the curve, including point addition, scalar multiplication, and pairing. For the BN254 curve, the scalar field is equals to **21888242871839275222246405745257275088548364400416034343698204186575808495617**, meaning scalar elements within **frontend.Variable** operate in the range **[0, ecc.BN254.ScalarField() - 1]**.

The application implements ERC20 methods using partially homomorphic encryption based on the Elliptic Curve ElGamal algorithm. This algorithm leverages the BabyJubJub Twisted Edwards curve, defined by the equation **ax^2 + y^2 = 1 + dx^2y^2** with parameters:

```
a = 168700
d = 168696

prime_finite_field =
21888242871839275222246405745257275088548364400416034343698204186575808495617
curve_order =
21888242871839275222246405745257275088614511777268538073601725287587578984328
base_point_order =
2736030358979909402780800718157159386076813972158567259200215660948447373041

base_point_x =
5299619240641551281634865583518297030282874472190772894086521144482721001553
base_point_y =
16950150798460657717958625567821834550301663161624707787222815936182638968203
```

In the ElGamal algorithm, both the public key generation and encryption process rely on scalar multiplication, where a scalar (e.g., the private key, plaintext message, etc.) is multiplied by the Base Point of the elliptic curve. The Base Point generates a subgroup with an order defined as **base_point_order**, which implies:

```
scalar_number * BasePoint == (scalar_number + base_point_order) * BasePoint
```

As they are equal, the El Gamal encryption also will produce the same points.

Base Point order in BabyJubJub curve is lower than the BN254 scalar field (**base_point_order < ecc.BN254.ScalarField()**), so the scalar numbers (**frontend.Variable** type) defined in circuits can have their aliases when they are multiplied on the Base Point. This can cause to proving to have a balance much more than actually is or transfer more amount of tokens and withdraw more amount than the user have.

```
/*
CheckBalance checks if the sender's balance is a well-formed ElGamal
ciphertext by decryption
*/
func CheckBalance(api frontend.API, bj *babyjub.BjWrapper, sender Sender) {
 decSenderBalanceP :=
bj.ElGamalDecrypt([2]frontend.Variable{sender.BalanceEGCT.C1.X,
sender.BalanceEGCT.C1.Y}, [2]frontend.Variable{sender.BalanceEGCT.C2.X,
sender.BalanceEGCT.C2.Y}, sender.PrivateKey)
 givenSenderBalanceP := bj.MulWithBasePoint(sender.Balance)
 bj.AssertPoint(givenSenderBalanceP, decSenderBalanceP.X,
decSenderBalanceP.Y)
}
```

bj.MulWithBasePoint(sender.Balance) will produce the same curve point if the balance will be **sender.Balance + BasePointOrder** in a case of **sender.Balance + BasePointOrder < BN254ScalarField**. So the **CheckBalance** can be bypassed.

When the **sender.Balance + BasePointOrder** is encrypted with the same random and the public key as the **sender.Balance** one, El Gamal encryption's C1, C2 values will remain the same for the **sender.Balance + BasePointOrder**. So **_isBalanceValid** method also will provide a valid value for the **providedBalance** as the **balanceHash** is the same for **sender.Balance** and **sender.Balance + BasePointOrder**

```
/**
 * @param _eGCT Elgamal Ciphertext
 * @return hash of the Elgamal Ciphertext CRH(eGCT)
 */
function _hashEGCT(EGCT memory _eGCT) internal pure returns (uint256) {
    return
        uint256(
            keccak256(
                abi.encode(_eGCT.c1.X, _eGCT.c1.Y, _eGCT.c2.X, _eGCT.c2.Y)
            )
        );
}
```

ETRP-1

UNSAFE USAGE OF TRANSFER/ TRANSFERFROM IN ENCRYPTEDERC DEPOSIT/WITHDRAW METHODS

SEVERITY: Medium

REMEDIATION:

The EncryptedERC contract should utilize OpenZeppelin's SafeERC20 library, specifically the safeTransfer() and safeTransferFrom() methods, in its deposit and withdraw functions instead of relying on transfer() and transferFrom().

STATUS: Fixed

DESCRIPTION:

The **EncryptedERC** token enables private transfers by utilizing partially homomorphic encryption, ensuring confidentiality during transactions. Moreover, it provides an opportunity for other ERC-20 tokens to be transferred securely in an encrypted manner. To achieve this, users deposit the desired amount of the ERC-20 token into the **EncryptedERC** contract:

```
function deposit(
    uint256 _amount,
    address _tokenAddress,
    uint256[7] memory _amountPCT
) public {
    ...
    IERC20 token = IERC20(_tokenAddress);
    uint256 dust;
    uint256 tokenId;
    address to = msg.sender;
    ...
```

```
    // this function reverts if the transfer fails
    token.transferFrom(to, address(this), _amount);

    (dust, tokenId) = _convertFrom(to, _amount, _tokenAddress, _amountPCT);

    // transfer the dust back to the user
    token.transfer(to, dust);

    emit Deposit(to, _amount, dust, tokenId);
}
```

EncryptedERC contract's **deposit** function assumes that ERC20 token's **transferFrom** reverts in case of a call failure. Some tokens do not implement the ERC20 standard properly. Not all ERC20 implementations **revert()** when there's a failure in **transfer()** or **transferFrom()**. In such cases, if the **transferFrom** call fails, the **deposit** function execution will continue, resulting in the amount being deposited into the **EncryptedERC** contract despite the failed transfer.

```
function _convertFrom(
    address _to,
    uint256 _amount,
    address _tokenAddress,
    uint256[7] memory _amountPCT
) internal returns (uint256 dust, uint256 tokenId) {
    uint8 tokenDecimals = IERC20Metadata(_tokenAddress).decimals();

    uint256 value = _amount;
    dust = 0;
    ...
    {
        uint256[2] memory publicKey = registrar.getUserPublicKey(_to);

        EGCT memory _eGCT = BabyJubJub.encrypt(
            Point({X: publicKey[0], Y: publicKey[1]}),
            value
        );

        EncryptedBalance storage balance = balances[_to][tokenId];

        if (balance.eGCT.c1.X == 0 && balance.eGCT.c1.Y == 0) {
            balance.eGCT = _eGCT;
        } else {
            balance.eGCT.c1 = BabyJubJub._add(balance.eGCT.c1, _eGCT.c1);
            balance.eGCT.c2 = BabyJubJub._add(balance.eGCT.c2, _eGCT.c2);
        }

        balance.amountPCTs.push(
            AmountPCT({pct: _amountPCT, index: balance.transactionIndex})
        );
        balance.transactionIndex++;

        _commitUserBalance(_to, tokenId);
    }

    return (dust, tokenId);
}
```

ETRP-5

DISCREPANCY BETWEEN CLEAR TEXT AND AMOUNT PCT WHEN DEPOSITING FEE ON TRANSFER TOKENS

SEVERITY: Medium

PATH:

contracts/EncryptedERC.sol#L472-L506

REMEDIATION:

We have two recommendations:

1. Either implement a blacklisting mechanism and avoid supporting such weird tokens
2. Or encrypt (both El Gamal and Poseidon) the exact amount that is transferred to the contract, which can be determined using the balance difference before and after the transfer

STATUS: Fixed

DESCRIPTION:

The **EncryptedERC.sol** contract will not work properly with a fee on transfer tokens, as when depositing such one via the **deposit()** function it will transfer to the contract the **_amount**:

```
        // this function reverts if the transfer fails
        token.transferFrom(to, address(this), _amount);
```

but the balance of the contract will be actually increased by the **_amount-fee**.

The **deposit()** function gets the Poseidon Encryption of the amount value (**_amountPCT**) in addition to the **_amount** parameter, and that **_amountPCT** is saved in the contract.

The issue is that the wrong amount is being encrypted, i.e. the **_amountPCT** is not the encryption of the actual deposit amount (**_amount-fee**).

The contract, on its turn, is doing El Gamal encryption of the deposit amount given by the front-end(**_amount**) via the **_convertFrom()** function (L668-L671):

```
EGCT memory _eGCT = BabyJubJub.encrypt(
    Point({X: publicKey[0], Y: publicKey[1]}),
    value
);
```

So now, when a user wants to withdraw their funds, the contract assumes the full **_amount** is available to the user. However, only **_amount-fee** was initially deposited. This discrepancy causes the contract to withdraw additional tokens from the pooled balance, thereby utilizing other users' funds to fulfill the withdrawal request.

```
    function deposit(
        uint256 _amount,
        address _tokenAddress,
        uint256[7] memory _amountPCT
    ) public {
        // revert if auditor key is not set
        if (!isAuditorKeySet()) {
            revert AuditorKeyNotSet();
        }

        // revert if contract is not a converter
        if (!isConverter) {
            revert InvalidOperation();
        }

        IERC20 token = IERC20(_tokenAddress);
        uint256 dust;
        uint256 tokenId;
        address to = msg.sender;

        // revert if the user is not registered to registrar contract
        if (!registrar.isUserRegistered(to)) {
            revert UserNotRegistered();
        }

        // this function reverts if the transfer fails
        token.transferFrom(to, address(this), _amount);

        (dust, tokenId) = _convertFrom(to, _amount, _tokenAddress,
_amountPCT);

        // transfer the dust back to the user
        token.transfer(to, dust);

        emit Deposit(to, _amount, dust, tokenId);
    }
```

ETRP-12

PROOF REUSE AND MALLEABILITY IN PRIVATEMINT FUNCTION BYPASSES AUDITOR CONTROLS IN PRIVATEMINT

SEVERITY: Low

PATH:

ava-labs/EncryptedERC/contracts/EncryptedERC.sol

REMEDIATION:

Implement nullifier tracking by hashing keccak256(abi.encodePacked(chainId, auditorPCT)) and storing it in a mapping to ensure each proof can only be used once per chain. This prevents both same-chain proof reuse and cross-chain replay attacks.

STATUS: Fixed

DESCRIPTION:

The **privateMint** function in the EncryptedERC contract, despite being restricted to the owner, allows the reuse of zero-knowledge proofs which affects the intended auditor control mechanism. While the mint circuit (**mint_circuit.go**) verifies the auditor's encryption and approval of the mint amount, the current implementation does not prevent the same proof from being used multiple times.

This implementation allows the owner to:

- Mint multiple times using a single auditor approval
- Exceed the auditor-approved mint amount
- Operate beyond the scope of auditor oversight

The issue is compounded by:

- Proof malleability, where manipulating the y-coordinates of the proof points can create multiple valid variations of the same proof
- Cross-chain vulnerability, as the proof verification doesn't include chain-specific parameters (like chainId). This allows an approved proof from one network to be replayed on other networks where the contract is deployed, effectively multiplying the impact of a single auditor approval across multiple chains

```
    function privateMint(
        address _user,
        uint256[8] calldata proof,
        uint256[22] calldata input
    ) external onlyOwner {
        if (isConverter) {
            revert InvalidOperation();
        }

        if (!isAuditorKeySet()) {
            revert AuditorKeyNotSet();
        }

        if (!registrar.isUserRegistered(_user)) {
            revert UserNotRegistered();
        }

        {
            // user public key should match
            uint256[2] memory userPublicKey =
registrar.getUserPublicKey(_user);
            if (userPublicKey[0] != input[0] || userPublicKey[1] !=
input[1]) {
                revert InvalidProof();
            }
        }

        {
            // auditor public key should match
            if (
```

```
                auditorPublicKey.X != input[13] ||
                auditorPublicKey.Y != input[14]
            ) {
                revert InvalidProof();
            }
        }

        mintVerifier.verifyProof(proof, input);
        _privateMint(_user, input);
    }
```

Proof of concept:

```
  it.only("", async () => {
            const {
                registrar: registrar_1,
                users: users_1,
                signers: signers_1,
                owner: owner_1,
                encryptedERC: encryptedERC_1,
                erc20s: erc20s_1
            } = await deployFixture(false);

   await call_register(users_1[0], registrar_1);
            await call_register(users_1[1], registrar_1);
            await call_register(users_1[2], registrar_1);
            await call_setAuditorPublicKey(users_1[2], owner_1,
encryptedERC_1);
            // user balance : 0n
            console.log("user balance 1", await getUserBalance(users_1[0],
erc20s_1[0], encryptedERC_1));

            // +100n
            let {proof: mintProof1, publicInputs: mintPublicInputs1} = await
call_privateMint(users_1[0], owner_1, 100n*(10n**6n), encryptedERC_1);

            // +100n
            await encryptedERC_1
                .connect(owner_1)
                .privateMint(users_1[0].signer.address, mintProof1,
mintPublicInputs1);

            const P =
BigInt("21888242871839275222246405745257275088696311157297823662689037894645
226208583");

            mintProof1[1] = (P - (BigInt(mintProof1[1]) % P)).toString();
            mintProof1[4] = (P - (BigInt(mintProof1[4]) % P)).toString();
            mintProof1[5] = (P - (BigInt(mintProof1[5]) % P)).toString();
```

```
            // +100n
            await encryptedERC_1
                .connect(owner_1)
                .privateMint(users_1[0].signer.address, mintProof1,
mintPublicInputs1);

            // user balance : 300n
            console.log("user balance 2", await getUserBalance(users_1[0],
erc20s_1[0], encryptedERC_1));
        })
```

ETRP-6

MISSING CONSTRAINTS ON MESSAGE PADDING IN POSEIDON DECRYPTION CIRCUIT

SEVERITY: Low

PATH:

zk/pkg/poseidon/poseidon_decryption.go#L6-L56

REMEDIATION:

Ensure padded bytes are constrained to zero, as recommended in the Poseidon Encryption paper.

STATUS: Fixed

DESCRIPTION:

The application implements Poseidon encryption to secure balance and transfer amount data. The encryption algorithm is detailed in the following paper: Encryption with Poseidon by Dmitry Khovratovich (https://drive.google.com/file/d/1EVrP3DzoGbmzkRmYnyEDcIQcXVU7GlOd/view).

During the decryption process, one of the verification steps is as follows:

“iv. If 3 does not divide l, verify that the last 3 − (l mod 3) elements of M are zero. If this condition is not met, the ciphertext must be rejected.”

However, the decryption circuit lacks constraints to enforce this condition. Specifically, when the message length (l) is not a multiple of 3, the required zero-padding is not adequately validated.

ETRP-7

IMPROPER VALIDATION OF MESSAGE PADDING DURING POSEIDON DECRYPTION IN EERC SDK

SEVERITY: Low

PATH:

src/crypto/poseidon/poseidon.ts#L199

REMEDIATION:

Change the condition from if (length > 3) to if (length % 3 > 0).

STATUS: Fixed

DESCRIPTION:

poseidonDecrypt function implements decryption of the transfer amounts and balances via Poseidon decryption algorithm described in the Encryption with Poseidon paper.

The current implementation validates message padding by checking if the message length exceeds 3 and evaluating **length % 3**.

```
if (length > 3) {
  if (length % 3 === 2) {
    checkEqual(
      msg[msg.length - 1],
      this.field.zero,
      this.field,
      "The last element of the message must be 0",
    );
  } else if (length % 3 === 1) {
    checkEqual(
      msg[msg.length - 1],
      this.field.zero,
      this.field,
      "The last element of the message must be 0",
    );
    checkEqual(
      msg[msg.length - 2],
      this.field.zero,
      this.field,
      "The second to last element of the message must be 0",
    );
  }
}
```

However, this approach misses cases where the message length is 1 or 2. In these scenarios, the condition **length % 3 != 0** still holds true, indicating the presence of padding, but the check fails due to the **length > 3** condition. As a result, padding in messages of length 1 or 2 remains unchecked.

ETRP-8

NON-UNIFORM DISTRIBUTION IN RANDOM NONCE GENERATION DUE TO JAVASCRIPT NUMBER PRECISION LIMITATIONS

SEVERITY: Low

REMEDIATION:

Replace the current implementation with built-in cryptographic random number generator
Crypto | Node.js v23.7.0 Documentation

STATUS: Fixed

DESCRIPTION:

The **randomNonce()** function in **jub/jub.ts** uses **Math.random()** and **Number()** type conversion to generate random nonces, which can lead to non-uniform distribution of values due to JavaScript's number precision limitations.

This implementation has two issues:

- **Math.random()** is not cryptographically secure
- Converting large BigInt values to Number type leads to precision loss beyond **Number.MAX_SAFE_INTEGER** (2^53 - 1)

```
console.log("Number.MAX_SAFE_INTEGER:", Number.MAX_SAFE_INTEGER); //
9007199254740991
console.log("2n ** 53n:", 2n ** 53n);                                //
9007199254740992
console.log(Number(2n ** 53n) + 1);                                  //
9007199254740992
console.log(Number(2n ** 53n) + 2);                                  //
9007199254740994
console.log(Number(2n ** 53n) + 3);                                  //
9007199254740996
console.log(Number(2n ** 53n) + 4);                                  //
9007199254740996
```

As shown above, numbers beyond **MAX_SAFE_INTEGER** lose precision, causing some values to be unreachable and others to be over-represented in the random distribution.

```
export const randomNonce = (): bigint =>
    BigInt(Math.floor(Math.random() * Number(2n ** 128n - 1n)) + 1);
```

ETRP-2

WRONG BABYJUBJUB CURVE ORDER INITIALIZATION

SEVERITY: Informational

PATH:

zk/pkg/babyjub/babyjub.go#L25

REMEDIATION:

Change the order from the prime to 21888242871839275222246405745257275088614511772685380736017252875 87578984328.

STATUS: Fixed

DESCRIPTION:

The circuits leverage the BabyJubJub Twisted Edwards curve for ElGamal encryption. The curve's order refers to the number of elliptic curve points defined over a finite field. For the BabyJubJub curve it is initialized as the finite field prime number which is not the number of curve points:

```
 order, _ :=
big.NewInt(0).SetString("218882428718392752222464057452572750885483644004160
343436982041865758084956l7", 10)
```

As defined in the ERC-2494: Baby Jubjub Elliptic Curve , the order equals to 21888242871839275222246405745257275088614511772685380736017252875875789843 28.

ETRP-4

PUBLIC KEY REGISTRATION PROOF REPLAY IN THE REGISTRAR CONTRACT

SEVERITY: Informational

PATH:

contracts/Registrar.sol#L43

REMEDIATION:

Add sender's wallet address in the proof inputs and use the address from the proof inputs instead of reading from the msg.sender value.

STATUS: Fixed

DESCRIPTION:

The Registrar contract is responsible for keeping the public keys of users. A user can register its public key via **register** method. The user must provide a proof for the public key ownership.

```
function register(
    uint256[8] calldata proof,
    uint256[2] calldata input
) external {
    address account = msg.sender;

    registrationVerifier.verifyProof(proof, input);

    require(!isUserRegistered(account), "UserAlreadyRegistered");

    _register(account, Point({X: input[0], Y: input[1]}));
}
```

As the proof doesn't include the sender's address itself, anyone can register other user's public key via replying that user's proof.

ETRP-9

MISSING EVENT EMISSION FOR AUDITOR PCT VALUES IN WITHDRAW FUNCTION

SEVERITY: Informational

REMEDIATION:

Modify the Withdraw event definition to include the auditorPCT array and emit these values along with the existing parameters

STATUS: Fixed

DESCRIPTION:

In the **EncryptedERC.sol** contract, the Withdraw event emission does not include the **auditorPCT** values that are collected from the input array. While the code processes these values by storing them in a memory array.

```
uint256[7] memory auditorPCT;
for (uint256 i = 0; i < 7; i++) {
    auditorPCT[i] = input[8 + i];
}

emit Withdraw(from, _amount, _tokenId);
```

ETRP-10

REDUNDANT PARAMETER IN WITHDRAW FUNCTION

SEVERITY: Informational

PATH:

contracts/EncryptedERC.sol#L541-L543

REMEDIATION:

Remove the _amount parameter and the if condition, and take the amount value from the proof, also fix in the EERC SDK's withdraw call.

STATUS: Fixed

DESCRIPTION:

The **withdraw** function in EncryptedERC contract accepts **_amount**, **_tokenId**, **proof**, **input**, **_balancePCT** parameters.

Inside the function there is a check:

```
{
    // _amount should match with the amount in the proof
    if (_amount != input[15]) {
        revert InvalidProof();
    }
}
```

As the **_amount** and **input[15]** parameters are given by the user, the check is redundant, as the user can give only one of them.

ETRP-11

SINGLE-STEP OWNERSHIP CHANGE INTRODUCES RISKS

SEVERITY: Informational

PATH:

contracts/EncryptedERC.sol#L26

REMEDIATION:

Use OpenZeppelin's Ownable2Step.sol.

STATUS: Fixed

DESCRIPTION:

The **EncryptedERC.sol** contract imports OZ's **Ownable.sol**, as the contract is non-upgradeable and have an **onlyOwner** functionalities, it is especially important that transfers of ownership should be handled with care.

```
function privateMint(
    address _user,
    uint256[8] calldata proof,
    uint256[22] calldata input
) external onlyOwner {
...
}

function setAuditorPublicKey(address _user) external onlyOwner {
...
}
```

ETRP-13

POTENTIAL DOS ATTACK ON ZK PROVER INFRASTRUCTURE VIA NEGATIVE SCALAR MULTIPLICATION

SEVERITY: Informational

PATH:

main/packages/baby-jubjub/src/baby-jubjub.ts#L72-L87

REMEDIATION:

1. Implement validation to ensure scalar values are within the valid field range before performing point multiplication
2. Modify the shiftRight implementation to use unsigned right shift behavior

STATUS: Acknowledged

DESCRIPTION:

An infinite loop issue has been discovered in the mulPointEscalar function. This occurs when the function is called with negative integer scalar values. The shiftRight operation used in scalar arithmetic preserves the sign of negative numbers, preventing the loop termination condition isZero(rem) from ever being satisfied.

This issue could be exploited to perform a Denial of Service (DoS) attack on the ZK Prover infrastructure by intentionally passing negative values, causing the system to hang indefinitely.

```
//ava-labs/ac-eerc-sdk/src/crypto/scalar.ts

  shiftRight(s: bigint, n: number): bigint {
    return s >> BigInt(n);
  },
```

```
export function mulPointEscalar(base: Point<bigint>, e: bigint):
Point<bigint> {
    let res: Point<bigint> = [Fr.e(BigInt(0)), Fr.e(BigInt(1))]
    let rem: bigint = e
    let exp: Point<bigint> = base

    while (!scalar.isZero(rem)) {
        if (scalar.isOdd(rem)) {
            res = addPoint(res, exp)
        }

        exp = addPoint(exp, exp)
        rem = scalar.shiftRight(rem, BigInt(1))
    }

    return res
}
```

ETRP-14

WRONG CURVE ORDER USAGE IN GRINDKEY FUNCTION

SEVERITY: Informational

PATH:

src/crypto/key.ts#L46

REMEDIATION:

Change SNARK_FIELD_SIZE to BabyJubJub curve order value. Also define an iteration count upper limit for the while loop to avoid from a DoS.

STATUS: Fixed

DESCRIPTION:

The **grindKey** function used during private key generation to protect from modulo bias. Upper limit of the range is selected BN254 scalar field size, but as it's used for BabyJubJub key generation, the limit must be the BabyJubJub curve order.

```
export const grindKey = (seed: string): string => {
  const limit = SNARK_FIELD_SIZE;
  const maxAllowedValue = SHA_256_MAX_DIGEST - (SHA_256_MAX_DIGEST % limit);

  let i = 0;
  let key = hashKeyWithIndex(seed, i);
  i++;

  // make sure that key is in the max allowed range
  while (key >= maxAllowedValue) {
    key = hashKeyWithIndex(seed, i);
    i++;
  }

  return (key % limit).toString(16);
};
```


hexens
×
Ava
Labs.