ZK Email Noir

Date	December 2024
Auditors	Heiko Fisch, George Kobakhidze, Rai Yang

1 Executive Summary

This report presents the results of our engagement with **Aztec** and **Mach 34** to review **ZKEmail.nr**, a port of the ZK Email Circom library to the Noir language by Aztec.

The time-boxed best-effort review was conducted over three weeks, from **November 2024** to **December 2024**, by **Heiko Fisch**, **George Kobakhidze**, and **Rai Yang**.

The ZK Email library allows its users to create and verify proofs of emails and some of their properties. The main source of verification comes from the DKIM signatures provided in email headers. These signatures and their parameters verify which header fields in the provided email are signed by the DKIM server of the DKIM-associated domain of the sender's email server.

The specific version of the library that is relevant for this audit is in its initial state and currently doesn't support all email functionality. Overall, this library doesn't strive for completeness (all true things can be verified) but focuses on the obligatory soundness (what can be verified must be true). For example, some characters that are allowed and even frequently used in practice in the email address fields per email protocol standards, such as the `+` sign, are **not** permitted in this library.

This library offers different and isolated functions that developers can utilize as they see fit. Specific header checks, email body verification (partial and complete), text content manipulation, and other functions are available via individual circuits for each.

Careful preparation of email data needs to be done prior to utilizing this library. Indeed, the Noir circuits require the input of email headers to be already cleaned of all DKIM irrelevant fields (i.e. those that are not covered by the DKIM signature), and the content must be canonicalized in the `relaxed` mode, as that is the only DKIM mode that is currently supported.

Many issues found in the audit are of the "edge case" variety. With unstructured data such as email subjects, to/from fields, and the email body content, properly configuring what is an acceptable boundary and what is not can be extremely difficult. Some of the issues in this audit will illustrate how clever attackers can abuse the lack of structure in this type of data along with insufficiently tight constraints to prove wrong statements. The audit report also illustrates some issues that call for more constraints so that less careful users of this library don't accidentally open up attack vectors to verify false information. Moreover, some auxiliary libraries or code sections borrowed from those libraries are not of the latest versions, which opens up some security issues as well.

Overall, the complexity of constraining unstructured data and the current state of the repository make opportunities for issues plentiful. While we tried to cover the most important areas, especially as far as soundness is concerned, the nature and amount of issues found make it likely that there are more problems that couldn't be uncovered in this time-constrained review.

2 Scope

Our review focused on the commit hash 2f81196ac04cd11a921816b52b432bb8c5eb3150. The list of files in scope can be found in the Appendix.

Only Noir code was in scope but not all of it:

- Tests (mostly) and examples were not reviewed, due to the limited time available.
- The file `macros.nr` is work-in-progress and was therefore excluded by the client.
- The circuits were reviewed in isolation, i.e., the question whether they're compatible with other parts of the stack was not investigated, nor were these other parts themselves.

It is also noteworthy that other Noir libraries which zkemail.nr uses are in active development, change frequently, and may therefore have a higher risk of containing bugs than is the case in more mature ecosystems. These libraries were also out of scope.

2.1 Objectives

Together with the client team, we identified the following priorities for our review:

1. Correctness of the implementation, consistent with the intended functionality and without unintended edge cases.
2. Circuits are sound.
3. Circuits are sufficiently complete to accommodate the examples provided.
4. Email protocol standards, such as DKIM standards, are taken into account.

3 System Overview

The ZK Email Noir library offers its users a set of modular functions to verify specific properties of email and, in some cases, any content.

3.1 lib.nr

This is essentially the entry point circuit of the library. It contains the `standard_outputs` function that outputs Pedersen hashes of DKIM public key (the root of trust) and of DKIM signature (a suggested email nullifier by the ZK Email team), critical components of any email verification scheme. Similarly, there is a bare-bones function `verify_email` present that can be used as an example of a simple email-verifying function to build upon.

3.2 dkim.nr

The core of the ZK Email library, the `dkim.nr` file contains circuits that verify the DKIM signature present in the header against the header content. DKIM is the root of trust of the present circuits, and it is responsible for the veracity of all provided data.

3.3 headers/mod.nr

The `headers/mod.nr` file provides functionality to constrain specific headers as they are provided to the circuits. The functionality is currently abstracted away to headers generally, without having a function for each header field such as "to", "from", "subject", and so on. The one exception is

a function that constrains the header field normally but also returns the last occurrence of a left angle bracket, a functionality required for constraining email address fields properly.

3.4 headers/email_address.nr

While header field names have general functions, the email address field values have their own circuits for verification. These circuits verify that the signed email address values contain only correct characters. However, it should be noted that this library errs on the side of caution and deems only a subset of email-standard-approved characters to be legal in these circuits. As such, somewhat common characters such as “+” signs are not allowed to be present in email addresses in these circuits.

3.5 body_hash.nr

Similarly as for email address, there are circuits just for retrieving and verifying the body hash present in the header. Together with the general SHA functions available in Aztec Noir libraries as well as the `partial_hash.nr` functions, the `body_hash.nr` circuits can prove and verify either portions of or the full email body in emails submitted to the circuits.

3.6 masking.nr

This file contains a small function that masks parts of the provided text and returns the result. It does not perform any special constraining and is not specific to email.

3.7 remove_soft_line_breaks.nr

This is a content manipulation circuit that offers the removal of soft line breaks, a common email body editing functionality. It can be used to verify that the provided text is indeed the correct result of removing soft line breaks from an original if both are submitted to a circuit.

3.8 partial_hash.nr

A general-purpose circuit not specific to email that provides its users functionality to partially hash messages and retrieve hash state mid-process. This could be useful if there is a small section of content that needs to be verified in a large text. An implementing developer may construct partial hash states of the text and only submit to the circuit the raw text of the section in question, later combining its hash state with others to achieve the final resulting hash that can be verified against the root of trust. Like a few above, this function can be used with any data content, not just email.

4 Security Specification

As this repository is that of a library as opposed to a live system that will hold funds, the security of the system is reliant on the particulars of email signatures, in our case the DKIM signature, and the context in which this library is used.

Indeed, essentially the entirety of email verification hinges on the DKIM signature of submitted emails being constructed properly, safely, and securely. Should the DKIM server or the DNS entry pointing to that DKIM server of a domain be compromised, this library may potentially be used to verify false information. Additionally, even if DKIM signature and all properties are correct, if the signed data is incorrect in the first place, the ZK Email library would simply verify that incorrect information. For example, improperly secured email servers may be tricked into verifying the wrong sender domain upon email receipt, therefore potentially signing off on a phishing email. The circuits provided in this library would simply verify that the DKIM has indeed signed off on the phishing email, but would provide no information on true validity of the data.

The implementation of the library itself may also affect the security of a system utilizing it. As is noted in a few issues, directly using some of the circuits present in this library is not enough to ensure that the circuits produce and verify the correct information. The data provided needs to be of certain sizes, formats, and otherwise properly configured prior to being submitted to the circuits.

Ultimately, the team deploying the ZK Email library into their system needs to both carefully review the security assumptions around original data veracity, such as the source of their emails and their signatures, and meticulously construct the entry-point circuits that will call ZK Email functions downstream so that only appropriate data enters. Additional constraints may need to be introduced on the data depending on the use cases of the implementation team.

5 Findings

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

5.1 Header Field Extraction Can Be Fooled by Passing a Simple-Canonicalized Header Critical

Description

Since emails are frequently modified in transit – in ways that are insignificant semantically but would impact signature verification – a canonicalization algorithm is applied to header and body (separately) before signing and, hence, signature verification. RFC 6376 defines two such algorithms: simple and relaxed. Currently, the codebase supports only the relaxed variant; for instance, extracting the DKIM signature header fails if the header field name after canonicalization is “DKIM-Signature” instead of “dkim-signature”, where the latter is what relaxed canonicalization invariably produces, while simple canonicalization lets the capital letters pass unmodified. While the restriction to relaxed canonicalization is not properly documented, we assume in this report that this is known and – at least for the time being – acceptable.

Nevertheless, it still shouldn’t be possible to prove something wrong when a simple-canonicalized and properly signed header is passed into the circuit. That is currently possible, though. Specifically, with relaxed header canonicalization, header fields are separated by CRLF, and CRLF can’t occur *within* a header field. In other words, if a sequence is directly preceded by CRLF (or starts at the very beginning of the entire header) and is directly followed by CRLF (or ends at the very end of the entire header) and doesn’t contain CRLF, then this sequence covers exactly one header field. That’s the algorithm for header field extraction implemented in `constrain_header_field` and `constrain_header_field_detect_last_angle_bracket`.

Simple canonicalization, on the other hand, allows the header field (body) to be split into multiple lines. Specifically, if a CRLF is followed by whitespace (space or horizontal tab), then the whitespace and what comes after it belong to the previous header field. In other words, a CRLF that’s followed by whitespace does not mark the end of a header field.

Hence, the current algorithm to extract a header field allows the following exploit: Provide a properly signed, simple-canonicalized header, so signature verification will pass. Then, by choosing a sequence that marks a header field up until the first CRLF+whitespace, extract only the start of the header field as the *entire* header field.

Extracting the end or a middle part of a header field as a full field is, in theory, possible too with this approach, but it might be more difficult to exploit it *meaningfully* because the extracted part would (1) start with whitespace, (2) have to contain a colon, and (3) all characters up to this colon,

including the whitespace at the beginning, would become the header field name. While this is technically allowed at the moment (see issue 5.3), the result is not a valid header field name because of the leading whitespace and possibly other characters before the colon that may not occur in header field names.

As a concrete exploit example, with the following occurring in a header that's canonicalized with the "simple" algorithm and then signed, it would be possible to prove that this email was sent from `alice@example.net`, instead of `bob@example.org`:

```
from: ":alice@example.net>"
 <bob.example.org>
```

Recommendation

First and foremost, `constrain_header_field` and `constrain_header_field_detect_last_angle_bracket` must not accept CRLF as a delimiter between header fields when it is followed by whitespace.

Regarding the example exploit above, it should be noted that a more stringent rule in `get_email_address` would've prevented that too: If no left angle bracket is present in the header field, the email sequence has to start right after the colon that follows the header field name and its end must also be the end of the header field. (And similarly, although that has nothing to do with the example above: If we do find a left angle bracket, we should not only verify that the email address sequence starts right after it but also that it is followed by a right angle bracket and the header field ends after that.) Likewise, in the event that other header value reading functions are introduced, such as for the "Subject" header, there should be similar constraints that ensure the header value begins right after the header name ends, and the header value ends when the header ends.

5.2 First Characters of Claimed Header Field Value Unchecked for CRLF Major

Description

A header field consists of the header field name, followed by a colon, and then the header field value, also called header field body. For a header that has been canonicalized with the "relaxed" algorithm, the header fields are separated by CRLF, and header fields themselves cannot contain CR or LF.

The following piece of code from the function `constrain_header_field` is supposed to verify that the claimed header field value contains no CR.

lib/src/headers/mod.nr:L62-L73

```
// check the header field is uninterrupted
let start_index = header_field_sequence.index + HEADER_FIELD_NAME_LENGTH + 1;
for i in (HEADER_FIELD_NAME_LENGTH + 1)..MAX_HEADER_FIELD_LENGTH {
    // is it safe enough to cut this constraint cost in half by not checking lf? i think so
    let index = start_index + i;
    if (index < header_field_sequence.index + header_field_sequence.length) {
        assert(
            header.get_unchecked(index) != CR,
            "Header field must not contain newlines"
        );
    }
}
```

However, as `start_index` is initialized to `header_field_sequence.index + HEADER_FIELD_NAME_LENGTH + 1` and the loop index `i` starts at `HEADER_FIELD_NAME_LENGTH + 1`, the first `index` we actually check to be unequal to CR is `header_field_sequence.index + 2 * HEADER_FIELD_NAME_LENGTH + 2` – whereas it *should* be `header_field_sequence.index + HEADER_FIELD_NAME_LENGTH + 1`. In other words, some indices remain unchecked.

The same problem occurs in `constrain_header_field_detect_last_angle_bracket`:

lib/src/headers/mod.nr:L123-L139

```
// check the header field is uninterrupted
let mut last_angle_bracket = 0;
let start_index = header_field_sequence.index + HEADER_FIELD_NAME_LENGTH + 1;
for i in (HEADER_FIELD_NAME_LENGTH + 1)..MAX_HEADER_FIELD_LENGTH {
    // is it safe enough to cut this constraint cost in half by not checking lf? i think so
    let index = start_index + i;
    if (index < header_field_sequence.index + header_field_sequence.length) {
        let byte = header.get_unchecked(index);
        assert(
            byte != CR,
            "Header field must not contain newlines"
        );
        if byte == 0x3c {
            last_angle_bracket = index;
        }
    }
}
```

An attacker could use this to join two header fields and claim that it is one. While there are limitations to exploiting this in practice because the number of unchecked positions is typically small, there is still enough room for attacks. Consider, for example, the following excerpt from a relaxed-canonicalized header:

```
subject:yes
to:noone@example.org
```

Due to the bug we described, it is possible to prove that the subject of this email contains or even starts with "no", which could clearly have severe implications.

Recommendation

In both functions, initialize `start_index` to `header_field_sequence.index`. This avoids the "double addition" of `HEADER_FIELD_NAME_LENGTH + 1`.

5.3 Missing Validation of Characters in `header_field_name` Major

Description

The `constrain_header_field` and `constrain_header_field_detect_last_angle_bracket` functions in `mod.nr` take as input the (signed) header, a sequence that is supposed to represent a complete header field (name, colon, and value), and the header field name of the header field we want to extract. The main task of these two functions is to make sure that the sequence doesn't cover *less* than an entire header field, doesn't cover *more* than one header field, and starts with `header_field_name`, followed by a colon.

These functions trust that `header_field_name` is indeed a valid header field name and don't perform any kind of validation on it. However, if `header_field_name` contains, for example, CRLF, then the sequence can spread over more than one header field. If it contains a colon, then part of the

actual header field body is attributed to the header field name. For example, if the “Subject” header field is `subject:foo:bar` (i.e., name is “subject” and value is “foo:bar”), then it would be possible to prove that there is a header field with name “subject:foo” and value “bar”.

Recommendation

Both functions have to verify that `header_field_name` contains only characters that are allowed in header field names, which is the range 0x21-0x7E (inclusive), but excluding the colon (0x3A).

Theoretically, it should be sufficient to assert that `header_field_name` doesn’t contain a colon. If we trust the signer to only sign RFC-compliant header fields, then any other character that is invalid in field names would have to come after the colon that marks the end of the field name. Hence, if `header_field_name` contains an invalid character at all, it also contains a colon. However, trying to be “too smart” with these things can be dangerous, and we actually recommend the more cautious approach from the previous paragraph.

It also makes sense to statically assert `HEADER_FIELD_NAME_LENGTH != 0`.

5.4 Missing Validation of Header Field Sequence Length Major

Description

The `constrain_header_field` function verifies that the claimed header field sequence is completely contained in the header:

lib/src/headers/mod.nr:L93-L95

```
// check the range of the sequence is within the header (so we can use get_unchecked)
let end_index = header_field_sequence.index + header_field_sequence.length;
assert(end_index <= header.len(), "Header field out of bounds of header");
```

There is, however, no check that the header field name and the following colon fit into the sequence. Nor is there a check that the sequence length is at most `MAX_HEADER_FIELD_LENGTH`, which is a generic parameter for the maximum length of the header field.

Regarding the first missing check, it would be possible to prove a header field that is shorter than `header_field_name`. (See also issue 5.3 in this context.)

The second missing check means that the part of a sequence that extends beyond `MAX_HEADER_FIELD_LENGTH` won’t be verified to not contain CR, so the sequence could span more than one actual header field.

lib/src/headers/mod.nr:L64-L73

```
for i in (HEADER_FIELD_NAME_LENGTH + 1)..MAX_HEADER_FIELD_LENGTH {
    // is it safe enough to cut this constraint cost in half by not checking lf? i think so
    let index = start_index + i;
    if (index < header_field_sequence.index + header_field_sequence.length) {
        assert(
            header.get_unchecked(index) != CR,
            "Header field must not contain newlines"
        );
    }
}
```

The same checks are also missing from the function `constrain_header_field_detect_last_angle_bracket`. Here, the part of a sequence after `MAX_HEADER_FIELD_LENGTH` remains not only unchecked for CR, but also for “<”, so the function might not actually find the last left angle bracket. This means that it could be possible in such a case to prove a wrong email address.

lib/src/headers/mod.nr:L126-L138

```
for i in (HEADER_FIELD_NAME_LENGTH + 1)..MAX_HEADER_FIELD_LENGTH {
    // is it safe enough to cut this constraint cost in half by not checking lf? i think so
    let index = start_index + i;
    if (index < header_field_sequence.index + header_field_sequence.length) {
        let byte = header.get_unchecked(index);
        assert(
            byte != CR,
            "Header field must not contain newlines"
        );
        if byte == 0x3c {
            last_angle_bracket = index;
        }
    }
```

Recommendation

To both functions, add the following two assertions:

```
assert(
    HEADER_FIELD_NAME_LENGTH < header_field_sequence.length,
    "Header field sequence too short to contain name and colon"
);
assert(
    header_field_sequence.length <= MAX_HEADER_FIELD_LENGTH,
    "Header field sequence longer than maximum length"
);
```

5.5 `partial_sha256_var_start` With Message Sizes Not Multiple of `BLOCK_SIZE` Gives Same `h` State for Different Data Major

Description

The `partial_sha256_var_start` function is designed to hash the initial part of a message and return a hash state `h`. This hash state can then be used with `partial_sha256_var_interstitial` to hash additional parts of the message or with `partial_sha256_var_end` to complete the hash of the entire message.

However, if the size of the hashed part of the message given to `partial_sha256_var_start` is not a multiple of `BLOCK_SIZE` (64 bytes in the provided circuits), the function will not hash the remaining data of the message that is less than the block size, resulting in an incorrect hash for this data. This incorrect hash may then be used with `partial_sha256_var_interstitial` and `partial_sha256_var_end,` leading to an incorrect final hash that will not match the signed body hash found in the DKIM header, causing the circuit validation to fail.

Moreover, any other message that has the same data in the fist part that fits into multiples of `BLOCK_SIZE`, and has remaining different data that is less than but fits in one block of `BLOCK_SIZE` will also produce the same hash. This will happen regardless of what other data is in the remaining part. In

other words, providing some message that is 64 bytes called `data1` and another message `data2=data1+"previous content is false"` to `partial_sha256_var_start` will give the same hash state.

In summary, by not enforcing the size of hashed data to be a multiple of `BLOCK_SIZE`, the circuits may not only produce a wrong hash state for the data but can even end up falsely verifying that different message data produce the same hash state.

Examples

lib/src/partial_hash.nr:L105-L110

```
    if (msg_start < N) & (msg_byte_ptr == BLOCK_SIZE) {
        h = sha256_compression(msg_u8_to_u32(msg_block), h);
    }
}

h
```

examples/partial_hash/src/main.nr:L44-L52

```
let signed_body_hash = get_body_hash(header, dkim_header_sequence, body_hash_index);

// finish the partial hash
let computed_body_hash = partial_sha256_var_end(partial_body_hash, body.storage(), body.len() as u64, partial_body_real_length);

// check the body hashes match
assert(
    signed_body_hash == computed_body_hash, "Sha256 hash computed over body does not match DKIM-signed header"
);
```

Recommendation

Constrain the size of the message to be the multiple of `BLOCK_SIZE` in the `partial_sha256_var_start` function.

5.6 `partial_sha256_var_interstitial` May Give the Same Hash State `h` for Different Data Objects if They Are Smaller Than `message_size` Major

Description

`partial_sha256_var_interstitial` function is used with a partially computed sha256 hash created with `partial_sha256_var_start` and a partial message to generate an interstate hash. However, when the input message's size (`N`) is less than the message size parameter (`message_size`) and the size is not a multiple of `BLOCK_SIZE` (64 bytes), the last unfilled block of the message would not be hashed. In other words, even though the expected amount of data to be hashed is `message_size`, only `BLOCK_SIZE*(N/BLOCK_SIZE)` amount of data will be hashed. Consequently, the remaining data is ignored, and the returned hash state `h` will be the same for different data objects as long as they share the initial `BLOCK_SIZE*(N/BLOCK_SIZE)` bytes.

In order to avoid this, the circuit does an assert on `message_size` being a multiple of `BLOCK_SIZE`:

lib/src/partial_hash.nr:L124

```
assert(message_size % BLOCK_SIZE == 0, "Message size must be a multiple of the block size");
```

However, as mentioned above, when array size `N` is smaller than `message_size`, this doesn't work as `N` would be different from `message_size` so it wouldn't be bound by the same constraints.

The result is that different data objects that share some initial bytes would return the same hash state `h` when submitted to `partial_sha256_var_interstitial` if:

- they are smaller than `message_size`
- are not a multiple of `BLOCK_SIZE`
- and fill the same amount of `BLOCK_SIZE` byte blocks.

For example, the below code validates different data structure hashes:

```
fn test_partial_hash_interstitial_var_different_arrays() {
    let mut data0 = [0; 64]; ////keeping these two here to make the original hash
    let mut data2 = [0; 64];
    let mut data1_original = [0; 64]; // second block not used. but keeping array to <128
    let mut data1_custom = [0; 100]; // make array size less than 128
    for i in 0..data0.len() {
        data0[i] = DATA[i];
        data1_original[i] = DATA[64 + i];
        data1_custom[i] = DATA[64 + i];
        data2[i] = DATA[128 + i];
    }
    data1_custom[90]=123; // and also actually fill some data in the second block for the modified data
    let pre_hash = partial_sha256_var_start(data0);
    let interstitial_hash = partial_sha256_var_interstitial(pre_hash, data1_original, 128); // increasing message_size== 128 so the developer may expect that 128 bytes will
    let hash = partial_sha256_var_end(
        interstitial_hash,
        data2,
        data2.len() as u64,
        DATA.len() as u64,
    );
    //now do the second hash
    let custom_interstitial_hash = partial_sha256_var_interstitial(pre_hash, data1_custom, 192); // increasing message_size== 192 so the developer may expect that 192 bytes
    let custom_hash = partial_sha256_var_end(
        custom_interstitial_hash,
        data2,
        data2.len() as u64,
        DATA.len() as u64,
    );
    let correct_hash = std::hash::sha256_var(DATA, DATA.len() as u64);
    assert_eq(hash, correct_hash);
    //assert equality
    assert_eq(hash, custom_hash);
}
```

Being able to validate the same hash state for different data objects with different intended message sizes to be hashed may cause significant soundness issues. This is especially significant for a more general-purpose library for partial hashing that may be utilized for more than just email bodies.

Examples

lib/src/partial_hash.nr:L123-L156

```
pub fn partial_sha256_var_interstitial<let N: u32>(mut h: [u32; 8], msg: [u8; N], message_size: u32) -> [u32; 8] {
    assert(message_size % BLOCK_SIZE == 0, "Message size must be a multiple of the block size");
    let num_blocks = N / BLOCK_SIZE;
    let mut msg_block: [u8; BLOCK_SIZE] = [0; BLOCK_SIZE];
    let mut msg_byte_ptr = 0; // Pointer into msg_block

    for i in 0..num_blocks {
        let msg_start = BLOCK_SIZE * i;
        let (new_msg_block, new_msg_byte_ptr) = unsafe {
            build_msg_block_iter(msg, N, msg_start)
        };
        if msg_start < N {
            msg_block = new_msg_block;
        }

        if !is_unconstrained() {
            // Verify the block we are compressing was appropriately constructed
            let new_msg_byte_ptr = verify_msg_block(msg, N, msg_block, msg_start);
            if msg_start < N {
                msg_byte_ptr = new_msg_byte_ptr;
            }
        } else if msg_start < N {
            msg_byte_ptr = new_msg_byte_ptr;
        }

        // If the block is filled, compress it.
        // An un-filled block is handled after this loop.
        if (msg_start < N) & (msg_byte_ptr == BLOCK_SIZE) & (msg_start < message_size){
            h = sha256_compression(msg_u8_to_u32(msg_block), h);
        }
    }

    h
}
```

examples/partial_hash/src/main.nr:L44-L52

```
let signed_body_hash = get_body_hash(header, dkim_header_sequence, body_hash_index);

// finish the partial hash
let computed_body_hash = partial_sha256_var_end(partial_body_hash, body.storage(), body.len() as u64, partial_body_real_length);

// check the body hashes match
assert(
    signed_body_hash == computed_body_hash, "Sha256 hash computed over body does not match DKIM-signed header"
);
```

Recommendation

Add a constraint for the input message's size `N` to be equal to or greater than `message_size`.

5.7 `partial_sha256_var_start` Messages for Size Smaller Than `BLOCK_SIZE` Returns the Same Hash State `h` for All Data Major

Description

As a special case of the issue described in issue 5.5 , when an array of size `N` is smaller than `BLOCK_SIZE`, the function `partial_sha256_var_start` does not even go into the logical loop to perform any hashing due to calculating the number of loops based on `N`:

lib/src/partial_hash.nr:L79-L84

```
let num_blocks = N / BLOCK_SIZE;
let mut msg_block: [u8; BLOCK_SIZE] = [0; BLOCK_SIZE];
let mut h: [u32; 8] = [1779033703, 3144134277, 1013904242, 2773480762, 1359893119, 2600822924, 528734635, 1541459225]; // Intermediate hash, star
let mut msg_byte_ptr = 0; // Pointer into msg_block

for i in 0..num_blocks {
```

Since `num_blocks=0` in this case, the loops are skipped and the hash state `h` remains unchanged after its initialization. Consequently, it is returned in its original state for all data passed to this function that is smaller than `BLOCK_SIZE`.

Practically, this can be used to prove that any sufficiently small data returns the same hash, which is false.

Recommendation

This can be addressed in two ways:

- Handle small data sizes by addressing this special case and complete the full hash with all the necessary SHA256 padding.
- Require `N` to be at least `BLOCK_SIZE` big.

5.8 Update Relevant Parts of `partial_hash.nr` Code to the Latest `sha256.nr` Library by Aztec Medium

Description

Since the beginning of this audit, Aztec has updated the `sha256.nr` library that is heavily utilized by the `partial_hash.nr` library of this repository. Among refactoring, some updates stand out such as additional constraints:

```
        if !is_unconstrained() {
            let new_msg_byte_ptr = verify_msg_block(msg, message_size, msg_block, msg_start);
            if msg_start < message_size {
                msg_byte_ptr = new_msg_byte_ptr;
                verify_msg_block_padding(msg_block, msg_byte_ptr);
            }
```

Specifically, `verify_msg_block_padding` has been added to ensure proper padding of SHA256 blocks at the end of the hash.

As the Aztec library and its updates are outside of scope of this audit, it is difficult to gauge the true impact of not having this constraint but it is recommended to err on the side of caution and update the audit repository's library as well.

Recommendation

Update the `partial_hash.nr` library to be in sync with Aztec's `sha256.nr` library. At least include the newly added constraint for verifying message block padding.

5.9 `partial_sha256_var_interstitial` Unnecessary Constraints When `N` Is Larger Than `message_size` by More Than `BLOCK_SIZE` Medium

Description

As described in issue 5.6 , there are issues in the function `partial_sha256_var_interstitial` with choosing to utilize `N` over `message_size` when calculating `num_blocks`. In the case that `N` is significantly larger than `message_size` so that it fits more multiples of `BLOCK_SIZE` in it, `num_blocks` will be greater than the amount of `BLOCK_SIZE` blocks that can fit into the requested `message_size` of to-be-hashed data:

lib/src/partial_hash.nr:L125

```
let num_blocks = N / BLOCK_SIZE;
```

This does not affect the resulting hash due to the final check that ensures that the hash of the current loop does not start beyond `message_size`:

lib/src/partial_hash.nr:L150-L152

```
if (msg_start < N) & (msg_byte_ptr == BLOCK_SIZE) & (msg_start < message_size){
    h = sha256_compression(msg_u8_to_u32(msg_block), h);
}
```

However, in a constrained execution, this loop still verifies that the potential block of data to be hashed is correct:

lib/src/partial_hash.nr:L138-L143

```
if !is_unconstrained() {
    // Verify the block we are compressing was appropriately constructed
    let new_msg_byte_ptr = verify_msg_block(msg, N, msg_block, msg_start);
    if msg_start < N {
        msg_byte_ptr = new_msg_byte_ptr;
    }
```

This verification performs a constraint on every character of the block:

lib/src/partial_hash.nr:L55-L60

```
for k in msg_start..msg_end {
    if k < message_size {
        assert_eq(msg_block[msg_byte_ptr], msg[k]);
        msg_byte_ptr = msg_byte_ptr + 1;
    }
}
```

This can get quite expensive for data objects with a large size `N` even if the requested `message_size` is intended to be small.

Recommendation

Consider calculating `num_blocks` using `message_size` as well `N` to avoid unnecessary expensive constraints and trying to access characters out of bounds. Also consider instead adding a check in the unconstrained execution that the current loop's `msg_start` is smaller than `message_size`, just as the last check prior to hashing does.

5.10 Several Leniencies in `get_body_hash` Medium

Description

The `get_body_hash` function in `body_hash.nr` works in four steps:

Step 1. It first calls `constrain_header_field` with the given `dkim_header_field_sequence` and "dkim-signature" as header field name:

lib/src/headers/body_hash.nr:L18-L22

```
// constrain the access of dkim signature field
let header_field_name: [u8; 14] = comptime {
    "dkim-signature".as_bytes()
};
constrain_header_field::<MAX_HEADER_LENGTH, MAX_DKIM_HEADER_FIELD_LENGTH, 14>(header, dkim_header_field_sequence, header_field_name);
```

We identified several vulnerabilities in `constrain_header_field`, but these have already been discussed in other findings. Hence, we want to assume in the rest of this discussion that `constrain_header_field` has been fixed and that `dkim_header_field_sequence` indeed marks the DKIM-Signature header field.

Next, we're interested in the location of the body hash within this header field. As the body hash is the base64-encoded SHA-256 hash of the email body (SHA-256 is the only hash function currently supported), it has a fixed length of 44 bytes. Hence, in order to specify the location of the body hash, we don't need a *sequence* – which consists of start index and length – but can just use its first position, which is given as `body_hash_index`. The next step is as follows:

Step 2. Assert that `body_hash_index` is within the DKIM-Signature header field.

lib/src/headers/body_hash.nr:L23-L27

```
// constrain access to the body hash
assert(
    body_hash_index > dkim_header_field_sequence.index
    & body_hash_index < dkim_header_field_sequence.end_index(), "Body hash index accessed outside of DKIM header field"
);
```

Note that this assertion is sloppier than it could be in the following two ways:

1. The DKIM-Signature header starts with “dkim-signature:” and the body hash has to be prepended with “bh=”, so the first possible position for the body hash is not `dkim_header_field_sequence.index + 1`, but `dkim_header_field_sequence.index + 18`. That is not exploitable, as the next step will verify that the body hash is indeed prepended with “bh=”, and there can no overlap between the strings “dkim-signature:” and “bh=”, but it is unnecessary to be less stringent than we can be.
2. We only verify that the *first* index of the body hash is within the DKIM-Signature header, not the *entire* body hash with its 44 characters. Hence, we might read past the end of the DKIM-Signature header and even access the header out of bounds, since `get_body_hash_unsafe` uses `get_unchecked`. One might hope that other constraints come to the rescue and effectively prevent that, but we don’t think that’s the case here, and even if it were, this should not be left to intricate reasoning but rather verified immediately – especially since it’s simple and cheap to make this constraint tighter. (Remark: While it is not important for this analysis, recall that the `end_index` function returns the first position *after* the sequence, not the last position *in* the sequence. So it is indeed only the first position of the body hash that is verified to be inside the DKIM header, not the first two.)

Step 3. Verify that the body hash is prepended by “bh=”.

lib/src/headers/body_hash.nr:L28-L35

```
let bh_prefix: [u8; 3] = comptime {
    "bh=".as_bytes()
};
for i in 0..3 {
    assert(
        header.get_unchecked(body_hash_index - 3 + i) == bh_prefix[i], "No 'bh=' prefix found at asserted bh index"
    );
}
```

The DKIM-Signature header is composed of a semicolon-separated list of tags, with tags such as “h=…”, “c=…”, or – the one we’re interested in – “bh=…”. However, the check in the code above doesn’t actually verify that the “bh=” is indeed the start of a tag; it could occur anywhere in the header field. Note that RFC 6376 allows the inclusion of “non-standard” tags in the DKIM-Signature header. So it can’t be ruled out that a tag like “xbh=…” is part of this header. Or the string “bh=” could be part of the *value* of some tag (e.g., “foo=…bh=…;”); the standard doesn’t exclude this possibility either, even if such things might be uncommon or even rare in practice.

Step 4. In this last step, the 44 bytes starting at position `body_hash_index` are fed into the `noir_base64` library’s decoding function to obtain and return the original body hash – that can then be compared to the *computed* hash of the body.

lib/src/headers/body_hash.nr:L36-L59

```
    // get the body hash
    get_body_hash_unsafe(header, body_hash_index)
}

/**
 * Get the body hash from the header without validating the access index
 *
 * @param MAX_HEADER_LENGTH - The maximum length of the email header
 * @param header - The email header as validated in the DKIM signature
 * @param body_hash_index - The asserted index to find the body hash at
 */
pub fn get_body_hash_unsafe<let MAX_HEADER_LENGTH: u32>(
    header: BoundedVec<u8, MAX_HEADER_LENGTH>,
    body_hash_index: u32
) -> [u8; 32] {
    // get the body hash
    let mut body_hash_encoded: [u8; BODY_HASH_BASE64_LENGTH] = [0; BODY_HASH_BASE64_LENGTH];
    for i in 0..BODY_HASH_BASE64_LENGTH {
        body_hash_encoded[i] = header.get_unchecked(body_hash_index + i);
    }
    // return the decoded body hash
    // idk why encode vs decode...
    base64_encode::<BODY_HASH_BASE64_LENGTH, 32>(body_hash_encoded)
}
```

The library function is erroneously named `base64_encode`, but it is actually the decoding function. More importantly, however, it does – in the version of `noir_base64` used: 0.2.2 – not verify that the given byte array contains only valid base64 characters; instead, it treats any character outside of [A-Za-z0-9+/] in the same way as the character “A”, as can be seen in the following table that is used to process the input:

```
            table: [
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0,// 0-9
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0,// 10-19
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0,// 20-29
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0,// 30-39
                0, 0, 0,// 40-42
                62,// 43
                0, 0, 0,// 44-46
                63,// 47
                52, 53, 54, 55, 56, 57, 58, 59, 60, 61,// 48-57
                0, 0, 0,// 58-60,
                0,// 61
                0, 0, 0,// 62-64
                0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25,// 65-90 (A-Z)
                0, 0, 0, 0, 0, 0,// 91-96
                26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51,// 97-122 (a-z)
                0, 0, 0, 0, 0,// 123-127
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
            ]
```

Hence, we cannot rely on this function to mitigate against the leniencies above or even, more generally, provide (additional) protection against reading from a wrong position; it will happily decode any character sequence even if said sequence contains characters that are not part of the base64 alphabet.

Recommendation

We recommend the following changes:

I. In step 2 above, tighten the assertion

lib/src/headers/body_hash.nr:L24-L27

```
assert(
    body_hash_index > dkim_header_field_sequence.index
    & body_hash_index < dkim_header_field_sequence.end_index(), "Body hash index accessed outside of DKIM header field"
);
```

to

```
  assert(
      body_hash_index >= dkim_header_field_sequence.index + 18
      & body_hash_index + BODY_HASH_BASE64_LENGTH <= dkim_header_field_sequence.end_index(), "Body hash index accessed outside of DKIM header field"
  );
```

II. In step 3, also make sure the “bh=” in front of the (claimed) body hash is actually a tag label. Specifically, assert that the “bh=” is either at the very beginning of the header field body (i.e., starts at position 14 of the header field, directly after “dkim-signature:”) or follows after a “; " (semicolon and space). Also assert that the end of alleged body hash either marks the end of the DKIM-Signature header too or is followed by a semicolon.

III. Use the newest version of the `noir_base64` library. The issue above in particular, as well as the reversed function names for encoding and decoding have been fixed in release v0.2.3. The latest release at the time of writing this report is v0.3.1. Since the library is in active development and probably hasn’t been audited and/or seen extensive use so far, we recommend reviewing the version used for the actual deployment for (1) compatiibility with ZK Email and (2) absence of bugs. *Note that we have not fully reviewed any version of `noir_base64` for either of these two properties, and this recommendation is purely based on the fact that the issues above are claimed to have been fixed in v0.2.3 and the general assumption that later versions are more mature (e.g., proper handling of padding has been introduced in v0.3.0).* See also issue 5.14.

On a more general note, while we recognize the need to be mindful of the number of constraints generated, we think that *some* (apparent) redundancy or (over-)cautiousness in the constraints adds a safety buffer that’s valuable. This is especially the case if the reasoning why some constraint is (seemingly) *not* necessary is “non-local” and includes other functions that might be spread across the codebase or even external libraries.

5.11 Missing or Weak Constraints Regarding Email Address Sequence Medium

Description

I. The maximum length of an email address is defined as a constant in `lib.nr`

lib/src/lib.nr:L21

```
global MAX_EMAIL_ADDRESS_LENGTH: u32 = 320;
```

and the functions `get_email_address` as well as `parse_email_address` in `email_address.nr` each return a `BoundedVec` of maximum size `MAX_EMAIL_ADDRESS_LENGTH`. Specifically, both functions have a parameter `email_address_sequence` and together are supposed to verify that this sequence indeed marks an email address – and then return the characters that this sequence contains as a `BoundedVec<u8, MAX_EMAIL_ADDRESS_LENGTH>`.

However, there is no check that `email_address_sequence.length` is smaller than or equal to `MAX_EMAIL_ADDRESS_LENGTH`, i.e., the given sequence could be longer than the maximum allowed email address. In this case, there would be problems: Just the first `MAX_EMAIL_ADDRESS_LENGTH` elements of the sequence would be checked to contain only allowed characters and would be returned as email address. On top of that, the length of the bounded vector would be set to a value that is larger than its maximum length:

lib/src/headers/email_address.nr:L49-L59

```
for i in 0..MAX_EMAIL_ADDRESS_LENGTH {
    let index = email_address_sequence.index + i;
    if index < email_address_sequence.end_index() {
        let letter = header.get_unchecked(index);
        email_address.storage[i] = letter;
        assert(
            EMAIL_ADDRESS_CHAR_TABLE[letter] == 1, "Email address must only contain acceptable characters"
        );
    }
}
email_address.len = email_address_sequence.length;
```

In other places in these functions, the *entire* sequence is treated as the part that is supposed to contain the address, leading to inconsistencies with the behavior described above:

lib/src/headers/email_address.nr:L41-L45

```
if email_address_sequence.end_index() < header.len() {
    assert(
        EMAIL_ADDRESS_CHAR_TABLE[header.get_unchecked(email_address_sequence.index + email_address_sequence.length)]
        == 3, "Email address must end with an acceptable character"
    );
```

II. The check that the email address sequence is contained in the header field sequence is weaker than it could and should be:

lib/src/headers/email_address.nr:L21-L25

```
// check email sequence is within header field
assert(
    email_address_sequence.index >= header_field_sequence.index
    & email_address_sequence.end_index() <= header_field_sequence.end_index(), "Email address sequence out of bounds"
);
```

Since the header field has to start with a field name, followed by a colon, we could actually assert that `email_address_sequence.index >= header_field_sequence.index + HEADER_FIELD_NAME_LENGTH + 1`.

Recommendation

We recommend the following two changes:

I. Add an assertion to `parse_email_address` as follows:

```
  assert(
      email_address_sequence.length <= MAX_EMAIL_ADDRESS_LENGTH,
      "Email address sequence exceeds maximum allowed length"
  );
```

II. Change the following assertion

lib/src/headers/email_address.nr:L22-L25

```
assert(
    email_address_sequence.index >= header_field_sequence.index
    & email_address_sequence.end_index() <= header_field_sequence.end_index(), "Email address sequence out of bounds"
);
```

to

```
  assert(
      email_address_sequence.index > header_field_sequence.index + HEADER_FIELD_NAME_LENGTH
      & email_address_sequence.end_index() <= header_field_sequence.end_index(), "Email address sequence out of bounds"
  );
```

Remark: Not all findings are independent of each other. One could, for example, argue that after fixing II and issue 5.4, I isn't a problem anymore. One might even try to reason what can and can't happen by taking into account that we assert that the header field name is followed by a colon and that the claimed email address is verified to not contain a colon. However, we believe that essential properties like the above should not rely on too intricate and perhaps "non-local" reasoning but should be verified directly.

5.12 Email Address Doesn't Support Some Special Characters Medium

Description

The `parse_email_address` function in `email_address.nr` doesn't support email addresses where there are spaces between the right angle bracket and the address, e.g. `<sender@contoso.com >` , `< sender@contoso.com >` that are compatible with the Microsoft Exchange mail server standard . The following assert would fail when the ending character is space. The accepted ending character are `>` , `r` and `n` .

```
if email_address_sequence.end_index() < header.len() {
       assert(
           EMAIL_ADDRESS_CHAR_TABLE[header.get_unchecked(email_address_sequence.index + email_address_sequence.length)]
           == 3, "Email address must end with an acceptable character"
       );
   }
```

Additionally, the function also doesn't support some commonly used special characters such as `+` or `_` in the address

Examples

lib/src/lib.nr:L33-L51

```
// allowable chars in an email address (js/src/utils:makeAllowableEmailCharsTable()
// "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789.-@" = 1
// "<: " = 2
// ">\r\n" = 3
global EMAIL_ADDRESS_CHAR_TABLE: [u8; 123] = [
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    3, 0, 0, 3, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 2, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 1, 1, 0, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 2, 0,
    2, 0, 3, 0, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 0, 0, 0, 0, 0, 0, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
    1, 1, 1,
];
```

lib/src/headers/email_address.nr:L41-L58

```
if email_address_sequence.end_index() < header.len() {
    assert(
        EMAIL_ADDRESS_CHAR_TABLE[header.get_unchecked(email_address_sequence.index + email_address_sequence.length)]
        == 3, "Email address must end with an acceptable character"
    );
}
// check the email address and assign
let mut email_address: BoundedVec<u8, MAX_EMAIL_ADDRESS_LENGTH> = BoundedVec::new();
for i in 0..MAX_EMAIL_ADDRESS_LENGTH {
    let index = email_address_sequence.index + i;
    if index < email_address_sequence.end_index() {
        let letter = header.get_unchecked(index);
        email_address.storage[i] = letter;
        assert(
            EMAIL_ADDRESS_CHAR_TABLE[letter] == 1, "Email address must only contain acceptable characters"
        );
    }
}
```

Recommendation

Add support for spaces between the right angle brackets and the email address and special characters in the address body

5.13 Header Field `To` Doesn't Support Multiple Recipient Addresses Medium

Description

In email_address.nr, `get_email_address` does not support multiple recipient addresses in the header field `To` , e.g. Receiver 1 `<receiver1@gmail.com>` , `"receiver2@consensys.net" <receiver2@consensys.net>` . `constrain_header_field_detect_last_angle_bracket` will return the location of the last bracket `<` in the last receiver `<receiver2@consensys.net>` , consequently `parse_email_address` would return the last receiver address (`receiver2@consensys.net`) and the previous one will be skipped. In the case of recipient addresses without brackets, e.g. `receiver1@gmail.com` , `receiver2@consensys.net` . The parsing would fail as the comma `,` is not matching the ending characters of `>` , `\r` , or `\n`

Examples

lib/src/headers/email_address.nr:L7-L63

```
pub fn get_email_address<let MAX_HEADER_LENGTH: u32, let HEADER_FIELD_NAME_LENGTH: u32>(
    header: BoundedVec<u8, MAX_HEADER_LENGTH>,
    header_field_sequence: Sequence,
    email_address_sequence: Sequence,
    header_field_name: [u8; HEADER_FIELD_NAME_LENGTH]
) -> BoundedVec<u8, MAX_EMAIL_ADDRESS_LENGTH> {
    // check field is uninterrupted and matches the expected field name
    let last_angle_bracket = constrain_header_field_detect_last_angle_bracket::<MAX_HEADER_LENGTH, MAX_EMAIL_ADDRESS_LENGTH + HEADER_FIELD_NAME_L
    // if angle bracket found, assert index is +1
    if last_angle_bracket != 0 {
        assert(
            email_address_sequence.index == last_angle_bracket + 1, "Email address must start immediately after '<' if bracket is present"
        );
    }
    // check email sequence is within header field
    assert(
        email_address_sequence.index >= header_field_sequence.index
        & email_address_sequence.end_index() <= header_field_sequence.end_index(), "Email address sequence out of bounds"
    );

    // constrained get email address
    parse_email_address(header, email_address_sequence)
}

pub fn parse_email_address<let MAX_HEADER_LENGTH: u32>(
    header: BoundedVec<u8, MAX_HEADER_LENGTH>,
    email_address_sequence: Sequence
) -> BoundedVec<u8, MAX_EMAIL_ADDRESS_LENGTH> {
    // check the sequence is proceeded by an acceptable character
    if email_address_sequence.index != 0 {
        assert(
            EMAIL_ADDRESS_CHAR_TABLE[header.get_unchecked(email_address_sequence.index - 1)] == 2, "Email address must start with an acceptable c
        );
    }
    if email_address_sequence.end_index() < header.len() {
        assert(
            EMAIL_ADDRESS_CHAR_TABLE[header.get_unchecked(email_address_sequence.index + email_address_sequence.length)]
            == 3, "Email address must end with an acceptable character"
        );
    }
    // check the email address and assign
    let mut email_address: BoundedVec<u8, MAX_EMAIL_ADDRESS_LENGTH> = BoundedVec::new();
    for i in 0..MAX_EMAIL_ADDRESS_LENGTH {
        let index = email_address_sequence.index + i;
        if index < email_address_sequence.end_index() {
            let letter = header.get_unchecked(index);
            email_address.storage[i] = letter;
            assert(
                EMAIL_ADDRESS_CHAR_TABLE[letter] == 1, "Email address must only contain acceptable characters"
            );
        }
    }
    email_address.len = email_address_sequence.length;
    // todo: should probably introduce a check for @

    email_address
}
```

Recommendation

Add support for multiple receivers for header field through integrating zk-regex into Noir .

5.14 Recommendations Regarding the Use of Libraries Medium

Description and Recommendation

The ZK Email Noir code makes use of several libraries, and while these libraries themselves were out of scope for this review, we happened to notice in the context of issue 5.10 that v0.2.2 of `noir_base64` – which is used in this project – has, at the time of this review, known issues. These have been fixed in later versions.

Noir is a relatively young language, and many Noir libraries are in active development and still change frequently as well as substantially. It is hard to give a good recommendation in such an environment, but we suggest the following guidelines:

1. Actively follow new releases and bugfixes in the libraries you're using.
2. Generally use a version without known issues, or at least make sure that the issues don't affect your project.
3. If possible, review the code yourself for e.g. compatibility, hidden or undocumented assumptions, and potentially even bugs.
4. Don't be over-confident in the security/corrrectness/robustness of the library. For instance, if in doubt, add extra constraints yourself instead of relying on the library to properly validate the input.

5.15 Questionable Unchecked Access of Bounded Vector Elements Minor

Description

In the `constrain_header_field` function, the following unchecked access in a bounded vector is potentially out of bounds:

lib/src/headers/mod.nr:L47

```
assert(header.get_unchecked(end_index + 1) == LF,
```

At this point, we have verified that `end_index <= header.len()` and that `end_index != header.len()`, so we know that `end_index < header.len()`, but `end_index + 1` could still be the first out-of-bounds position in the bounded vector.

The same situation occurs in `constrain_header_field_detect_last_angle_bracket`:

lib/src/headers/mod.nr:L108

```
assert(header.get_unchecked(end_index + 1) == LF,
```

However, if the unchecked access was indeed out-of-bounds (and all constraints satisfied), then the header would end in CR. This shouldn't be the case for a well-formed header, and since the header is signed by a trusted party, we may assume that this can't happen.

Nevertheless, we believe that something as fundamental as preventing out-of-bounds access should not rely on circuit-external factors like trust assumptions. In fact, we think it should not even be based on somewhat elaborate reasoning about the circuits, but it should be fairly obvious from looking at the code that every access is within bounds.

Recommendation

In both functions, either replace the unchecked access with a checked one, or add an additional `assert` before the unchecked access. As a general rule and as discussed above, we recommend making sure that every unchecked access is quite obviously within bounds. If a function, for efficiency reasons, doesn't want to verify itself that access is within bounds, it should be clearly documented that it is the caller's responsibility to ensure this.

5.16 Missing Tests for Partial Hash When Message Size Is Not Multiple of 64 Bytes Minor

Description

There are no tests for partial hash functions `partial_sha256_var_start`, `partial_sha256_var_interstitial`, `partial_sha256_var_end` when the message size is not multiple of block size (64 bytes), which causes the occurrence of multiple issues issue 5.5, issue 5.6 and issue 5.7.

Recommendation

Add the missing tests for partial hash functions.

5.17 Redundant Check Whether Sequence Is Within Bounds Minor

Description

The first thing the `constrain_header_field` function does is verify that the given sequence doesn't extend beyond the length of the header:

lib/src/headers/mod.nr:L27-L31

```
// check that the sequence is within bounds
assert(
    header_field_sequence.index + header_field_sequence.length <= header.len(),
    "Header field out of bounds"
);
```

In the next step, the exact same check is repeated:

lib/src/headers/mod.nr:L32-L34

```
// check the range of the sequence is within the header (so we can use get_unchecked)
let end_index = header_field_sequence.index + header_field_sequence.length;
assert(end_index <= header.len(), "Header field out of bounds of header");
```

The same happens in `constrain_header_field_detect_last_angle_bracket`:

lib/src/headers/mod.nr:L88-L95

```
// check that the sequence is within bounds
assert(
    header_field_sequence.index + header_field_sequence.length <= header.len(),
    "Header field out of bounds"
);
// check the range of the sequence is within the header (so we can use get_unchecked)
let end_index = header_field_sequence.index + header_field_sequence.length;
assert(end_index <= header.len(), "Header field out of bounds of header");
```

Recommendation

Since the checks are redundant, one of them can be removed in both `constrain_header_field` and `constrain_header_field_detect_last_angle_bracket`. Since the variable `end_index` that is introduced for the second check is used again later in the functions, it makes more sense to remove the first check than the second.

5.18 Unnecessary Addition in `find_zeroes` Minor

Description

In `remove_soft_line_breaks.nr`, there is a function that determines whether a series of characters represents a line break, and, if so, marks them all to be zeroed out. Specifically, the function looks for a 3-character block that constitutes a soft line break:

lib/src/remove_soft_line_breaks.nr:L31-L35

```
for i in 0..N - 2 {
    is_break[i] = (encoded[i] == 0x3D)
        & (encoded[i + 1] == 0x0D)
        & (encoded[i + 2] == 0x0A);
}
```

As can be seen above, the result is that only the first of the characters in the 3-char block is marked `1` in the `is_break[]` array, namely the character that corresponds to `0x3D`. Consequently, the following 2 characters in the block would **necessarily** not be marked in `is_break[]`, as they don't start with `0x3D`, and, in fact, they are confirmed to be `0X0D` and `0X0A`, in that order.

Since this is a 3-char block, the function needs to give special treatment around the message boundaries. For example, the last character of an array can not be marked as `1` in `is_break[]` since it doesn't have 2 more characters after it (on top of it not even being in the above loop as it only goes up to `N-3` and not `N-1`). To compute whether the last character needs to be zeroed out, the function looks at two characters before it and sees if any of them contain a soft break:

lib/src/remove_soft_line_breaks.nr:L41

```
should_zero[N - 1] = is_break[N - 2] + is_break[N - 3];
```

However, since a soft line break is a 3-character block, the penultimate character (`is_break[N-2]`) can not be a soft line break, as it only has 1 character ahead of it, instead of the required 2. So, there is no need to evaluate it, and the function can determine whether to zero-out character `N-1` by evaluating `is_break[N-3]` alone.

Recommendation

The line quoted above could be simplified to:

```
should_zero[N - 1] = is_break[N - 3];
```

Appendix 1 - Files in Scope

This audit covered the following files:

File	SHA-1 hash
lib/src/headers/body_hash.nr	`632515855b5b81acf21ccdf73f19fc7ce5549587`
lib/src/headers/email_address.nr	`d9c6893f0afe849cd39e6f1ae376fed6e13edb81`
lib/src/headers/mod.nr	`f083ac1a88d95710313d9ec0de0061baf3f5f90b`
lib/src/dkim.nr	`4bfeb2a020f7f888f573348a03784bf85be56783`
lib/src/lib.nr	`e7159dec69e4249493d73ed9df719e51fb331faf`
lib/src/masking.nr	`c070370b4422f9fe8e5a1e7b33d78a9593d806a7`
lib/src/partial_hash.nr	`ad86deba56bc5abf8a3c53829f142c67f45337b3`
lib/src/remove_soft_line_breaks.nr	`56f144003314dc3df4f25a69fc4b7a528f4148b2`

Appendix 2 - Disclosure

Consensys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via Consensys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any third party in any respect, including regarding the bug-free nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any third party by virtue of publishing these Reports.

A.2.1 Purpose of Reports

The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of code and only the code we note as being within the scope of our review within this report. Any Solidity code itself presents unique and unquantifiable risks as the Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond specified code that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. In some instances, we may perform penetration testing or infrastructure assessments depending on the scope of the particular engagement.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

A.2.2 Links to Other Web Sites from This Web Site

You may, through hypertext or other computer links, gain access to web sites operated by persons other than Consensys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that Consensys and CD are not responsible for the content or operation of such Web sites, and that Consensys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that Consensys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. Consensys and CD assumes no responsibility for the use of third-party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

A.2.3 Timeliness of Content

The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice unless indicated otherwise, by Consensys and CD.