Veridise

# Auditing Report

## Hardening Blockchain Security with Formal Methods

FOR


Z-imburse

Veridise Inc.

February 25, 2025

► **Prepared For:**

Mach34
https://mach34.space/

► **Prepared By:**

Tyler Diamond
Ian Neal
Benjamin Sepanski

► **Contact Us:**

contact@veridise.com

► **Version History:**

| | |
|---|---|
| Feb. 25, 2025 | V2 |
| Dec. 03, 2024 | V1 |
| Nov. 26, 2024 | Initial Draft |

# Contents



 

# 1 Executive Summary

From Nov. 11, 2024 to Nov. 25, 2024, Mach34 engaged Veridise to conduct a security assessment of their Z-imburse. The security assessment covered smart contracts and zero-knowledge circuits written for the Aztec Network which enable receivers of grants (called "claimants") to prove receipt of an email matching reimbursement conditions in order to automatically receive funds. Veridise conducted the assessment over 6 person-weeks, with 3 security analysts reviewing the project over 2 weeks on commit `61f276f`. Due to the heavy use of Aztec-specific libraries and methodologies, Veridise engineers also investigated some of the Aztec protocol and standard library smart contracts invoked by Mach34 smart contracts.

**Project Summary.** Z-imburse has three major components. The first component consists of the `verifiers`. These are zero-knowledge circuits involved in extracting email body properties, such as reimbursement amounts, and verifying the DKIM signature via zkemail*. The Linode verifier is the only verifier that was completed and marked for review.

The second component is the `escrow`. The `escrow` contains the reimbursement entitlements themselves, and handles logic relating to calling the `verifiers`. The security assessment covered only the `escrow` and Linode `verifiers`.

The final component of the project is the `registry`. This contract contains all registered DKIM public keys, verifies the bytecode of `escrow` contracts, and acts as a central point for discovering one's active escrows. This project was referenced as necessary for the review, but otherwise considered out of scope.

**Code Assessment.** The Z-imburse developers provided the source code of the Z-imburse contracts for the code review. The source code appears to be mostly original code written by the Z-imburse developers. It contains some documentation in the form of READMEs and documentation comments on functions and storage structures. To facilitate the Veridise security analysts understanding of the code, the Z-imburse developers also wrote and shared documentation that provided more detail on the intentions of the project and design decisions they made.

The source code contained a test suite, which the Veridise security analysts noted covered testing of most `escrow` and `registry` functions, along with tests for the Linode verifier.

**Summary of Issues Detected.** The security assessment uncovered 18 issues, 8 of which are assessed to be of high or critical severity by the Veridise analysts. Specifically, reimbursements may be claimed after revocation (V-ZIM-VUL-001) or after previously being claimed (V-ZIM-VUL-002, V-ZIM-VUL-003, V-ZIM-VUL-008), the reimbursed amount may be increased up to the maximum value (V-ZIM-VUL-004, V-ZIM-VUL-005), major validations may be bypassed to submit invalid e-mails (V-ZIM-VUL-006), and revoking entitlements is currently impossible

* `https://github.com/zkemail/zkemail.nr`

(V-ZIM-VUL-007). The Veridise analysts also identified 1 medium-severity issue (V-ZIM-VUL-009) which describes the high level of trust placed in claimants, as well as 3 low-severity issues describing the high level of trust placed in the reimbursing party (V-ZIM-VUL-010), invalid non-existence checks (V-ZIM-VUL-011), and incorrect date computations (V-ZIM-VUL-012). Finally, the analysts identified 4 warnings, and 2 informational findings.

All 18 issues, 18 issues have been acknowledged by the Mach34. Of the 18 acknowledged issues, Mach34 has fixed 12 issues and provided partial fixes to 2 more. This includes the 2 high-severity issues and 6 critical issues.

**Recommendations.** After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Z-imburse.

*Warnings and unconstrained functions.* Resolve compiler warnings so that newly introduced issues can be identified more easily. Additionally, wrap all `unconstrained` calls in `unsafe` blocks. Also, verify if a function called from the standard library is unsafe before using it.

*Constraining over casting.* Avoid using the type casting expression `as u32`. Instead, take a `u32` and cast it to a `Field` to assert equality. For example, issues V-ZIM-VUL-004 and V-ZIM-VUL-005 allow an attacker to easily receive the maximum reimbursement without a valid e-mail. In both attacks, an attacker constructs a value which is *much* too large to be reimbursed, and is then cast to a `u32`. If the result had instead been constrained to be a `u32` already, successful attacks would have been much more difficult to construct.

*Header constraints.* Constrain every header field to have begun after the end sequence of a previous header field in order to avoid injection of false values in various header field bodies. Since zkemail's `constrain_header` performs out-of-bounds checks on `BoundedVec` accesses, constraining every header field accessed by z-imburse would have prevented V-ZIM-VUL-006 from being exploitable. This applies especially to the checks validating the `from` and `subject` fields.

**Disclaimer.** We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

# 2 Project Dashboard

**Table 2.1:** Application Summary.

| Name | Version | Type | Platform |
|---|---|---|---|
| Z-imburse | `61f276f` | Noir | Aztec |

**Table 2.2:** Engagement Summary.

| Dates | Method | Consultants Engaged | Level of Effort |
|---|---|---|---|
| Nov. 11–Nov. 25, 2024 | Manual | 3 | 6 person-weeks |

**Table 2.3:** Vulnerability Summary.

| Name | Number | Acknowledged | Fixed |
|---|---|---|---|
| Critical-Severity Issues | 6 | 6 | 6 |
| High-Severity Issues | 2 | 2 | 2 |
| Medium-Severity Issues | 1 | 1 | 0 |
| Low-Severity Issues | 3 | 3 | 0 |
| Warning-Severity Issues | 4 | 4 | 2 |
| Informational-Severity Issues | 2 | 2 | 2 |
| TOTAL | 18 | 18 | 12 |

**Table 2.4:** Category Breakdown.

| Name | Number |
|---|---|
| Data Validation | 7 |
| Logic Error | 4 |
| Maintainability | 3 |
| Under-Constrained Circuit | 1 |
| Denial of Service | 1 |
| Access Control | 1 |
| Optimization | 1 |

# 3 Security Assessment Goals and Scope

## 3.1 Security Assessment Goals

The engagement was scoped to provide a security assessment of Z-imburse's smart contracts. During the assessment, the security analysts aimed to answer questions such as:

- Are the results of all `unconstrained` functions properly constrained?
- Are any unsafe methods such as `get_unchecked()` properly validated?
- Are any common zero-knowledge vulnerabilities such as missing state checks, incorrect control-flow handling, or privacy leakage present?
- Is all parsing implemented robustly, preventing injection of user-controlled data?
- Are nullifiers deterministic?
- Can claimants claim reimbursements when they have been nullified by the admin or been claimed already?
- Can any information about the claimant be leaked through the protocol?
- Is the `zkemail.nr` library used correctly to validate e-mail addresses?
- Can a claimant receive a reimbursement larger than they would receive if they submitted the e-mail publicly?
- Can an admin intentionally prevent a valid reimbursement from occurring?

## 3.2 Security Assessment Methodology & Scope

**Security Assessment Methodology and Scope.** To address the questions above, the security assessment involved a thorough review by human experts.

The scope of this security assessment is limited to the below folders of the source code provided by the Z-imburse developers, which contains the Noir circuits and Aztec Network contract implementation of the Z-imburse. The Veridise analysts referenced Aztec Network and Noir source code to understand functions from dependencies, and referenced out-of-scope code as necessary to understand the application.

At the direction of the Mach34 team, Veridise did not focus on the functionality relating to the United reimbursements, and instead focused most efforts on the Linode reimbursements. The Veridise analysts reviewed the `tests/` directory listed below for appropriate integration testing, but were unable to successfully run the full test suite due to errors with Aztec Network's version 0.57.0 known to Mach34.

1. `contracts/z_imburse_escrow/`
    a) `src/main.nr`
    b) `src/types/entitlement_note.nr`
    c) `src/verifiers.nr`
    d) `src/library_methods/linode.nr`
    e) `src/library_methods/entitlements.nr`

2. `contracts/z_imburse_registry/src/test/escrow.nr`

*Methodology*. Veridise security analysts inspected the provided tests and read the Z-imburse documentation. They then began a review of the code.

During the security assessment, the Veridise security analysts regularly met with the Z-imburse developers to ask questions about the code.

## 3.3 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 3.1.

**Table 3.1:** Severity Breakdown.

| | Somewhat Bad | Bad | Very Bad | Protocol Breaking |
|---|---|---|---|---|
| Not Likely | Info | Warning | Low | Medium |
| Likely | Warning | Low | Medium | High |
| Very Likely | Low | Medium | High | Critical |

The likelihood of a vulnerability is evaluated according to the Table 3.2.

**Table 3.2:** Likelihood Breakdown

| | |
|---|---|
| Not Likely | A small set of users must make a specific mistake |
| Likely | Requires a complex series of steps by almost any user(s)<br>- OR -<br>Requires a small set of users to perform an action |
| Very Likely | Can be easily performed by almost anyone |

The impact of a vulnerability is evaluated according to the Table 3.3:

**Table 3.3:** Impact Breakdown

| | |
|---|---|
| Somewhat Bad | Inconveniences a small number of users and can be fixed by the user |
| Bad | Affects a large number of people and can be fixed by the user<br>- OR -<br>Affects a very small number of people and requires aid to fix |
| Very Bad | Affects a large number of people and requires aid to fix<br>- OR -<br>Disrupts the intended behavior of the protocol for a small group of users through no fault of their own |
| Protocol Breaking | Disrupts the intended behavior of the protocol for a large group of users through no fault of their own |

# 4 Vulnerability Report

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 4.1 summarizes the issues discovered:

**Table 4.1:** Summary of Discovered Vulnerabilities.

| ID | Description | Severity | Status |
|---|---|---|---|
| V-ZIM-VUL-001 | Revoked entitlements can still be claimed | Critical | Fixed |
| V-ZIM-VUL-002 | Linode spot entitlements can be repeatedly . . . | Critical | Fixed |
| V-ZIM-VUL-003 | United spot entitlements can be repeatedly . . . | Critical | Fixed |
| V-ZIM-VUL-004 | Amount may be increased up to maximum | Critical | Fixed |
| V-ZIM-VUL-005 | Small linode billing amounts receive . . . | Critical | Fixed |
| V-ZIM-VUL-006 | BoundedVec marshaling bypasses validation | Critical | Fixed |
| V-ZIM-VUL-007 | Entitlements are irrevocable | High | Fixed |
| V-ZIM-VUL-008 | Non-deterministic recurring . . . | High | Fixed |
| V-ZIM-VUL-009 | Excess reimbursements may be sold to . . . | Medium | Acknowledged |
| V-ZIM-VUL-010 | Centralization Risk | Low | Acknowledged |
| V-ZIM-VUL-011 | Invalid non-existence checks for notes and . . . | Low | Acknowledged |
| V-ZIM-VUL-012 | Incorrect date computations | Low | Acknowledged |
| V-ZIM-VUL-013 | Missing/incorrect documentation | Warning | Partially Fixed |
| V-ZIM-VUL-014 | The fields of an EntitlementNote should . . . | Warning | Fixed |
| V-ZIM-VUL-015 | Unused/duplicate program constructs | Warning | Partially Fixed |
| V-ZIM-VUL-016 | EntitlementNote.eq missing checks | Warning | Fixed |
| V-ZIM-VUL-017 | check_nullifier_txe should be unconstrained | Info | Fixed |
| V-ZIM-VUL-018 | Optimization opportunities | Info | Fixed |

## 4.1 Detailed Description of Issues

### 4.1.1 V-ZIM-VUL-001: Revoked entitlements can still be claimed

| | | | |
|---|---|---|---|
| **Severity** | Critical | **Commit** | 61f276f |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `contracts/z_imburse_escrow/src/main.nr` | | |
| **Location(s)** | revoke_entitlement() | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/23` | | |

When an admin grants an entitlement to a claimant, the `give_entitlement()` library method is called which calls `EntitlementSet.add_entitlement_with_receipt()`. This latter method will insert the `EntitlementNote` into the map storage at the keys for both the `admin` and `recipient`. Note that since the Note header for each insertion will be different from the `storage_slot` being different, the note hash and nullifier will be different for each entry.

When requesting a reimbursement, the `linode()` and `reimburse_united()` library methods look up the `EntitlementNote` in the map keyed with the recipient's address.

However, the `revoke_entitlement()` function looks up the `EntitlementNote` to nullify keyed with the admin's address. Therefore, claimants can still claim reimbursements as the `EntitlementNote` scoped to their address has not been nullified.

**Impact** Claimants can claim entitlements even after the admin has intended to revoke them.

**Recommendation** Stop inserting the `EntitlementNote` at two separate keys in order to avoid the confusion that caused this issue. If both insertions are required, then revoke the claimant's `EntitlementNote` entry as well in `revoke_entitlement()`.

Note that usage of `get_notes()` can be dangerous, as it does not emit a nullifier for the note, and may return nullified notes. See https://github.com/AztecProtocol/aztec-packages/pull/4940.

**Developer Response** We now track a map of `SharedImmutables` which are set to `true` to nullify an entitlement. A shared entitlement nullifier is now used between the admin and the recipient.

**Updated Veridise Response** The core idea behind the provided fix does resolve this issue. However, it should be noted that

1. Shared Immutable was recently removed: https://github.com/AztecProtocol/aztec-packages/commit/a9f3b5f6e7e5bc9d4bc9c0600b492a5e0cd2c1d9010fe1d9d69f1c6ff0cd2b9cf444a2f123e84fd74d3f4fdcffa9746ac7d30497
2. The `status` flag of `options` is unused by `pop_notes()`, so adding `set_status(NoteStatus.ACTIVE)` will only mislead code readers: https://github.com/Mach-34/z-imburse/compare/audit/make-entitlements-revocable..audit/make-revoked-entitlements-non-claimable#diff-ab22c99dc5c1ec92a40d1993c666c8a42e1fa55fb279463fa83ef7571de

In general, the Veridise team recommends relying on the network-native nullifiers rather than reimplementing them manually. This may require changing the structure of the recurring entitlement notes.

### 4.1.2 V-ZIM-VUL-002: Linode spot entitlements can be repeatedly claimed

| | | | |
|---|---|---|---|
| **Severity** | Critical | **Commit** | 61f276f |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `contracts/z_imburse_escrow/src/main.nr` | | |
| **Location(s)** | reimburse_linode_spot(), check_nullifier() | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/23` | | |

The `reimburse_linode_spot()` function calls the `EntitlementNote.check_nullifier_txe()` method to calculate the nullifier and is then subsequently emitted.

```
let nullifier = entitlement.check_nullifier_txe(&mut context); // <-- doesn't
    actually constrain nullifier to exist

// broadcast the note nullifier
context.push_nullifier(nullifier);
```

**Snippet 4.1:** Snippet from `contracts/z_imburse_escrow/src/main.nr:reimburse_linode_spot()`

However, the `check_nullifier_txe()` method calls `compute_nullifier_without_context()` to calculate the nullifier value. This is an unconstrained function, so any value can be used during witness generation. Therefore, there is no guarantee on the link between the Note and the returned nullifier. Be aware that this same issue exists in `check_nullifier()`.

**Impact** Any value can be emitted as the nullifier, and therefore one can continuously reimburse themselves as long as they had a spot `EntitlementNote` destined for themselves at any point in time.

**Recommendation** Do not use `compute_nullifier_without_context()` to compute the nullifier, use the constrained `compute_nullifier()`. Additionally, do not attempt to prove that a nullifier has not been emitted before, simply rely on the kernel circuits which will perform this check anyways.

**Developer Response** The developers have acknowledged the issue and now calculate nullifiers via the `EntitlementNote.derive_shared_nullifier()` function, which is constrained.

### 4.1.3 V-ZIM-VUL-003: United spot entitlements can be repeatedly claimed

| | | | |
|---|---|---|---|
| **Severity** | Critical | **Commit** | 61f276f |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `contracts/z_imburse_escrow/src/library_methods/united.nr` | | |
| **Location(s)** | reimburse_united() | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/23` | | |

The `reimburse_united()` function uses the `check_nullifier()` method to calculate the nullifier for an `EntitlementNote`. As mentioned in V-ZIM-VUL-002, then `check_nullifer()` method calls `compute_nullifier_without_context()`. This is an unconstrained method, and therefore any value can be returned from it. Therefore, there is no guarantee on the link between the Note and the returned nullifier.

**Impact** Any value can be emitted as the nullifier, and therefore one can continuously reimburse themselves as long as they had a spot `EntitlementNote` destined for themselves at any point in time.

**Recommendation** Do not use `compute_nullifier_without_context()` to compute the nullifier, use the constrained `compute_nullifier()`. Additionally, do not attempt to prove that a nullifier has not been emitted before, simply rely on the kernel circuits which will perform this check anyways.

**Developer Response** The developers have acknowledge the issue, but note that United entitlements were not in scope for the audit. Similar to the Linode issue, this is now fixed by using the `EntitlementNote.derive_shared_nullifier()` calculation, which is constrained.

### 4.1.4 V-ZIM-VUL-004: Amount may be increased up to maximum

| | | | |
|---|---|---|---|
| **Severity** | Critical | **Commit** | 61f276f |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `circuits/zimburse_verifiers/src/linode/utils.nr` | | |
| **Location(s)** | extract_billed_amount() | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/25` | | |

The amount billed by Linode is parsed directly from the body of the email. This is done by taking a user-supplied sequence indicating the start and stop index of the amount. After taking some care to determine the number of digits (represented as `power + 1` in the below code), the code loops over the ASCII characters in the sequence (except for periods and commas), accumulating the value into `amount`.

```
for i in 1..MAX_BILLED_AMOUNT {
        if i < amount_sequence.length {
            let byte = body.get_unchecked(amount_sequence.index + i);
            // check that byte is not a comma or period
            if (byte != 44) & (byte != 46) {
                amount = amount + ((byte as Field - 48) * (10 as Field).pow_32(power)
    );
                power = power - 1;
            }
        }
    }
```

**Snippet 4.2:** Snippet from `extract_billed_amount()` intended to loop over the numerals in the sequence. `MAX_BILLED_AMOUNT` is 13, intended to be a limit on the representation length of the `amount`.

No validation is performed to ensure that each `byte` accumulated into `amount` is a numeric digit. Additionally, no validation is performed to ensure that the sequence end stops at the end of the number. This means that, by extending the sequence to include the ASCII characters from the email after the actual billed amount, a malicious prover may make the computed `amount` much larger than intended.

Compounding the effect of these two missing validations, the `amount` is accumulated into a `Field`. This is done since regular-sized amounts will not overflow. However, since minimal validation is performed on the sequence length, this gives the attacker a large amount of freedom to compute a huge `amount`, ensuring they receive the maximum reimbursement.

An example attack is shown in the below proof-of-concept.

```
#[test]
fn test_extract_billed_amount() {
    let value = "$1,000.00.\r\n\r\nThank you.".as_bytes();
    let mut storage: [u8; MAX_LINODE_EMAIL_BODY_LENGTH] = [0;
    MAX_LINODE_EMAIL_BODY_LENGTH];
    let sequence_length = value.len();
    let text_length = LB_PAYMENT_TEXT_LEN + sequence_length;
    for i in 0..text_length{
        if i < LB_PAYMENT_TEXT_LEN {
            storage[i] = LINODE_BILLING_PAYMENT_TEXT.as_bytes()[i];
```

```
        }
        else {
            storage[i] = value[i - LB_PAYMENT_TEXT_LEN];
        }
    }
    let body: BoundedVec<u8, MAX_LINODE_EMAIL_BODY_LENGTH> = BoundedVec {
        storage,
        len: text_length
    };
    let amount_sequence = Sequence {
        index: LB_PAYMENT_TEXT_LEN,
        length: sequence_length
    };
    let extracted_amount = extract_billed_amount(body, amount_sequence);
    assert(extracted_amount == 3579499008, extracted_amount);
}
```

**Snippet 4.3:** Proof-of-concept demonstrating the amount may be increased.

**Impact** Every claimant may receive the maximum reimbursement by forcing the computed `amount` to be much larger than the actual amount.

**Recommendation** Consider forcing the user to provide the expected value, then computing its expected serialization. This will ensure the snippet from the body is a valid dollar amount.

**Developer Response** The developers acknowledge the issue and now ensure the bytes read are valid numeric characters.

**Updated Veridise Response** This does resolve the issue, but may be vulnerable to changes in the e-mail structure. For future versions, we would recommend fully parsing the e-mail.

### 4.1.5 V-ZIM-VUL-005: Small linode billing amounts receive maximum reimbursement

| | | | |
|---|---|---|---|
| **Severity** | Critical | **Commit** | 61f276f |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `circuits/zimburse_verifiers/src/linode/utils.nr` | | |
| **Location(s)** | extract_billed_amount() | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/25` | | |

Substituting `$0.67` into the proof-of-concept defined in V-ZIM-VUL-004 demonstrates that, instead of outputting `67` cents, `extract_billed_amount()` returns `1781993765` cents, triggering a maximal reimbursement for any budget less than $17 million.

This strange behavior is due to the `power = power - 1` expression when the character after the `$` is `0`. `power` is supposed to represent the power of 10 to weight each digit by. When the amount is `$0.67`, the `power` is set to `2` instead of `3`. In this case, the power used in the final iteration of the sequence is `-1` (in the base field) instead of `0`.

```
// ensure first character is '$'
assert(body.get_unchecked(amount_sequence.index) == 36);

// if second character is a zero then decrement power
if body.get_unchecked(amount_sequence.index + 1) == 48 {
    power = power - 1;
}

for i in 1..MAX_BILLED_AMOUNT {
    if i < amount_sequence.length {
        let byte = body.get_unchecked(amount_sequence.index + i);
        // check that byte is not a comma or period
        if (byte != 44) & (byte != 46) {
            amount = amount + ((byte as Field - 48) * (10 as Field).pow_32(power));
            power = power - 1;
        }
    }
}
```

**Snippet 4.4:** Snippet from `extract_billed_amount()`.

**Impact** Users may intentionally bill small amounts to receive the maximum reimbursement.

**Recommendation** As recommended in V-ZIM-VUL-004, take the `amount` as input and serialize it, rather than the other way around.

Additionally, increase the size of the test suite for `extract_billed_amount()` to contain a large number of randomly generated amounts, as well as certain edge cases such as each power of 10, and one less than each power of 10.

Finally, note that if the amount were *constrained to be* a `u32` instead of *cast* to a `u32` (which performs truncation), this error would have led to denial of service instead of theft of funds. As

a general rule, constraining the value to be small is safer than casting it to a small value (if it is known a priori to be small during correct execution).

**Developer Response** The developers have acknowledged the issue and now parse the `amount` backwards, increasing the power whenever a numeric digit is encountered.

**Updated Veridise Response** This does address the issue, but may introduce severe errors if the e-mail format changes. See related comments in V-ZIM-VUL-005.

### 4.1.6 V-ZIM-VUL-006: BoundedVec marshaling bypasses validation

| | | | |
|---|---|---|---|
| **Severity** | Critical | **Commit** | 61f276f |
| **Type** | Under-Constrained Ci | **Status** | Fixed |
| **File(s)** | `circuits/zimburse_verifiers/src/linode/constants.nr` | | |
| **Location(s)** | marshal() | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/28` | | |

`marshal()` converts the `LinodeBillingParamsContract` into a `LinodeBillingParams` struct.

```
fn marshal(self) -> LinodeBillingParams {
    LinodeBillingParams {
        header: BoundedVec { storage: self.header, len: self.header_length },
```

**Snippet 4.5:** Snippet from `marshal()`

Constructing a `BoundedVec` using user-supplied values for the `storage` allows a malicious prover to supply non-zero values in the entries from the length index out to the capacity. This violates a key invariant of the `BoundedVec` that is relied upon for several checks. For example, `check_from_linode_billing()` and `check_subject_linode_billing_receipt()` do not check that the provided indices are less than the header length, relying on the assertions to fail due to the assumption that out of bounds accesses will return zero.

**Impact** A malicious prover may bypass the `from` check and `subject` check. Since Linode uses the same DKIM key for `billing@linode.com` as they do for their customer support line, an attacker could phish the support line and then supply fake header values in the space between `self.header_length` and the vector capacity.

**Recommendation** Construct the `BoundedVecs` by pushing items. This will ensure that entries past the length remain zero.

Additionally, call `constrain_header_field()` before accessing *any* header.

**Developer Response** The developers acknowledge the issue and now check all entries past the `self.header_length` are 0 values.

### 4.1.7 V-ZIM-VUL-007: Entitlements are irrevocable

| | | | |
|---|---|---|---|
| **Severity** | High | **Commit** | 61f276f |
| **Type** | Denial of Service | **Status** | Fixed |
| **File(s)** | `contracts/z_imburse_escrow/src/main.nr` | | |
| **Location(s)** | revoke_entitlement() | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/22`, b6cc52e | | |

The `revoke_entitlement()` function is used to nullify an entitlement so that a claimant cannot claim reimbursements for it anymore. The entitlement in question is received from the `nullify_entitlement()` function which will look up the entitlement from the `EntitlementSet` and return the entitlement via the `pop_notes()` function. Using this `PrivateSet` function automatically broadcasts nullifiers for all Notes returned from it.

However, later in the `revoke_entitlement()` function, the `nullifier` is calculated and used with `context.push_nullifier()`. This will cause invocation of the `revoke_entitlement()` function to always revert, as a duplicate nullifier is broadcasted.

**Impact** Entitlements can never be revoked, and therefore claimants may claim more reimbursements than the admin intends.

**Recommendation** Remove the `context.push_nullifier(nullifier)` call as it is unnecessary.

**Developer Response** The developers removed the extra `context.push_nullifier(nullifier)` from the `revoke_entitlement()` function. They also added a test to ensure the revocation succeeds on the happy path.

### 4.1.8 V-ZIM-VUL-008: Non-deterministic recurring reimbursement nullifier

| | | | |
|---|---|---|---|
| **Severity** | High | **Commit** | 61f276f |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | contracts/z_imburse_escrow/src/types/entitlement_note.nr, circuits/date_parser/src/lib.nr | | |
| **Location(s)** | derive_recurring_nullifier(), to_unix_month() | | |
| **Confirmed Fix At** | https://github.com/Mach-34/z-imburse/pull/26/ | | |

`to_unix_month()` is intended to return a unique timestamp for each month. However, it ignores the timezone offset and includes the offset from the hours/minutes/seconds of the email timestamp.

This means that, by changing the local timezone at the beginning of a month, or by repeatedly requesting email receipts at different times of day, an attacker may cause `to_unix_month()` to output multiple distinct timestamps for the same email.

```
(days_in_complete_years + days_in_current_year) * 86400 + datetime.time as u32
```

**Snippet 4.6:** Snippet from `to_unix_month()`

The result of `to_unix_month()` is used to derive a monthly nullifier for recurring reimbursements.

```
fn derive_recurring_nullifier(self, timestamp: Field) -> Field {
      poseidon2_hash_with_separator(
          [self.randomness, timestamp],
          GENERATOR_INDEX__NOTE_NULLIFIER as Field
      )
  }
```

**Snippet 4.7:** Definition of `derive_recurring_nulliifer()`. The `timestamp` used is the result of applying `to_unix_month()` to the email header's timestamp.

Since distinct timestamps returned from `to_unix_month()` lead to distinct nullifier, an attacker can claim the recurring reimbursement multiple times during a single month.

**Impact** An attacker may claim the funds from a monthly recurring reimbursement once per distinct timestamp they are able to produce.

If a service resends copies of a receipt by request, this allows unlimited reimbursements by the claimant.

**Recommendation** To fix `to_unix_month()`, do not include the hours, minutes, or seconds. Additionally, factor in the local timezone to the timestamp.

In general, since many services will resend copies of email receipts upon request, using the email timestamp for a nullifier is prone to exploitation. Instead, a unique ID associated to the reimbursed expenditure (e.g. a combined order ID number and order date) should be used to nullify the reimbursement.

**Developer Response** The developers have acknowledged the issue and now extract the payment month and year from the body of the email. This defeats re-requesting an email receipt, and additionally the time and timezone are now irrelevant to the date calculation as long as Linode is consistent with the body of the email.

### 4.1.9 V-ZIM-VUL-009: Excess reimbursements may be sold to third-parties

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | 61f276f |
| **Type** | Data Validation | **Status** | Acknowledged |
| **File(s)** | See issue description | | |
| **Location(s)** | See issue description | | |
| **Confirmed Fix At** | N/A | | |

The standard guarantee of a public reimbursement program is that the claimant has performed some action with a third-party service which caused the service to deliver a receipt to the claimant. The claimant may then submit the receipt to the reimbursing party as evidence that the action was performed.

While this leaves room for dishonesty (e.g. returning the reimbursed item and buying a less expensive alternative, or pocketing the cash), it does ensure the reimbursed party did at one point in time perform the required action.

The z-imburse system aims to provide a privacy-preserving implementation of this scheme. However, it does not currently validate the recipient of an email. This creates a potential market for emails conforming to the reimbursement requirements. Malicious parties may build a protocol on top of z-imburse to sell off excess funds.

For example, suppose Alice has been given an entitlement for a $5,000 reimbursement for Linode expenses. Alice expects to only spend $3,000 on Linode, so starts an on-chain auction for the remaining $2,000. Suppose Bob wins the auction for $1,000. Bob is a Linode user already who has spent $4,500. Bob submits the reimbursal through the auction contract on behalf of Alice. The auction contract awards Bob his $1,000, and sends the remaining $3,500 to Alice.

**Impact** With a properly engineered contract, these auctions could be trustlessly performed in a fully private setting between two anonymized parties. This acts as a scaling mechanism for fraud, making it easier to perform and more difficult to detect.

Note that, in principle, this could still be done with a public reimbursement system. For example, Alice could perform her $3,000 of compute, and then offer to run $2,000 worth of compute for someone at a discounted rate. The main distinction here is the private setting.

**Recommendation** Include (salted) hashes of the emails in the public receipts for each entitlement to enable auditing. Consider validating the `to` field or account number in future designs to reduce the ease of selling excess reimbursements.

**Developer Response** While this is a very valid issue to resolve, similar to the nonstandard issue of email parsing in zkemail.nr we are electing to defer this issue to a future version of zkemail.nr that uses regex to extract out email addresses. We intend on fixing this issue by binding entitlements to email addresses in the future.

### 4.1.10 V-ZIM-VUL-010: Centralization Risk

| | | | |
|---|---|---|---|
| **Severity** | Low | **Commit** | 61f276f |
| **Type** | Access Control | **Status** | Acknowledged |
| **File(s)** | See issue description | | |
| **Location(s)** | See issue description | | |
| **Confirmed Fix At** | N/A | | |

Similar to many projects, Mach34's z-imburse declares an administrator role that is given special permissions. In particular, users must be aware of the following potential administrative abuses:

1. `contracts/z-imburse_escrow/`
    a) The escrow admin provides no solvency guarantees on their ability to pay out reimbursements. Even if they do originally deposit enough funds to pay out reimbursements, they can reimburse themselves the entire contract balance.
    b) Reimbursement recipients must validate that the time range of their reimbursements are satisfiable.
2. `contracts/z-imburse_registry`
    a) The registry admin can register any arbitrary publickey hash, and therefore he may allow arbitrary reimbursements to incorrectly validate inside of the escrow.

**Impact** If a private key were stolen, a hacker would have access to sensitive functionality that could compromise the project. For example, a malicious admin could attempt to incentivize users to perform tasks with false promises of payment.

**Recommendation** As these are all particularly sensitive operations, we would encourage the developers to utilize a decentralized governance or multi-sig contract as opposed to a single account, which introduces a single point of failure.

**Developer Response** The developers have acknowledged the centralization risks, and included the below comments.

We considered requiring escrow administrators to ensure an escrow had sufficient liquidity before creating a new entitlement. This might work with spot entitlements but would require a bound on recurring entitlements (a potentially desirable feature in itself). Ultimately, however, we do not want to impose undue capital constraints on organizations that may not have $100,000,000 token treasuries and are managing cash flow on a monthly or quarterly basis, therefore providing liquidity just in time. Potentially, we could optionally enable this feature, but in general we intend on expecting social coordination to remedy this failure.

We certainly agree with using multisigs! This is something that we consider out of scope of our protocol, however. If an organization wishes to drive their logic through a multisig, upcoming projects like Obsidian Wallet will allow a gnosis-like UI for managing this interaction. This is not ready yet so we will not do anything right now, but in the future our remedy would simply be to strongly recommend the use of a multisig when creating a new escrow

**Updated Veridise Response** We have included an additional centralization risk regarding the `z-imburse_registry` admin. We originally intended to include this risk, but overlooked including it in the report generation. A possible solution to this is that DNS record verification could provide assurances that an organization intends to use a given public key.

### 4.1.11 V-ZIM-VUL-011: Invalid non-existence checks for notes and nullifiers

| Severity | Low | Commit | 61f276f |
|---|---|---|---|
| Type | Data Validation | Status | Acknowledged |
| File(s) | See issue description | | |
| Location(s) | See issue description | | |
| Confirmed Fix At | N/A | | |

In various parts of the protocol there are two areas that do not provide the security guarantees that they imply:

1. Notes are checked for existence when using `PrivateSet.get_notes()` in order to prevent creating a duplicate Note. This does not provide any safety as a user could simply omit providing the Note to the PXE. See, for example, https://github.com/AztecProtocol/aztec-packages/pull/4940.
2. Nullifiers are checked for non-inclusion via the `check_nullifier_exists()` call. This function is unconstrained and therefore its result does not provide any safety guarantees on if a nullifier already exists.

The core of both of these issues is an attempt to prove exclusion of information for the corresponding state trees.

These two issues appear in the following locations:

- `contracts/z_imburse_escrow/src/library_methods/entitlements.nr`:
  - `give_entitlement()` implements (1).
- `contracts/z_imburse_escrow/src/types/entitlement_note.nr`:
  - `check_and_emit_recurring_nullifier()` implements (2).
  - `check_nullifier_txe()` implements (2).

#### Impact

1. In the usage of the protocol, the only usage of this check is to prevent an admin from giving a duplicate entitlement. Given that no safety is provided from this check, it is unnecessary.
2. Usage of this check gives the developers and/or readers of the contracts false assurances surrounding the existence of a nullifier for a given Note.

#### Recommendation

1. Document this check clearly, and note on the function that it is not fully constrained. Consider moving this check to the front-end to improve implementation clarity.
2. Although one may use `prove_nullifier_non_inclusion()`, this limits the ability for a transaction to be included in a block. Additionally, the kernel circuit prevents duplicate nullifiers already, so the logic of the contract should just assume that a nullifier has not been created and emit the nullifier after processing.

**Developer Response** The developers have been notified of the issue but have yet to respond with acknowledgement or fixes.

### 4.1.12 V-ZIM-VUL-012: Incorrect date computations

| | | | |
|---|---|---|---|
| **Severity** | Low | **Commit** | 61f276f |
| **Type** | Logic Error | **Status** | Acknowledged |
| **File(s)** | `circuits/date_parser/src/lib.nr` | | |
| **Location(s)** | to_unix() | | |
| **Confirmed Fix At** | N/A | | |

z-imburse parses a formatted date into a unix timestamp for use as a recurring nullifier, and to ensure spot timestamps are used in a valid timeframe. At the time of the audit, some errors were present in the library leading to an off-by-one error in the days.

```
#[test]
fn test_main() {
    let date_string = "1 Sep 2024 23:22:12 +0400 ".as_bytes(); // padded 1
    // its off a lil bit
    // should be 1725247332
    // should be 1725333732
    let date = parse_date_string(date_string);
    let unix = to_unix(date);
    println(unix);
}
```

**Snippet 4.8:** Snippet from `circuits/date_parser/src/lib.nr`.

Note that the above comment is off by exactly 86400, or one day's worth of seconds. However, the comment is also incorrect. The correct unix timestamp of "1 Sep 2024 23:22:12 +0400" is `1725218532`, which is exactly 8-hours offset from `1725247332`.

```
from datetime import datetime

# The date string
date_string = "1 Sep 2024 23:22:12 +0400"

# Define the format string
format_string = "%d %b %Y %H:%M:%S %z"

# Parse the date string
parsed_date = datetime.strptime(date_string, format_string)

# Print the parsed date
print("Parsed Date:", parsed_date)
print("Unix Timestamp:", int(parsed_date.timestamp()))

# OUTPUT
# parsed Date: 2024-09-01 23:22:12+04:00
# Unix Timestamp: 1725218532
```

**Snippet 4.9:** Python script evaluating the unix timestamp of "1 Sep 2024 23:22:12 +0400".

There are two errors causing these discrepancies:

1. The first error is an off-by-one error in `to_unix()` and `to_unix_month()`. In the below snippet, the day of the month (`datetime.day`) is added to the `days_in_current_year`,

which is supposed to represent the number of complete days which have occurred in the current year. Note, however, that on the 1st day of a month, 0 complete days have occurred in that month. Instead, `datetime.day - 1` should be used.

```
days_in_current_year += (datetime.day + datetime.add_leap_day as u8) as u32;
```

**Snippet 4.10:** Computation including the day of the month into the days which have passed in the current year.

2. The second error is reversed handling of the offset. An offset of `-0500` means the timestamp is 5 hours *behind* UTC. To restore the UTC timezone, one must then *add* 5 hours. However, `parse_offset()` returns `subtract=true` when a negative offset is provided.

**Impact** The incorrect timestamp is returned, which may lead to invalid reimbursements being approved around the beginning of each month. Additionally, users may have their e-mails wrongfully denied.

**Recommendation** Fix the off-by-one error and flip the handling of the sign for timezone offsets.

Consider adding a larger test suite. This should include several random cases, and edge cases including:

1. The 0 timestamp.
2. The maximum 32-bit timestamp.
3. Dates before, on, and after leap day in leap years (flexing all possible cases of the year being a multiple of four, one hundred, and four-hundred).
4. Dates before and after leap day in non-leap years.

Additionally, consider adding a large number of random test cases. The Veridise analysts validated the above two fixes on one hundred random timestamps using the below python script to generate test cases.

```python
import random
import pytz
from datetime import datetime, timezone

def generate_noir_tests(num_tests):
    # Initialize Noir test script
    noir_code = []
    noir_code.append('#[test]\nfn test_random_date_string_to_unix() {\n')

    for i in range(num_tests):
        # Generate a random 32-bit timestamp
        timestamp = random.randint(0, 2**32 - 1)

        # Get a random timezone
        tz = set(pytz.all_timezones_set)
        tz = list(tz)
        rand_zone_name = random.choice(pytz.all_timezones)
        rand_zone = pytz.timezone(rand_zone_name)

```

```
        # make date-string
        date_string = datetime.fromtimestamp(timestamp, tz=rand_zone).strftime("%-d %
    b %Y %H:%M:%S %z")
        if len(date_string) < 26:
            date_string = date_string + "\\0"

        # Generate Noir test code
        noir_code.append(f'    let date_string_{i} = "{date_string}".as_bytes();')
        noir_code.append(f'    let date_{i} = parse_date_string(date_string_{i});')
        noir_code.append(f'    let unix_{i} = to_unix(date_{i});')

        noir_code.append(f'    assert(unix_{i} == {timestamp}, unix_{i});\n')

    # Close the function
    noir_code.append('}\n')

    return "\n".join(noir_code)

# Number of tests to generate
num_tests = 100

# Generate and print Noir tests
if __name__ == "__main__":
    noir_test_code = generate_noir_tests(num_tests)
    print(noir_test_code)
```

**Developer Response** The developers have acknowledged the issue and won't provide a fix as they no longer will use the `parse_date_string()` function. They will later fix the issue and utilize the Veridise provided test script when they separate out date parsing into its own project.

### 4.1.13 V-ZIM-VUL-013: Missing/incorrect documentation

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 61f276f |
| **Type** | Maintainability | **Status** | Partially Fixed |
| **File(s)** | See issue description | | |
| **Location(s)** | See issue description | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/24` | | |

The following items have incorrect documentation or are lacking documentation:

1. `contracts/generators/src/lib.nr`:
   a) The generators should document what they are used for and where they were generated from.
2. `contracts/z_imburse_escrow/src/library_methods/hashing.nr`:
   a) The documentation comment on `finish_hash()` mentions the `destination` instead of the `date` when describing `hash_state_before_date`.
3. `contracts/z_imburse_escrow/src/main.nr`:
   a) The functions related to United reimbursements such as `reimburse_united_spot()` and `united_deferred_verification()` should be marked as WIP or not to be used for production.
4. `contracts/z_imburse_escrow/src/verifiers.nr`:
   a) In `verify_linode()`, the `billed_amount` is multiplied by `10000`. This should be documented and additionally should instead be a constant defined variable.
5. `contracts/z_imburse_registry/src/main.nr`
   a) `register_rescrow()`: `exected_escrow_address` should be `expected`.
   b) In `register_dkim()`, `register_dkim_bulk()` and `check_and_register_participant()`, the admin checking error message should specify the sender is not the admin.
6. `circuits/date_parser/src/lib.nr`:
   a) The todo comment above `compute_leap_years()` is out of date.
7. `circuits/zimburse_verifiers/src/linode/utils.nr`:
   a) In `extract_billed_amount()` the comment on the declaration of `commas` is incomplete.
8. Document why the `compressed_string` library was copied from the standard library into this project.

**Impact** Missing or incorrect documentation may lead to misuse of the library down the road.

**Recommendation** Fix the documentation for the above functions.

**Developer Response** The developers acknowledge the issue and implemented the recommendations except for those for (2).

### 4.1.14 V-ZIM-VUL-014: The fields of an EntitlementNote should be verified

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 61f276f |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `contracts/z_imburse_escrow/src/types/entitlement_note.nr` | | |
| **Location(s)** | is_spot() | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/29` | | |

The `new()` function simply takes in the values of the `EntitlementNote` fields. However, other code in `EntitlementNote` assumes certain relationships between field values which is not actually verified in `new()`

1. The `spot` variable is passed to `new()`. However, the `is_spot()` function is defined as below and may diverge from the passed in value.

```
fn is_spot(self) -> bool {
    self.date_start != 0
}
```

**Snippet 4.11:** Snippet from `contracts/z_imburse_escrow/src/types/entitlement_note.nr:is_spot()`

2. The `destination` is not enforced to be `ZERO_DESTINATION` when `!spot`.

#### Impact

1. Although `is_spot()` is not utilized, this definition of spot and the variable passed to `new()` may diverge.
2. The destination may be set even for a recurring entitlement.

#### Recommendation

1. Do one of the following:
   a) Remove `is_spot()` as it is unused.
   b) Perform the check that `spot == (date_start != 0)` in the `new()` function.
   c) Simply return the `spot` variable from `is_spot()` and specify to callers of `new()` the required behavior between `spot` and `date_start`.
2. Validate the value of `destintation` based on the value of `spot`.

**Developer Response** The developers acknowledge the issue and have removed `is_spot()`. Additionally, they validate in `new()` the values of `date_end`, `date_start`, and `destination` when `spot` is true.

### 4.1.15 V-ZIM-VUL-015: Unused/duplicate program constructs

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 61f276f |
| **Type** | Maintainability | **Status** | Partially Fixed |
| **File(s)** | See issue description | | |
| **Location(s)** | See issue description | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/20`, 4450f37 | | |

**Description** The following program constructs are unused or duplicate functionality:

1. `circuits/date_parser/src/lib.nr`:
   a) `to_unix()`: This function's implementation is nearly identical to `to_unix_month()`, except for the final inclusion of the timestamp/offset.
2. `circuits/zimburse_verifiers/src/utils.nr`:
   a) `constrain_header_field()` reimplements functionality from `zkemail.nr`.
3. `contracts/z_imburse_escrow/src/types/escrow_definition.nr`:
   a) `TITLE_SERIALIZED_LENGTH` would better be defined as `(TITLE_LENGTH + 30) / 31`.
   b) `ESCROW_DEFINITION_LENGTH` would better be defined as `TITLE_SERIALIZED_LENGTH + 3`.
4. `contracts/z_imburse_escrow/src/verifiers.nr`:
   a) The `verifier_ids` are also defined in `contracts/z_imburse_registry/src/verifiers.nr`.
5. `contracts/z_imburse_registry/src/main.nr`:
   a) The loop inside of `register_dkim_bulk()` and the `constructor()` should be abstracted out to its own function, as they are the exact same.
   b) The `register_dkim()` function contains the same logic as the loop bodies mentioned above, save the admin check. This functionality should also be abstracted away so all 3 functions use the same key registration logic.

**Impact** These constructs may become out of sync with the rest of the project, leading to errors if used in the future.

**Developer Response** The developers acknowledge the issue and have provided de-duplication for items (3) - (5).

### 4.1.16 V-ZIM-VUL-016: EntitlementNote.eq missing checks

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 61f276f |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `contracts/z_imburse_escrow/src/types/entitlement_note.nr` | | |
| **Location(s)** | <EntitlementNote as Eq>::eq() | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/21`, aeeac49 | | |

The `eq()` implementation for `EntitlementNote` fails to check for equality of the `start_date`, `stop_date`, and `destination` fields.

```
impl Eq for EntitlementNote {
    fn eq(self, other: Self) -> bool {
        (self.recipient == other.recipient)
            & (self.max_value == other.max_value)
            & (self.randomness == other.randomness)
            & (self.verifier_id == other.verifier_id)
    }
}
```

**Snippet 4.12:** Implementation of the `Eq` trait

**Impact** Future implementations may treat unequal entitlement notes as equal.

**Recommendation** Check all fields for equality.

**Developer Response** The developers acknowledge the issue and now check for equality for the missing fields. The fix commit augments the equality function for EntitlementNote to perform checks on missing fields: `date_end`, `date_start`, `destination`, and `spot`

### 4.1.17 V-ZIM-VUL-017: check_nullifier_txe should be unconstrained

| | | | |
|---|---|---|---|
| **Severity** | Info | **Commit** | 61f276f |
| **Type** | Maintainability | **Status** | Fixed |
| **File(s)** | `contracts/z_imburse_escrow/src/types/entitlement_note.nr` | | |
| **Location(s)** | check_nullifier_txe() | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/23` | | |

The `check_nullifier_txe()` function calls `compute_nullifier_without_context()` and `check_nullifier_exists()`. Both of these are unconstrained functions. Therefore `check_nullifier_txe()` should also be marked as unconstrained.

**Impact** Callers of this may mistakenly believe that the returned nullifier is constrained to be correct and does not exist.

**Recommendation** Mark `check_nullifier_txe()` as unconstrained. Additionally, any other functions that call `compute_nullifier_without_context()` (such as the normal `check_nullifier`) should also be marked as unconstrained.

**Developer Response** We removed this function as part of the fix for V-ZIM-VUL-007.

### 4.1.18 V-ZIM-VUL-018: Optimization opportunities

| | | | |
|---|---|---|---|
| **Severity** | Info | **Commit** | 61f276f |
| **Type** | Optimization | **Status** | Fixed |
| **File(s)** | See issue description | | |
| **Location(s)** | See issue description | | |
| **Confirmed Fix At** | `https://github.com/Mach-34/z-imburse/pull/27` | | |

In the following locations, the auditors identified missed opportunities for constraint optimizations:

- `contracts/z_imburse_registry/src/main.nr`:
  - In `get_participant_escrows()` and `get_managed_escrows()`, the if statement inside of the loop is unnecessary as the loop only iterates through `notes.len()`.

**Impact** Circuits may perform poorly or gas may be wasted, costing users extra time and funds.

**Recommendation** Perform the optimizations.

**Developer Response** The developers removed the unnecessary if-statement.

# Glossary

**Aztec Network** An Ethereum Layer-2 (zk-rollup) where smart contracts that compile to zero-knowledge circuits enable computations over private state. See the official website at https://aztec.network/. 1, 4

**DKIM** DomainKeys Identified Mail (DKIM) Signatures are commonly used to authenticate e-mail senders. See IETF RFC 6376 to learn more (https://datatracker.ietf.org/doc/html/rfc6376). 1

**Noir** A DSL for writing zero-knowledge circuits in a high-level, rust-like syntax. See https://noir-lang.org to learn more. 4

**smart contract** A self-executing contract with the terms directly written into code. Hosted on a blockchain, it automatically enforces and executes the terms of an agreement between buyer and seller. Smart contracts are transparent, tamper-proof, and eliminate the need for intermediaries, making transactions more efficient and secure. 1, 33

**zero-knowledge circuit** A cryptographic construct that allows a prover to demonstrate to a verifier that a certain statement is true, without revealing any specific information about the statement itself. See https://en.wikipedia.org/wiki/Zero-knowledge_proof for more. 1, 33