ZKSECURITY

Audit of Self EU ID cards Circuits, OFAC Checks, and Smart Contracts

Date: June 17th, 2025

Introduction

On June 17th, 2025, zkSecurity was engaged to perform an audit of the Self project. The assessment lasted for one week and focused on the implemented changes to support european ID MRTD cards, the implementation of OFAC checks and the changes to the smart contracts. The audit was conducted on the self repository at commit `4f18c75` and was carried out by two consultants.

Scope

The audit focused on the following subset of the Self project:

1. The circuits that support European ID MRTD cards, which includes the registration of documents and the disclosure of information. In particular the emphasis was on the following circuits:
    - `circuits/register/register_id.circom`
    - `circuits/disclose/vc_and_disclose_id.circom`
    - `circuits/utils/passport/disclose/disclose_id.circom`
2. The implementation of OFAC checks for both IDs and passports:
    - All circuits in `circuits/utils/passport/ofac/`
3. The proof of non-inclusion using sparse Merkle trees (SMT) used by the OFAC checks:
    - `circuits/utils/crypto/merkle-trees/smt.circom`
4. Changes to the smart contracts, in particular the smart contracts belonging to the V2.
    - `contracts/contracts/IdentityVerificationHubImplV2.sol`
    - `contracts/contracts/libraries/GenericFormatter.sol`
    - `contracts/contracts/libraries/CustomVerifier.sol`
    - `contracts/contracts/libraries/CircuitAttributeHandlerV2.sol`
    - `contracts/contracts/abstract/SelfVerificationRoot.sol`
    - `contracts/contracts/constants/CircuitConstantsV2.sol`
    - `contracts/contracts/constants/AttestationId.sol`
    - `contracts/contracts/interfaces/IIdentityVerificationHubV2.sol`
    - `contracts/contracts/interfaces/ISelfVerificationRoot.sol`

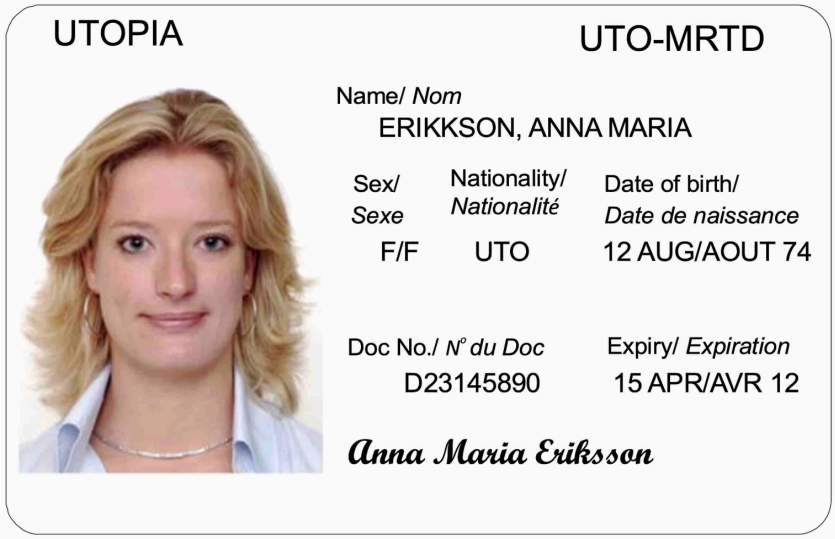
- `contracts/contracts/registry/IdentityRegistryIdCardImplV1.sol`

Note: zkSecurity recently completed a broad review of the Self OpenPassport monorepo, so this engagement concentrated solely on the newly added functionality outlined above. In our prior assessment, we recommended a more exhaustive examination of the entire codebase (given the time constraints and complexity of certain findings) and we are now working with the client to schedule further evaluations.

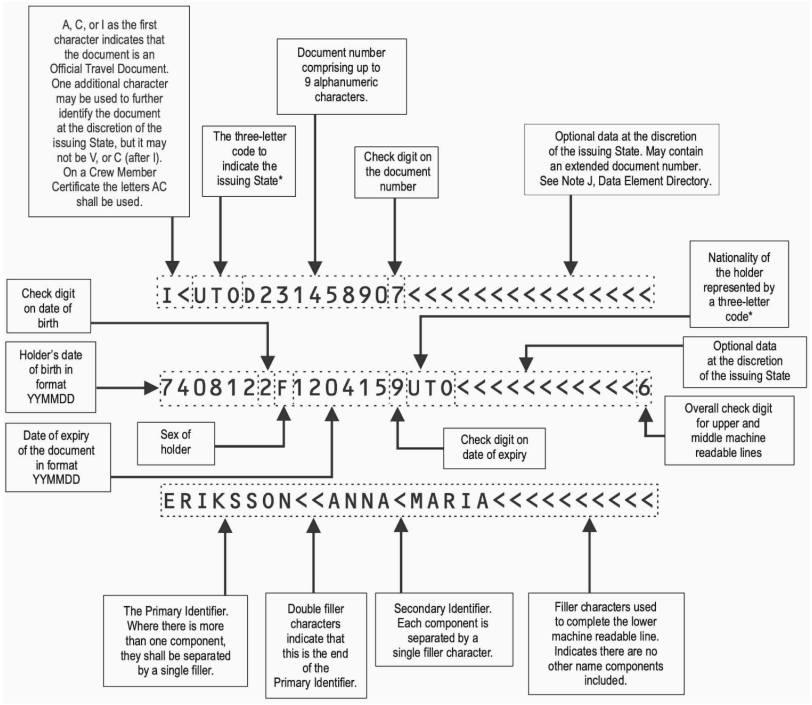
Overview of the scope

Overview of European ID MRTD cards

The front of a European ID Machine Readable Travel Document (MRTD) card typically adheres to the following design (from "Specifications for TD1 Size Machine Readable Official Travel Documents (MROTDs)"):



The back of the card contains the Machine Readable Zone (MRZ) which holds all the data displayed on the front of the card as well as optional data that the issuer can add. See below for a detailed breakdown of what the structure of this data looks like (Appendix from the specification):



This section is referred to as the DG1 field and contains all the information that users can selectively reveal in the circuit.

Changes in the circuits to support European ID MRTD cards

In order to support EU ID cards in addition to passports, the Self project has refactored the circuits to support the new ID format. In Self, the process of handling ID information is split into three steps:

1. **Registration of a DSC**: a user can provide a proof that a DSC certificate is valid and can store the certificate information on-chain.
2. **Registration**: a user can prove that they own an ID signed by a valid DSC and can store a hiding commitment to the ID on-chain.
3. **Disclosure**: a user can prove that they own an ID stored on-chain, and that the ID satisfies certain properties, such as its holder's age being greater than some amount, and not being associated with a sanctioned entity according to the OFAC list.

EU ID cards share most of their logic with Passports, but there are minor differences:

- The DG1 field has a different structure and length: 93 bytes for passports and 95 bytes for EU IDs.
- The OFAC checks are different: both check that the hash values of `(name, date of birth)` and `(name, year of birth)` are not in the OFAC list, but the passport can additionally check that the hash value of `(passport number, nationality)` is not in the OFAC list.

- The attestation ID is different: 1 for passports and 2 for EU IDs. This is done to distinguish between the two different types of IDs, since the commitments are stored in the same Merkle tree.

To accommodate these differences, the Self project has created the following new circuits for EU IDs:

- `circuits/register_id/register_id.circom`
- `circuits/disclose/vc_and_disclose_id.circom`
- `circuits/utils/passport/disclose/disclose_id.circom`
- `circuits/utils/passport/ofac/ofac_name_dob_id.circom`
- `circuits/utils/passport/ofac/ofac_name_yob_id.circom`

Note that there is no separate circuit for the `dsc.circom` circuit, which is used to verify the DSC certificate, as there is no difference between passports and EU IDs in this case.

Circuits implementing OFAC checks

All the OFAC checks are implemented using a sparse Merkle tree (SMT) data structure, for which we provide a brief overview below. The SMT is used as a key-value store, and supports proving both inclusion and non-inclusion of a given key in the tree. In the case of OFAC checks, the values are always set to `1`, since the only information stored is whether a key is present in the tree or not.

The trees are computed off-circuit by the Self team, taking in input the public list of OFAC sanctioned entities. There are three different trees used to match different data of the ID or passport:

1. Matching the passport number and the nationality: this provides an absolute and high confidence match (only for passports).
2. Matching the name and date of birth: this provides a high probability match (for both passports and IDs).
3. Matching the name and year of birth: this provides a partial match (for both passports and IDs).

For each sanctioned entity the relevant information is extracted by the Self team, and hashed with Poseidon to create a `key` that is then stored in the SMT. The resulting tree, and in particular its root, is then published. In the disclose circuit, the relevant information is extracted from the ID or passport, hashed, and then, to prove non-inclusion in the OFAC list, an SMT non-inclusion proof must be provided to the circuit, which ensures that the ID is not associated with a sanctioned entity.

As an example, for the passport number and nationality check, the following template is used:

```
template OFAC_PASSPORT_NUMBER(nLevels) {
    signal input dg1[93];

    signal input smt_leaf_key;
    signal input smt_root;
    signal input smt_siblings[nLevels];

    component poseidon_hasher = Poseidon(12);
    for (var i = 0; i < 9; i++) { // passport number
        poseidon_hasher.inputs[i] <== dg1[49 + i];
    }
    for (var i = 0; i < 3; i++) { // nationality
        poseidon_hasher.inputs[9 + i] <== dg1[59 + i];
    }

    signal output ofacCheckResult <== SMTVerify(nLevels)
(poseidon_hasher.out, smt_leaf_key, smt_root, smt_siblings, 0);
}
```

From the data group 1 (DG1) of the ID or passport, the passport number is extracted from bytes 49 to 57. Then the nationality is extracted from bytes 59 to 61. The ASCII values are hashed using the Poseidon hash function, and the resulting hash is used as the lookup key in the SMT. The `SMTVerify` template is then used, configured in non-inclusion mode, to verify that the provided Merkle proof is a valid non-inclusion proof for the given key.

Sparse Merkle tree overview

The base implementation of the SMT is based on zk-kit and is used off-circuit to compute the tree root and the Merkle proofs. A sparse Merkle tree is a data structure that allows for an authenticated key-value store: in most implementations the tree is *complete*, meaning that every possible key is included as a leaf in the tree. In zk-kit's implementation, this structure is more similar to an uncompacted trie, where the leaves can be at different depths.

The idea is that a leaf can be either:

- Zero, meaning that no key is present having a binary prefix corresponding to the leaf's position.
- Non-zero, meaning that a key is present in the tree having a binary prefix corresponding to the leaf's position: in this case the leaf is the hash of the key and the value associated with it.

The tree supports both inclusion and non-inclusion proofs.

- To prove that a **key is included** in the tree, the prover needs to provide the opening to the corresponding leaf and the correct authentication path to the root.

- To prove that a **key is not included** in the tree, the prover needs to provide the opening to the closest leaf to the key and the correct authentication path to the root. Due to the way the tree is constructed, if the closest leaf opening does not match the key, but matches the key's prefix up until the leaf's depth, then the key is not included in the tree. In other words, the leaves are stored in the tree at the position of the minimal binary prefix that is not shared with any other key in the tree.

Overview of V2 smart contracts

The V2 system represents an architectural improvement over V1, introducing more structured configuration management, enhanced verification flows, and better separation of concerns between the hub and registry layers. The contracts support both passport and EU ID card attestations, with extensible verifier mappings for other cryptographic signature schemes.

The V2 Self contracts center around `IdentityVerificationHubImplV2`, which continues to manage interactions between users' zero-knowledge proofs and the on-chain identity registry. The contract follows an upgradeable proxy pattern using ERC-7201 storage.

V2 introduces a structured configuration system where verification rules are stored using deterministic config IDs generated via SHA256 hashing. Configurations are set through `setVerificationConfigV2()`. A sample config is defined as follows:

```
struct VerificationConfigV2 {
    bool olderThanEnabled;
    uint256 olderThan;
    bool forbiddenCountriesEnabled;
    uint256[4] forbiddenCountriesListPacked;
    bool[3] ofacEnabled;
}
```

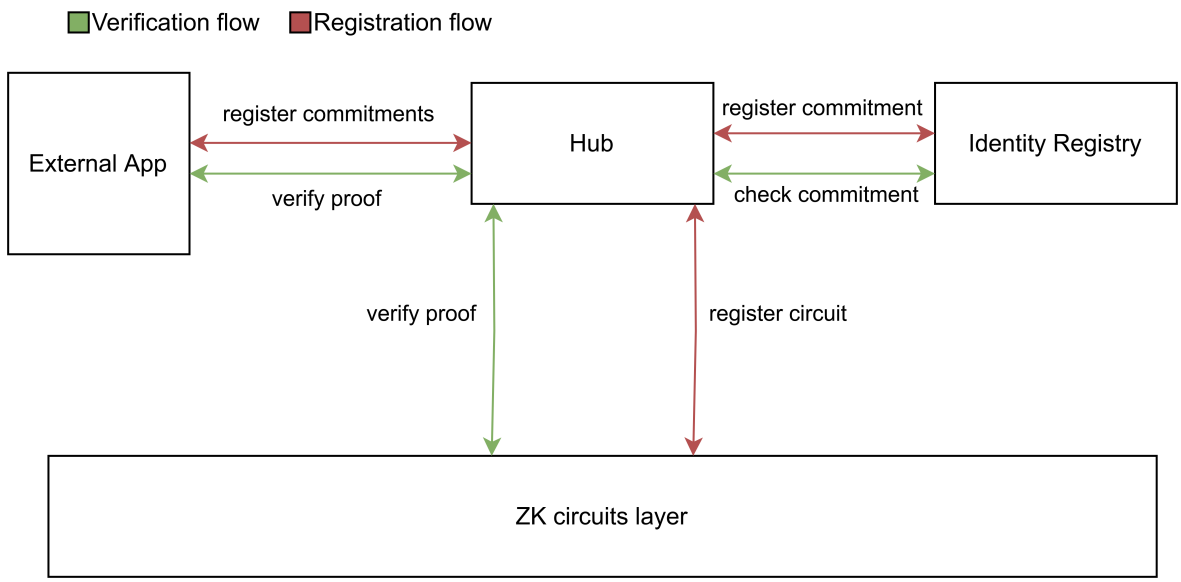
The hub integrates with multiple circuit verifiers organized by attestation type (passport, ID) and signature algorithm. Circuit constants are managed through `CircuitConstantsV2` which provides indices for extracting specific values from proof outputs.

External applications integrate through the `SelfVerificationRoot` abstract contract, which provides a standardized interface for proof verification. Applications call `verifySelfProof()` and receive callbacks through `onVerificationSuccess()`.

The IdentityRegistry contracts are the on-chain ledger for user identity commitments. They use a Sparse Merkle Tree (SMT) to store and prove the existence of identity commitments.

Two logic flows are particularly relevant in how they interact with all the components of the V2 system. First is the need to register committments, and inherently the circuit. The second is an external app (the `SelfVerificationRoot`) wanting to verify a proof. The external app

communicates exclusively with the Hub, while the Hub relays its demands to registry and the ZK layer.



The ZK layer consists of both on-chain and off-chain components. On-chain, we conduct `_basicVerification`, accomplishing the following: 1. Scope and user identifier checks 2. Merkle root validation against registry 3. Date validation 4. Groth16 ZK proof verification

Findings

Below are listed the findings found during the engagement. High severity findings can be seen as so-called "priority 0" issues that need fixing (potentially urgently). Medium severity findings are most often serious findings that have less impact (or are harder to exploit) than high-severity findings. Low severity findings are most often exploitable in contrived scenarios, if at all, but still warrant reflection. Findings marked as informational are general comments that did not fit any of the other criteria.

ID	COMPONENT	NAME	RISK
#00	circuits/utils/crypto/merkle-trees/smt.circom	An attacker can craft a fake non-inclusion proof for a given key due to an aliasing bug in the SMT verifier	**High**
#01	IdentityVerificationHubImplV2.sol	Defaulting to a zeroed config on unknown configId enables verification bypass	**High**
#02	circuits	Missing boolean constraints in the Merkle tree path leads to an attacker being able to craft a fake Merkle proof for an arbitrary leaf	**High**
#03	circuits	The registration and disclosure circuits lack range checks for the input indices	**Medium**

ID	COMPONENT	NAME	RISK
#04	circuits	Duplicated logic between Passport and ID circuits	**Informational**
#05	contracts	Minor Code Hygiene and Optimizations	**Informational**
#06	tests	Off-by-two indexing in ID disclosure tests	**Informational**
#07	SMT	The sparse Merkle tree stores truncated hashes instead of full hashes, which increases the risk of a hash collision	**Informational**

#00 - An attacker can craft a fake non-inclusion proof for a given key due to an aliasing bug in the SMT verifier

Severity: High **Location:** circuits/utils/crypto/merkle-trees/smt.circom

Description.

The sparse Merkle tree data structure that is used to perform all the OFAC checks support a non-inclusion proof that can be used to prove that a key is not present in the tree. The way a non-inclusion proof is contructed is by providing the opening of a leaf node, which shares the same bit prefix as the key being proven not to be present. In other words, the opened leaf is the "closest" leaf to the key opened. The verifier checks that the provided leaf is included in the tree by checking that the path and the siblings hashes reconstruct correctly the root hash of the tree. Then, the verifier checks that the provided opening leaf is not equal to the key present in the tree.

The verifier circuit first computes the binary representation of the key to be searched in the following way:

```
// Convert the full key to bits and take only nLength bits
component num2Bits = Num2Bits(254);  // Using 254 bits for poseidon
output
num2Bits.in <== virtualKey;
```

Unfortunately, the `Num2Bits(254)` component presents an aliasing issue. Looking at the code of `Num2Bits` in circomlib, the circuit witnesses the binary representation in the `out` array, and then checks that

$$\text{in} = \sum_{i=0}^{n-1} 2^i \cdot \text{out}[i]$$

```
template Num2Bits(n) {
    signal input in;
    signal output out[n];
    var lc1=0;

    var e2=1;
    for (var i = 0; i<n; i++) {
        out[i] <-- (in >> i) & 1;
        out[i] * (out[i] -1 ) === 0;
        lc1 += out[i] * e2;
        e2 = e2+e2;
    }

    lc1 === in;
}
```

This computation is done modulo the field size, which is smaller than 2^{254}. Therefore, for approximately a quarter of the possible keys, an attacker can witness a different binary representation of the key. More concretely, if the original key is k and the field prime is p, if $k < 2^{254} - p$ the attacker can witness the binary representation of $k + p$, which will be accepted by the circuit. However, this new representation will not be equal to the original key's representation, and the new path that will be checked by the circuit will be different. This means that an attacker can craft a fake non-inclusion proof for a key which is present in the tree, by just opening the leaf corresponding to the position of $k + p$ instead of k.

Impact.

As a result of this aliasing issue, an attacker can craft a fake non-inclusion proof for a key that is present in the tree. Since every OFAC check currently implemented relies on a non-inclusion proof, this means that the attacker can bypass any OFAC check. Notice that this can be exploited only for keys that are smaller than $2^{254} - p$, which occurs approximately for one quarter of the keys.

Recommendation.

We recommend replacing the `Num2Bits` component with `Num2Bits_strict`, which does not have the aliasing issue. We also recommend replacing the later `Num2Bits` component with `Num2Bits_strict` in the same file

```
signal path_in_bits_reversed[nLength] <== Num2Bits(nLength)(smallKey);
```

or, alternatively, adding an assertion to check that `nLength` is less or equal than `253`, to avoid aliasing issues in case of changes to the `nLength` parameter in the future.

Client response.

The client has acknowledged the issue and addressed it in commit `99e8eec`.

#01 - Defaulting to a zeroed config on unknown configId enables verification bypass

Severity: High **Location:** IdentityVerificationHubImplV2.sol

Description.

The `configId` embedded in the `userContextData` acts as a policy selector: it points to a specific, owner-registered `VerificationConfigV2` struct that encodes which checks (OFAC, forbidden-countries, age gates, etc.) should be applied during a user's proof verification. When a proof is submitted, the hub decodes the first 32 bytes as `configId`, looks up the corresponding config and then enforces those rules in its `customVerify` logic. In this way, contract owners define the allowed verification behaviors off-chain and users choose which policy to invoke on-chain simply by supplying its `configId`.

```
function _executeVerificationFlow(
        SelfStructs.HubInputHeader memory header,
        bytes memory proofData,
        bytes calldata userContextData
    ) internal returns (bytes memory output, uint256 destChainId, bytes
memory userDataToPass) {
        bytes32 configId;
        uint256 userIdentifier;
        bytes calldata remainingData;
        {
            uint256 _destChainId;
            (configId, _destChainId, userIdentifier, remainingData) =
_decodeUserContextData(userContextData);
            destChainId = _destChainId;
        }

        {
            bytes memory config = _getVerificationConfigById(configId);
            ...
```

To manage these configurations, in `contracts/contracts/IdentityVerificationHubImplV2.sol` there is a function `setVerificationConfigV2` that defines a verification configuration policy set by the owner. There is also a `verificationConfigV2Exists` that helps checking if the configuration exists. However there are two issues with the current design. First, in the current logic users can choose the `configID` they want to, which is potentially different (and weaker) to the one intended for the context. Second, the `verificationConfigV2Exists` is not used inside the `_getVerificationConfigById` function. Here, if a malicious user or relayer passes an

arbitrary ID it defaults to a struct with all values set to 0 (false) if `configID` is not defined in the `VerificationConfigs` map:

```
function _getVerificationConfigById(bytes32 configId) internal view
returns (bytes memory config) {
        IdentityVerificationHubV2Storage storage $v2 =
_getIdentityVerificationHubV2Storage();
        SelfStructs.VerificationConfigV2 memory verificationConfig =
$v2._v2VerificationConfigs[configId];
        config = GenericFormatter.formatV2Config(verificationConfig);
    }
```

We have tested this on minimalistic smart contract that implements this function locally and queried it for instance on the non existing ID:

```
configID =
"0x1111111111111111111111111111111111111111111111111111111111111111"
```

and obtained:

```
{
    "0": "tuple(bool,uint256,bool,uint256[4],bool[3]):
false,0,false,0,0,0,0,false,false,false"
}
```

which is the configuration for:

```
olderThanEnabled: false
olderThan: 0
forbiddenCountriesEnabled: false
forbiddenCountriesListPacked: [0,0,0,0]
ofacEnabled: [false,false,false]
```

Moreover the existing helper `verificationConfigV2Exists` is vulnerable to an attack where the malicious user precomputes the hash of the default “all zeroes” configuration, since this will be the output of the mapping for any ID:

```
    function verificationConfigV2Exists(bytes32 configId) external view
virtual onlyProxy returns (bool exists) {
        SelfStructs.VerificationConfigV2 memory config =
getVerificationConfigV2(configId);
        return generateConfigId(config) == configId;
    }
```

In sum, there is an underlying design issue allowing malicious users or relayers to choose a trivial verification configuration. Although the contract provides `verificationConfigV2Exists(configId)`, this helper is incorrectly implemented and it is never invoked, so `_getVerificationConfigById` blindly returns defaults ("all zeroes").

Impact.

Given `configID` comes from userContextData, this vulnerability can be used to completely avoid verification checks based on the "all false" verification configuration. An attacker can thus bypass OFAC screening, country-ban checks, and age gates simply by choosing an arbitrary configId, because all of the `ofacEnabled`, `forbiddenCountriesEnabled`, and `olderThanEnabled` flags will be false. For instance:

```
function verifyPassport(
        SelfStructs.VerificationConfigV2 memory verificationConfig,
        SelfStructs.PassportOutput memory passportOutput
    ) internal pure returns (SelfStructs.GenericDiscloseOutputV2 memory)
{
        if (
            verificationConfig.ofacEnabled[0] ||
verificationConfig.ofacEnabled[1] || verificationConfig.ofacEnabled[2]
        ) {
            if (
                !CircuitAttributeHandlerV2.compareOfac(
                    AttestationId.E_PASSPORT,
                    passportOutput.revealedDataPacked,
                    verificationConfig.ofacEnabled[0],
                    verificationConfig.ofacEnabled[1],
                    verificationConfig.ofacEnabled[2]
                )
            ) {
                revert InvalidOfacCheck();
            }
        }
        ....
```

Recommendation. It is recommended to review the design of the custom configuration to avoid users to choose arbitrary and trivial values for `configID`. If this is something that is context dependent, it should be possible for the dapp to independently enforce a given `configID`, without relying on the ID declared by the user, which could be maliciously chosen. It is recommended also to fix and use the existing `verificationConfigV2Exists` helper to gate lookups and revert early. It is important that the fix keeps track on whether `configID` has really been set by the owner to verify its existence (for instance by means of an auxiliary mapping on IDs).

Client response. Client has acknowledged the issue, and is working on a fix to prevent malicious users or relayers to choose the `configID`, and to revert on non-existing IDs.

#02 - Missing boolean constraints in the Merkle tree path leads to an attacker being able to craft a fake Merkle proof for an arbitrary leaf

Severity: High **Location:** circuits

Description.

The `BinaryMerkleRoot` template is used in multiple places in the circuits to recover the root of a binary Merkle tree. This template is used to verify the Merkle proof of a leaf in the tree, by checking that the provided path and siblings reconstruct the honest root hash of the tree. The template is implemented as follows.

```
template BinaryMerkleRoot(MAX_DEPTH) {
    signal input leaf, depth, indices[MAX_DEPTH], siblings[MAX_DEPTH];

    signal output out;

    signal nodes[MAX_DEPTH + 1];
    nodes[0] <== leaf;

    signal roots[MAX_DEPTH];
    var root = 0;

    for (var i = 0; i < MAX_DEPTH; i++) {
        var isDepth = IsEqual()([depth, i]);

        roots[i] <== isDepth * nodes[i];

        root += roots[i];

        var c[2][2] = [ [nodes[i], siblings[i]], [siblings[i], nodes[i]]
];
        var childNodes[2] = MultiMux1(2)(c, indices[i]);

        nodes[i + 1] <== Poseidon(2)(childNodes);
    }

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

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

Notice that this template does not enforce any boolean constraints on the `indices` array. The `MultiMux1` component used to compute the inputs to the `Poseidon`, and is

implemented as follows.

```
template MultiMux1(n) {
    signal input c[n][2];  // Constants
    signal input s;   // Selector
    signal output out[n];

    for (var i=0; i<n; i++) {

        out[i] <== (c[i][1] - c[i][0])*s + c[i][0];

    }
}
```

This template works correctly when the `s` input is a boolean value, but it does not enforce that the `s` input is a boolean value.

As a result, an attacker can craft fake Merkle proofs for any leaf value of their choosing by carefully constructing the `indices` and `siblings` array such that the input to the Poseidon call matches a real leaf pair. More concretely, an attacker can choose an arbitrary `leaf` value and does need to solve the following system of equations over the Circom field:

```
(sibling - leaf) * index + leaf = honest_left_leaf
(leaf - sibling) * index + sibling = honest_right_leaf
```

where `honest_left_leaf` and `honest_right_leaf` are the two nodes directly below the honest root of the tree. This system of equations is trivial to solve, as it is a linear system with two equations and two unknowns (the `index` and the `sibling`). The computed Poseidon hash by the template will be equal to the honest root hash of the tree. At this point, they can use the computed index and sibling as a valid length-1 proof for any leaf of their choosing.

Impact.

As a result of this issue, an attacker can craft a fake Merkle proof for any leaf value of their choosing. Since the Merkle tree is used to perform multiple checks throughout the circuits, this means that the attacker can bypass any check that relies on a Merkle proof, for example the inclusion of the CSCA and DSC certificates in the certificates trees.

Recommendation.

We recommend adding boolean constraints to the `indices` array in the `BinaryMerkleRoot` template, to ensure that the `MultiMux1` component works correctly.

Client response.

The client addressed this issue independently in commit `8801c6c` as a result of a coordinated disclosure with Ethereum Foundation PSE team, which is the owner of the library containing the `BinaryMerkleRoot` template. After independently discovering the issue, we also initiated a coordinated disclosure with the Ethereum Foundation PSE team, which was already aware of the issue and is planning to address it in a future release of the library.

#03 - The registration and disclosure circuits lack range checks for the input indices

Severity: Medium **Location:** circuits

Description.

In multiple places in the registration and disclosure circuits, there is the usage of `LessEqThan` template, for example in `register_id.circom`:

```
signal dsc_pubKey_offset_in_range <== LessEqThan(12)([
    dsc_pubKey_offset + dsc_pubKey_actual_size,
    raw_dsc_actual_length
]);
dsc_pubKey_offset_in_range === 1;
```

There are actually two separate issues with this code.

- The computation `dsc_pubKey_offset + dsc_pubKey_actual_size` could overflow or underflow the field, leading to a wrong check semantics.
- The `LessEqThan` template checks that the first input is less than or equal to the second input, but it assumes that both of the inputs are between 0 and $2^{12} - 1$. If this is not the case, the circuit will produce some unexpected behaviour, as for example `LessEqThan(12)` with input values $p - 1$ and 0 will return 1 instead of 0. In this context, the intended range of the indices is to be between 0 and $2^{12} - 1$, so this check could pass even if the indices are outside this range, leading to unexpected behaviour in the circuit.

Impact.

An attacker can provide incorrect indices to the circuit, for example negative indices, which pass the range checks. We could not find any concrete exploit, as incorrect indices make the proof fail in subsequent checks. However, they can lead to unexpected behaviour in the circuits, if for example some other circuits rely on the fact that indices are in the correct range.

Recommendation.

We recommend to ensure that every computation done with indices does not overflow or underflow the field, by adding additional range checks. Additionally, we recommend to add range checks to the input signals of the `LessEqThan` template, to ensure that there is no unexpected behaviour when the inputs are outside the intended range.

Client response.

Client has acknowledged this issue and addressed it on commit `49de549`.

#04 - Duplicated logic between Passport and ID circuits

Severity: Informational **Location:** circuits

Description. The newly introduced circuits for ID cards are almost identical to the ones for passport, changing only the length of the data group and the positions of some attributes within the data group. This logic is specified in:

- `circuits/register/register.circom`
- `circuits/register_id/register_id.circom`
- `circuits/disclose/vc_and_disclose.circom`
- `circuits/disclose/vc_and_disclose_id.circom`
- `circuits/utils/passport/disclose/disclose.circom`
- `circuits/utils/passport/disclose/disclose_id.circom`

This currently does not pose a security threat, but future changes to this circuits need to be strictly checked across document types, and make more likely the introduction of divergent behaviour between them.

For instance the only substantial difference between `register_id.circom` and `register.circom` in 185 lines of code are two lines, namely:

```
    var attestation_id = 1; /// or =2 for IDs
```

and:

```
    var DG1_LEN = 93; /// or =95 for IDs
```

Recommendation. It is recommended to consider a refactor that keeps the common bevahiour in one file, with specializations for the length and positions for various document types.

Client response. Client has acknowledged that this can be done for certain currently duplicated files like the registration circuits, but could be more difficult for others. They will explore this issue further and consider a refactor.

#05 - Minor Code Hygiene and Optimizations

Severity: Informational **Location:** contracts

Description.

Several small clean-ups across the codebase can shave gas and tighten maintenance:

- The modifier `onlyProxy` is applied to the internal function `getVerificationConfigV2`. Given that all external functions which call `getVerificationConfigV2` are covered by the `onlyProxy` modifier, there is no need for internal functions to use the check as well.
- The `IdCardAttributeHandler.sol` is superseded by `CircuitAttributeHandlerV2` and currently not used.
- In `CircuitAttributeHandlerV2` the OFAC field positions for passports is defined as (90,92) but for IDs this is (92,92) instead of (92,93). This does not currently affect the logic since OFAC positions are computed with respect to start position but this can be unified for consistency and avoid issues in future changes.
- In `IdentityRegistryIdCardImplV1.sol` the import `import {UUPSUpgradeable} from "@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol";` is not necessary given that this is handled by `upgradeable/ImplRoot.sol` Similarly for `IdentityRegistryImplV1.sol` and `UUPSUpgradeable,Ownable2StepUpgradeable`.
- `Strings` is imported but not used in `IdentityRegistryImplV1.sol`.

Client response. Client has acknowledged these suggestions and is working on fixes.

#06 - Off-by-two indexing in ID disclosure tests

Severity: Informational **Location:** tests

Description.

The tests in `circuits/tests/disclose/vc_and_disclose_id.test.ts` exercise the `circuits/circuits/disclose/vc_and_disclose_id.circom` circuit. However, there are some bugs that make these tests inaccurate:

In line 144 it is necessary to loop until 90 bytes instead of 88, given the format for ID cards.

```
        for (let i = 0; i < 88; i++) {
          if (selector_dg1[i] == '1') {
            const char = String.fromCharCode(Number(inputs.dg1[i + 5]));
            assert(reveal_unpacked[i] == char, 'Should reveal the right
character');
          } else {
            assert(reveal_unpacked[i] == '\x00', 'Should not reveal');
          }
        }
```

Similarly, in line 190-191 the checks should be on positions 90 and 91, instead of 88 and 89.

```
    expect(reveal_unpacked[88]).to.equal('\x00');
    expect(reveal_unpacked[89]).to.equal('\x00');
```

Recommendation.

Currently, the tests pass because bytes 89 and 90 are not supposed to be revealed in the test. We recommend to fix this nevertheless to allow the test to catch future bugs in this region.

Client response.

Client has acknowledged the issue and patched the tests accordingly on commit `bf8cf6c`.

#07 - The sparse Merkle tree stores truncated hashes instead of full hashes, which increases the risk of a hash collision

Severity: Informational **Location:** SMT

Description.

Different sparse Merkle trees are used to perform all the checks related to OFAC. The way they are used is to store the hash of banned entities in the leaves. Then, it is possible to provide a non-inclusion proof for a given key, which prover that the data contained in the passort or ID card are not in the tree.

Currently, the leaves of the sparse Merkle tree store a truncated hash of 64 bits, instead of the full hash of 254 bits. This means that, even though very low, the probability of an honest user colliding with a banned entity is not zero, which can lead to false positives in the OFAC checks.

Impact.

If an honest user data hash collides with a banned entity for the first 64 bits, the honest user will be rejected by the OFAC check. This event occurs with probability

$$p = \frac{\#\,\text{leaves in the tree}}{2^{64}}$$

where the number of leaves in the tree is the number of banned entities.

Recommendation.

We recommend changing the sparse Merkle tree implementation to store the full 254-bit hash instead of the truncated 64-bit hash. Additionally, due to the way Merkle proofs are constructed, this change would not require any modifications to the existing circuits or proofs. Indeed, the length of the proof is determined by the largest common prefix shared between two keys in the set, and not by the length of the hash stored in the leaves.

Client response.

Client has acknowledged this issue and, since it is not a critical issue, is considering addressing it in the future by storing the full 254-bit hash in the leaves of the sparse Merkle tree.