Competitive Security Assessment

UniPassEmailCircuits

Jun 26th, 2023

Secure3

secure3.io

Summary

This report is prepared for the project to identify vulnerabilities and issues in the smart contract source code. A group of NDA covered experienced security experts have participated in the Secure3's Audit Contest to find vulnerabilities and optimizations. Secure3 team has participated in the contest process as well to provide extra auditing coverage and scrutiny of the finding submissions.

The comprehensive examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static analysis tools to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and security best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in four severity levels: Critical, Medium, Low, Informational. For each of the findings, the report has included recommendations of fix or mitigation for security and best practices.

Overview

Project Detail

Project Name	UniPassEmailCircuits
Platform & Language	Rust
Codebase	• https://github.com/UniPassID/UniPass-email-circuits • audit commit - 406c5692dceade9ffaa2b79da4d230a23252f4de • final commit - e1bd81675f38e3051801325f0a23f2d49145f0d1
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

Code Vulnerability Review Summary

Vulnerability Level	Total	Reported	Acknowledged	Fixed	Mitigated	Declined
Critical	**5**	**0**	**1**	**0**	**2**	**2**
Medium	**4**	**0**	**1**	**3**	**0**	**0**
Low	**11**	**0**	**3**	**4**	**2**	**2**
Informational	**25**	**0**	**2**	**17**	**3**	**3**

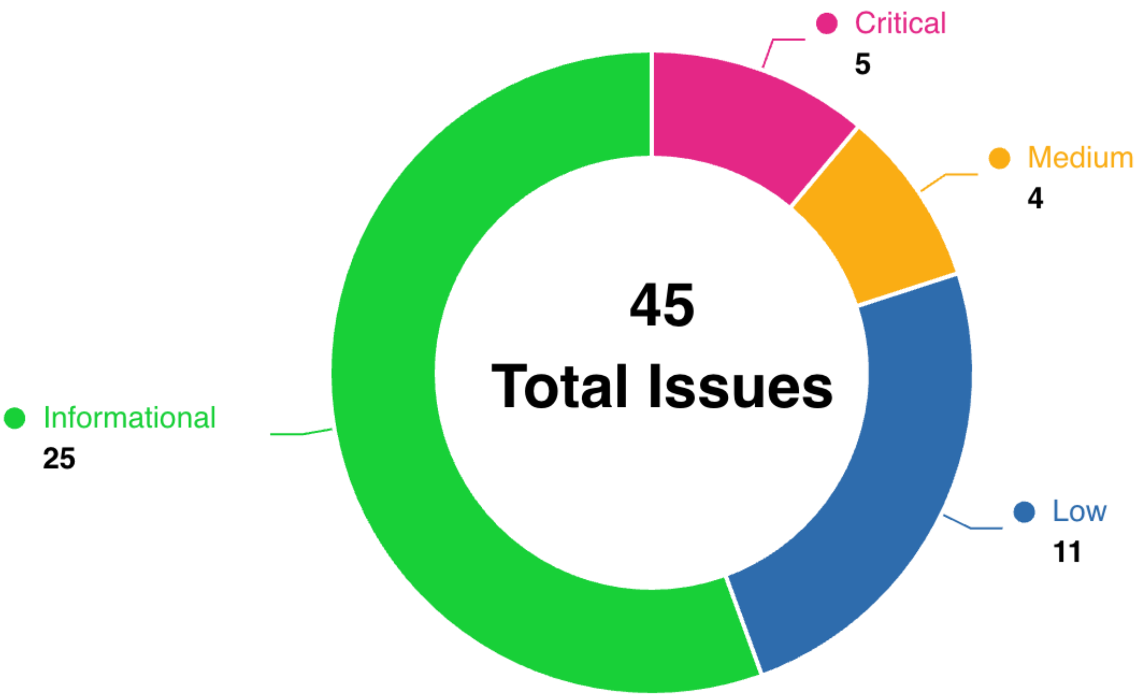
Audit Scope

File	Commit Hash
UniPass-email-circuits/crates/plookup-sha256/src/sha256.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/plookup-sha256/src/prover/mod.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/plookup-sha256/src/composer/mod.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/prover/src/circuit/openid.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/plookup-sha256/src/verifier.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/plookup-sha256/src/kzg10.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/prover/src/utils.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/prover/src/circuit/circuit_2048_triple.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/plookup-sha256/src/composer/mimc.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/plookup-sha256/src/utils.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/prover/src/circuit/circuit_2048.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/prover/src/circuit/circuit_1024.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/plookup-sha256/src/composer/substring.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/prover/src/parameters/iden3.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/prover/src/types.rs	406c5692dceade9ffaa2b79da4d230a23252f4de
UniPass-email-circuits/crates/plookup-sha256/src/prover/widget/lookup.rs	406c5692dceade9ffaa2b79da4d230a23252f4de

UniPass-email-circuits/crates/plookup-sha256/src/prover/widget/permutation	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/composer/lookup.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/prover/src/circuit/base64.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/email-parser/src/parser.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/email-parser/src/types.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/prover/widget/substring.r	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/composer/arithmetic.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/prover/widget/range.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/prover/prover_key.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/prover/widget/arithmetic.	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/prover/src/parameters.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/transcript.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/prover/widget/mimc.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/composer/range.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/prover/widget/pubmatch.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/composer/permutation.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/lib.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**

UniPass-email-circuits/crates/plookup-sha256/src/prover/widget/mod.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/email-parser/src/error.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/plookup-sha256/src/proof.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/prover/src/error.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/prover/src/circuit.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/prover/src/lib.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**
UniPass-email-circuits/crates/email-parser/src/lib.rs	**406c5692dceade9ffaa2b79da4d230a23252f4de**

Code Assessment Findings



ID	Name	Category	Severity	Status	Contributor
UPC-1	Missing check email DKIM-Signature in circuit	Signature Forgery or Replay	Critical	Mitigated	zircon
UPC-2	Vulnerability Similar To Fronzen Heart Attack on Plonk	Signature Forgery	Critical	Declined	lfzkoala
UPC-3	Write to Arbitrary Storage Location in `parser.rs` and `sha256.rs`	Write to Arbitrary Storage Location	Critical	Declined	newway55

UPC-4	ZK prove can be replay	Signature Forgery or Replay	Critical	Mitigated	zircon
UPC-5	Zero Bug in Plonk Verifier	Signature Forgery or Replay	Critical	Acknowledged	Ifzkoala
UPC-6	Infinite loop and Denial of Service (DoS) attack in find_subsequence function	Language Specific	Medium	Fixed	Ifzkoala, BradMoonUESTC
UPC-7	Unsafe Use of Indexing	Language Specific	Medium	Acknowledged	Ifzkoala
UPC-8	Weak Sources of Randomness risk in `mimc.rs` , `types.rs` and `parameters.rs`	Weak sources of Randomness	Medium	Fixed	Ifzkoala, newway55
UPC-9	Wires should be appended to the root position in `Composer::finalize`	Logical	Medium	Fixed	alansh
UPC-10	Avoid using assert! in production code	Code Style	Low	Mitigated	Ifzkoala
UPC-11	Base58 is malleable	Logical	Low	Declined	zircon
UPC-12	Denominator may not be always invertible	Language Specific	Low	Fixed	Ifzkoala
UPC-13	Insecure Deserialization	Logical	Low	Acknowledged	Ifzkoala, BradMoonUESTC
UPC-14	Integer Overflow and Underflow risk in `arithmetic.rs`	Integer Overflow and Underflow	Low	Declined	newway55
UPC-15	Lack Input Validation	Code Style	Low	Fixed	Ifzkoala, BradMoonUESTC
UPC-16	Logical risk in `Prover implementation` for `insert` functions	Logical	Low	Mitigated	newway55
UPC-17	Public Mutable References	Logical	Low	Acknowledged	BradMoonUESTC

UPC-18	Unsafe type conversion in prepare_generic_params	Integer Overflow and Underflow	Low	Acknowledged	BradMoonUESTC
UPC-19	Use of String::from_utf8_lossy for potentially non-UTF8 data	Language Specific	Low	Fixed	lfzkoala
UPC-20	min size of coset not accurate in `ProverKey::new`	Logical	Low	Fixed	alansh
UPC-21	Code Style in `sha256.rs` , prover `mod.rs` and `parser.rs`	Code Style	Informational	Mitigated	newway55
UPC-22	Gas Optimization in `arithmetic.rs` in `Composer` implementation in `add` function	Gas Optimization	Informational	Declined	newway55
UPC-23	Inconsistent Hash Function Used	Logical	Informational	Declined	lfzkoala
UPC-24	Missing deserialization result validation	Logical	Informational	Acknowledged	BradMoonUESTC
UPC-25	No error handling for empty email_header_bytes and email_addr_pepper_bytes	Privilege Related	Informational	Mitigated	BradMoonUESTC
UPC-26	Padding bytes may introduce unnecessary computations	Privilege Related	Informational	Declined	BradMoonUESTC
UPC-27	Publicly Exposed Constant	Privilege Related	Informational	Mitigated	lfzkoala
UPC-28	Unnecessary cloning of the chunk_messages	Language Specific	Informational	Fixed	lfzkoala
UPC-29	Unnecessary mut in the get_value_by_key function	Code Style	Informational	Acknowledged	lfzkoala
UPC-30	Unused code	Code Style	Informational	Fixed	alansh
UPC-31	Using `Cargo Clippy` Specification Code	Code Style	Informational	Fixed	Hellobloc
UPC-32	`PCKey` has both `pub max_degree` and `pub fn max_degree`	Code Style	Informational	Fixed	alansh
UPC-33	`blind_and_coset_fft` should ensure the highest blinding coefficient is not zero	Logical	Informational	Fixed	alansh

UPC-34	`epicycles`/`wires` could be updated in place in `Composer::finalize`	Gas Optimization	Informational	Fixed	alansh
UPC-35	`if` and `while` are using the same condition in `Prover::new`	Code Style	Informational	Fixed	alansh
UPC-36	comment wrong for coset_generator	Logical	Informational	Fixed	alansh
UPC-37	coset for `sigma_4` is not removed in `PermutationWidget::compute_quotient_contribution`	Logical	Informational	Fixed	alansh
UPC-38	dangling branch in padding_bytes	Logical	Informational	Fixed	alansh
UPC-39	duplicate code in Table::spread_table and Table::spread_table_2in1	Code Style	Informational	Fixed	alansh
UPC-40	loop code can be more breviate in `Prover::new`	Code Style	Informational	Fixed	alansh
UPC-41	missing assert! in test_setup function	Logical	Informational	Fixed	alansh
UPC-42	performance issue in `blind_t`	Gas Optimization	Informational	Fixed	alansh
UPC-43	q_arith selector is not actually used in Composer::fully_costomizable_poly_gates	Code Style	Informational	Fixed	alansh
UPC-44	typo in Composer::fully_costomizable_poly_gates	Code Style	Informational	Fixed	alansh
UPC-45	using both ark_std and std	Code Style	Informational	Fixed	alansh

UPC-1:Missing check email DKIM-Signature in circuit

Category	Severity	Status	Contributor
Signature Forgery or Replay	**Critical**	**Mitigated**	**zircon**

Code Reference

- code/UniPass-email-circuits/crates/email-parser/src/parser.rs#L15-L112

```
fn parse_header(
    dkim_msg: &[u8],
    dkim_header: &Header,
    from_pepper: Vec,
    from: String,
) -> ParserResult<(PublicInputs, PrivateInputs)> {
    let from_index = match find_subsequence(dkim_msg, b"from:") {
        Some(index) => index,
        None => return Err(ParserError::HeaderFormatError),
    };

    let from_end_index = match find_subsequence(&dkim_msg[from_index..], b"\r\n") {
        Some(index) => index + from_index,
        None => return Err(ParserError::HeaderFormatError),
    };

    let from_left_index =
        match find_subsequence(&dkim_msg[from_index..], format!("<{}>", from).as_bytes()) {
            Some(index) => {
                if index < from_end_index {
                    from_index + index + 1
                } else {
                    match find_subsequence(dkim_msg, from.as_bytes()) {
                        Some(index) => index,
                        None => return Err(ParserError::HeaderFormatError),
                    }
                }
            }
            None => match find_subsequence(dkim_msg, from.as_bytes()) {
                Some(index) => index,
                None => return Err(ParserError::HeaderFormatError),
            },
        };

    let from_right_index = from_left_index + from.len() - 1;

    let subject_index = match find_subsequence(dkim_msg, b"subject:") {
        Some(index) => index,
        None => return Err(ParserError::HeaderFormatError),
    };

    let subject_right_index = match find_subsequence(&dkim_msg[subject_index..], b"\r\n") {
        Some(index) => subject_index + index,
        None => return Err(ParserError::HeaderFormatError),
    };

    let dkim_header_index = match find_subsequence(dkim_msg, b"dkim-signature:") {
        Some(index) => index,
        None => return Err(ParserError::HeaderFormatError),
    };

    let sdid_index = {
        let d_index = match find_subsequence(&dkim_msg[dkim_header_index..], b"d=") {
            Some(index) => dkim_header_index + index,
            None => return Err(ParserError::HeaderFormatError),
        };

        match find_subsequence(&dkim_msg[d_index..], dkim_header.sdid.as_bytes()) {
            Some(index) => d_index + index,
            None => return Err(ParserError::HeaderFormatError),
        }
    };

    let sdid_right_index = sdid_index + dkim_header.sdid.len();

    let selector_index = {
        let s_index = match find_subsequence(&dkim_msg[dkim_header_index..], b"s=") {
            Some(index) => dkim_header_index + index,
            None => return Err(ParserError::HeaderFormatError),
        };

        match find_subsequence(&dkim_msg[s_index..], dkim_header.selector.as_bytes()) {
            Some(index) => s_index + index,
            None => return Err(ParserError::HeaderFormatError),
        }
    };

    let selector_right_index = selector_index + dkim_header.selector.len();

    let private_input = PrivateInputs {
        email_header: dkim_msg.to_vec(),
        from_pepper,
        from_index,
        from_left_index,
        from_right_index,
        subject_index,
        subject_right_index,
        dkim_header_index,
        selector_index,
        selector_right_index,
        sdid_index,
        sdid_right_index,
    };

    let public_input: PublicInputs = (&private_input).into();

    return Ok((public_input, private_input));
}
```

Description

zircon : We can examine the circuit's PrivateInputs and PublicInputs to determine that the ZKP circuit design checks some relevant aspects of the email, but it does not use the circuit to verify the email's DKIM-Signature.

The impact is that the verifier cannot truly confirm that the email was sent by the claimed sender. Attackers or intermediaries could use forged emails to generate proofs.

Consider below POC

```
//Prover use a fake email to generate prove
```

Recommendation

zircon : Implementing DKIM-Signature validation within the circuit typically involves using the RSA signature algorithm, which is commonly employed in email systems.

Client Response

Mitigated,DKIM signatures are verified by other contracts outside of the circuit.

UPC-2:Vulnerability Similar To Fronzen Heart Attack on Plonk

Category	Severity	Status	Contributor
Signature Forgery	Critical	Declined	Ifzkoala

Code Reference

- code/UniPass-email-circuits/crates/prover/src/circuit/circuit_2048_triple.rs#L28-L102
- code/UniPass-email-circuits/src/circuit_2048_triple.rs#L153-L154

```
    pub fn new(private_inputs: Vec) -> Result {
        assert_eq!(private_inputs.len(), 3);

        let mut all_email_header_bytes = vec![];
        let mut all_email_addr_pepper_bytes = vec![];
        let mut all_email_header_pub_matches = vec![];
        let mut all_from_left_indexes = vec![];
        let mut all_from_lens = vec![];

        for mut private_inputs in private_inputs {
            let email_addr_bytes = private_inputs.email_header
                [private_inputs.from_left_index..private_inputs.from_right_index + 1]
                .to_vec();
            let mut email_addr_pepper_bytes = email_addr_bytes.clone();
            email_addr_pepper_bytes.append(&mut private_inputs.from_pepper);

            let email_header_bytes = private_inputs.email_header.clone();

            // set any byte of "pub match string" to "0" is OK.
            // (you can only remain bytes for format checking, set all other bytes to 0)
            let mut email_header_pub_match = email_header_bytes.clone();

            for i in 0..email_header_pub_match.len() {
                if private_inputs.from_index == 0 {
                    if i >= private_inputs.from_index && i < private_inputs.from_left_index {
                        continue;
                    }
                } else {
                    if i >= private_inputs.from_index - 2 && i < private_inputs.from_left_index {
                        continue;
                    }
                }

                if i == private_inputs.from_right_index + 1 {
                    continue;
                }

                if private_inputs.subject_index == 0 {
                    if i >= private_inputs.subject_index && i < private_inputs.subject-
```

```
_right_index {
                    continue;
                }
            } else {
                if i >= private_inputs.subject_index - 2
                    && i < private_inputs.subject_right_index
                {
                    continue;
                }
            }

            if i == private_inputs.dkim_header_index - 2 {
                break;
            }

            email_header_pub_match[i] = 0;
        }

        let from_len =
            (private_inputs.from_right_index - private_inputs.from_left_index + 1) as u32;
        let from_left_index = private_inputs.from_left_index as u32;

        all_email_header_bytes.push(email_header_bytes);
        all_email_addr_pepper_bytes.push(email_addr_pepper_bytes);
        all_email_header_pub_matches.push(email_header_pub_match);
        all_from_left_indexes.push(from_left_index);
        all_from_lens.push(from_len);
    }

    Ok(Self {
        email_header_bytes: all_email_header_bytes,
        email_addr_pepper_bytes: all_email_addr_pepper_bytes,
        email_header_pub_matches: all_email_header_pub_matches,
        from_left_indexes: all_from_left_indexes,
        from_lens: all_from_lens,
    })
}

let circuit =
    Email2048TripleCircuitInput::new(all_email_private_inputs[0..3].to_vec()).unwrap();
```

Description

lfzkoala : If a zero-knowledge proof protocol is insecure, a malicious prover can forge a zk proof that will succeed verification. Depending on the details of the protocol, the forged proof can potentially be used to “prove” anything the prover wants. Additionally, many zk protocols use what is known as a Fiat-Shamir transformation. Insecure implementations of the Fiat-Shamir transformation can allow attackers to successfully forge proofs. See more details about this attack in https://blog.trailofbits.com/2022/04/13/part-1-coordinated-disclosure-of-vulnerabilities-affecting-girault-bulletproofs-and-plonk/

UniPass uses Plonk with lookup table as the zk protocol and I believe the zk protocol is secure against such attack, but the circuit of the zk protocol is newly designed by the UniPass team, in which to construct a circuit, it only takes input the private parts of the email bytes, i.e., `circuit = Email2048TripleCircuitInput::new(all_email_private_inputs[0..3].to_vec()).unwrap()`, and the `new()` function is designed to take private inputs to construct the circuit.

According to the Frozen heart attack, it should be safe to include public inputs as well in the `new()` function. Although the circuit has nothing to do with the Fiat-Shamir Transformation, it still hashed all the email inputs. I believe this may not be as serious as the frozen heart attack, but it should be more secure to include the public inputs into the hash value to construct the circuit.

Recommendation

lfzkoala : Improve the `new()` function with additional input `public_inputs: Vec<PublicInputs>` to construct the circuit and synthesize. i.e.,
`pub fn new(private_inputs: Vec<PrivateInputs>, public_inputs: Vec<PublicInputs>) -> Result<Self, ProverError>` This improvement also works for circuit 1024, 2048 and 2048 triple.

Client Response

Declined,PublicInput is generated from PrivateInput and belongs to circuit input. The location indicated by this item has nothing to do with Frozen Heart Attack, because this problem is at the algorithm level.

UPC-3:Write to Arbitrary Storage Location in `parser.rs` and `sha256.rs`

Category	Severity	Status	Contributor
Write to Arbitrary Storage Location	**Critical**	**Declined**	**newway55**

Code Reference

- code/UniPass-email-circuits/crates/email-parser/src/parser.rs#L9
- code/UniPass-email-circuits/crates/plookup-sha256/src/sha256.rs#L706-L713

```
9:fn find_subsequence(haystack: &[u8], needle: &[u8]) -> Option {

706:        let a_value = value_u64 % 8;
707:        value_u64 >>= 3;
708:        let b_value = value_u64 % 16;
709:        value_u64 >>= 4;
710:        let c_value = value_u64 % 2048;
711:        value_u64 >>= 11;
712:        let d_value = value_u64;
713:        value_u64 >>= 14;
```

Description

newway55 : The bitwise shift operations are not handled correctly.

The reason for performing a bitwise shift by 3 is to extract the 3 least significant bits from a 64-bit integer representation of a value.

- This operation effectively divides the value by 8 and discards any remainder, leaving only the 3 least significant bits.
- By performing a bitwise shift by 3 on the input word, we can extract the 3 least significant bits for the first chunk (a), and then divide the remaining bits by 16, 2048, and 16384 (2^(4+11)) to obtain the remaining chunks (b, c, and d).
- Impact : If this operation is not performed correctly, it can lead to incorrect or unexpected behavior, resulting in errors or security vulnerabilities. For example, if the value is not correctly divided by 8, there may be a remainder left over, leading to incorrect bit extraction.

Consider below POC contract

```
#[test]
fn test_sha256sigma0form_a_b_values() {
    use crate::composer::Composer;
    use ark_ff::Field;
    use super::{Sha256sigma0form, Sha256Word};

    // Set up the composer
    let mut composer = Composer::<Fp>::new(4);

    // Set up the word
    let word_var = composer.alloc_input(Fp::zero()).unwrap();
    let word = Sha256Word::new(word_var);

    // Create the Sha256sigma0form instance
    let sha256sigma0form = Sha256sigma0form::new(&mut composer, &word).unwrap();

    // Assert that a_value is calculated correctly
    let expected_a_value = Fp::from(sha256sigma0form.var.value().unwrap() % (1 << 3));
    assert_eq!(sha256sigma0form.a.value().unwrap(), expected_a_value);

    // Assert that b_value is calculated correctly
    let expected_b_value = Fp::from((sha256sigma0form.var.value().unwrap() >> 3) % (1 << 4));
    assert_eq!(sha256sigma0form.b.value().unwrap(), expected_b_value);
}
```

newway55 : - Can allow an attacker to write outside the bounds of the allocated memory, leading to unintended behavior and possibly opening up avenues for exploitation.

Consider below POC contract

This test case creates a buffer with 10 bytes and calls find_subsequence() with three different needles:

- A needle that is longer than the buffer, causing out-of-bounds memory access.
- A needle that is not present in the buffer.
- A needle that is partially present in the buffer, causing out-of-bounds memory access.

```rust
#[test]
fn test_find_subsequence_out_of_bounds() {
    // Create a buffer with 10 bytes
    let buffer = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    // Call find_subsequence with a needle that is longer than the buffer
    let needle = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11];
    let result = find_subsequence(&buffer, &needle);

    // Assert that the function returns None
    assert!(result.is_none());

    // Call find_subsequence with a needle that is not present in the buffer
    let needle = [11, 12, 13];
    let result = find_subsequence(&buffer, &needle);

    // Assert that the function returns None
    assert!(result.is_none());

    // Call find_subsequence with a needle that is partially present in the buffer
    let needle = [5, 6, 7, 8, 9, 10, 11];
    let result = find_subsequence(&buffer, &needle);

    // Assert that the function returns None
    assert!(result.is_none());
}
```

Recommendation

newway55 : The line let a_value = value_u64 % 8;, 8 should be replaced with 1 << 3 to perform a bitwise shift by 3 bits. Similarly, in let b_value = value_u64 % 16;, 16 should be replaced with 1 << 4. Finally, in let c_value = value_u64 % 2048;, 2048 should be replaced with 1 << 11.

Consider below fix in the `Sha256sigma0form` implementation

```
impl Sha256sigma0form {
    /// split the word into a|b|c|d (3,4,11,14)bits
    fn new<F: Field>(cs: &mut Composer<F>, word: &Sha256Word) -> Result<Self, Error> {
        let value = cs.get_assignment(word.var);

        let tmp = value.into_repr();
        let value_u64 = tmp.as_ref();
        let mut value_u64 = value_u64[0].clone();

        let a_value = value_u64 & ((1 << 3) - 1);
        value_u64 >>= 3;
        let b_value = value_u64 & ((1 << 4) - 1);
        value_u64 >>= 4;
        let c_value = value_u64 & ((1 << 11) - 1);
        value_u64 >>= 11;
        let d_value = value_u64;
        value_u64 >>= 14;
        assert_eq!(value_u64, 0);

.......
```

newway55 : - Add checks :

- Needle is not empty.
- max_index() is within bounds.
- Avoid out-of-bounds memory access : starts_with() instead of comparing byte slices

Consider below fix in the `` function

```
fn find_subsequence(haystack: &[u8], needle: &[u8]) -> Option<usize> {
    if needle.is_empty() {
        return Some(0);
    }

    let max_index = haystack.len().saturating_sub(needle.len());
    for i in 0..=max_index {
        if haystack[i..].starts_with(needle) {
            return Some(i);
        }
    }

    None
}
```

Client Response

Declined,I couldn’t understand what problem was pointed out, and the auditor needs to supplement the detailed description and case, and the suggested modification 1<<3 and 8 are the same.

UPC-4:ZK prove can be replay

Category	Severity	Status	Contributor
Signature Forgery or Replay	Critical	Mitigated	zircon

Code Reference

- code/UniPass-email-circuits/crates/email-parser/src/types.rs#L75-L91

```
#[derive(Clone, Serialize, Deserialize)]
#[serde(rename_all = "camelCase")]
pub struct PublicInputs {
    #[serde(
        deserialize_with = "deserialize_hex_string",
        serialize_with = "serialize_hex_string"
    )]
    pub header_hash: Vec,
    #[serde(
        deserialize_with = "deserialize_hex_string",
        serialize_with = "serialize_hex_string"
    )]
    pub from_hash: Vec,
    pub subject: String,
    pub selector: String,
    pub sdid: String,
}
```

Description

zircon : After generating a proof for an email in the circuit, the public inputs and the proof need to be submitted to the verifier for validation. Since the public input data does not contain any one-time-use data, such as a Nullifier, the proof is reusable. An attacker, upon obtaining these public proofs and inputs, can resubmit them for verification without the need for the actual owner to send a new email.

The impact is that In a replay attack, an attacker tries to reuse (replay) a previously captured valid proof without knowing the contents of the proof. If the system does not take appropriate measures to prevent such attacks, an attacker could exploit this vulnerability to commit fraud, such as forging transactions or gaining unauthorized access by impersonating an identity.

Consider below POC

```
let res = verifier.verify(&pckey.vk, &proof, &sha256_of_srs);
println!("verify result: {}", res);
//Do something important here.
//replay again
let res2 = verifier.verify(&pckey.vk, &proof, &sha256_of_srs);
println!("verify result: {}", res2);
//Do something important again here
```

Recommendation

zircon : To prevent replay attacks, zero-knowledge proof systems typically employ one of the following methods:

1. Randomness: Introducing randomness is an effective way to prevent replay attacks. When creating a zero-knowledge proof, the prover can generate a random number as part of the proof. The verifier also needs to consider this random number when checking the proof. This way, even if an attacker captures a valid proof, they cannot replay it, as each new proof contains a different random number.
2. Context Binding: Binding the proof to a specific context can also improve the system's security. For example, the proof can include a unique identifier related to the transaction (such as a timestamp or transaction ID) to ensure it can only be used in a specific scenario. When verifying the proof, the verifier checks whether the identifier in the proof matches the expected context.
3. One-Time Tokens: In some scenarios, one-time tokens can be used to prevent replay attacks. These tokens are generated during each interaction, and the prover needs to include this token when creating the proof. The verifier, when validating the proof, needs to ensure that the token is valid and has not been used before.

Consider below fix in the `PublicInputs` struct

```
pub struct PublicInputs {
    //...snip code..
    // Add nullifier in public inputs
    nullifier: u64
}
```

Client Response

Mitigated,Public Inputs will place a nonce for verification, and DKIM has a signature that is verified externally for anti-replay and checked by a circuit other than zk.

UPC-5:Zero Bug in Plonk Verifier

Category	Severity	Status	Contributor
Signature Forgery or Replay	**Critical**	**Acknowledged**	**Ifzkoala**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/verifier.rs#L74-L563

```
    pub fn verify(&mut self, pcvk: &VKey, proof: &Proof, sha256_of_srs: &Vec) -> bool {
        log::trace!("verify time start:");
        let start = Instant::now();

        let mut z_labels = vec!["z".to_string(), "z_lookup".to_string()];
        if self.composer_config.enable_private_substring {
            z_labels.push("z_substring".to_string());
        }
        let verify_open_zeta_labels =
            gen_verify_open_zeta_labels(self.program_width, self.composer_config.enable_lookup);
        let verify_open_zeta_omega_labels = gen_verify_open_zeta_omega_labels(self.composer_config);

        let mut trans = TranscriptLibrary::new();
        // transcript SRS-hash, vkdata-hash, public-inputs
        {
            // sha256 SRS, put into the transcript
            trans.update_with_u256(sha256_of_srs);

            let pi_num = self.public_input.len() as u64;
            let v0_domainsize = self.domain.size() as u128;
            let omega = self.domain.generator();
            let mut vcomms = vec![];
            let mut g2xbytes = vec![];

            let verify_comms_labels =
                gen_verify_comms_labels(self.program_width, self.composer_config);
            for str in &verify_comms_labels {
                let comm = self.commitments[str];
                let tmp = comm.0;
                let mut bytes = [0u8; 64];
                let _ = tmp.write(bytes.as_mut());
                let mut x = [0u8; 32];
                for j in 0..32 {
                    x[32 - j - 1] = bytes[j];
                }
                let mut y = [0u8; 32];
                for j in 32..64 {
                    y[64 - j - 1] = bytes[j];
                }
                if tmp.is_zero() {
                    vcomms.push(x);
                    vcomms.push(x);
                } else {
                    vcomms.push(x);
                    vcomms.push(y);
                }
            }
            let g2x = pcvk.beta_h;
            let mut bytes = [0u8; 128];
            let _ = g2x.write(bytes.as_mut());
            let mut xc0 = [0u8; 32];
            for j in 0..32 {
                xc0[32 - j - 1] = bytes[j];
            }
            let mut xc1 = [0u8; 32];
            for j in 32..64 {
                xc1[64 - j - 1] = bytes[j];
            }
            let mut yc0 = [0u8; 32];
            for j in 64..96 {
                yc0[96 - j - 1] = bytes[j];
            }
            let mut yc1 = [0u8; 32];
            for j in 96..128 {
                yc1[128 - j - 1] = bytes[j];
            }
            g2xbytes.push(xc0);
            g2xbytes.push(xc1);
            g2xbytes.push(yc0);
            g2xbytes.push(yc1);

            let mut prehasher = Sha256::new();
            prehasher.update(pi_num.to_be_bytes());
            prehasher.update(v0_domainsize.to_be_bytes());
            prehasher.update(omega.into_repr().to_bytes_be());
            for v in vcomms {
                prehasher.update(v);
            }
            for v in g2xbytes {
                prehasher.update(v);
            }
            let result = prehasher.finalize();
```

```
                trans.update_with_u256(result);

                for pi in &self.public_input {
                    trans.update_with_fr(pi);
                }
            }

            // step 1
            for (i, ci) in proof.commitments1.iter().enumerate() {
                trans.update_with_g1::(&ci.0);
                self.commitments.insert(format!("w_{}", i), ci.clone());
            }
            let eta = trans.generate_challenge::();

            // step 2
            trans.update_with_g1::(&proof.commitment2.0);
            self.commitments
                .insert(format!("s"), proof.commitment2.clone());
            let beta = trans.generate_challenge::();
            let gamma = trans.generate_challenge::();

            // step 3
            for (ci, str) in proof.commitments3.iter().zip(&z_labels) {
                if str == "z_lookup" {
                    continue;
                }
                trans.update_with_g1::(&ci.0);
                self.commitments.insert(str.to_string(), ci.clone());
            }
            let mut beta_1 = F::zero();
            let mut gamma_1 = F::zero();
            if self.composer_config.enable_lookup {
                beta_1 = trans.generate_challenge::();
                gamma_1 = trans.generate_challenge::();
                trans.update_with_g1::(&proof.commitments3[1].0);
                self.commitments
                    .insert(format!("z_lookup"), proof.commitments3[1].clone());
            }

            let alpha = trans.generate_challenge::();

            // step 4
            for (i, ci) in proof.commitments4.iter().enumerate() {
```

```
            trans.update_with_g1::(&ci.0);
            self.commitments.insert(format!("t_{}", i), ci.clone());
        }
        let zeta = trans.generate_challenge::();

        for (ei, str) in proof.evaluations.iter().zip(&verify_open_zeta_labels) {
            trans.update_with_fr(ei);
            self.evaluations.insert(format!("{}_zeta", str), *ei);
        }
        for (ei, str) in proof
            .evaluations_alt_point
            .iter()
            .zip(&verify_open_zeta_omega_labels)
        {
            trans.update_with_fr(ei);
            self.evaluations.insert(format!("{}_zeta_omega", str), *ei);
        }
        let v = trans.generate_challenge::();

        trans.update_with_g1::(&proof.Wz_pi.0);
        trans.update_with_g1::(&proof.Wzw_pi.0);
        let u = trans.generate_challenge::();

        let lagrange_1_zeta = self.domain.evaluate_lagrange_polynomial(1, &zeta);
        // let lagrange_n_zeta = self
        //     .domain
        //     .evaluate_lagrange_polynomial(self.domain.size(), &zeta);
        let alpha_2 = alpha.square();
        // let alpha_3 = alpha * alpha_2;

        // cal r_complement
        let mut tmp = F::one();
        for i in 0..self.program_width - 1 {
            tmp *= self.evaluations[&format!("w_{}_zeta", i)]
                + beta * self.evaluations[&format!("sigma_{}_zeta", i)]
                + gamma
        }
        let mut alpha_combinator = alpha;
        let r_permu = alpha_combinator
            * (tmp
                * (self.evaluations[&format!("w_{}_zeta", self.program_width - 1)] + gamma)
                * self.evaluations["z_zeta_omega"]
                + alpha * lagrange_1_zeta);
        // permu
        alpha_combinator *= alpha_2;
        // lookup
        let r_lookup = alpha_combinator
            * (
                self.evaluations["z_lookup_zeta_omega"]
                    * gamma_1
                    * (beta_1 * self.evaluations["s_zeta_omega"] + (beta_1 + F::one()) * gamma_1)

                    + alpha * lagrange_1_zeta
                // + alpha_2 * lagrange_n_zeta
            );
        alpha_combinator *= alpha_2;
        let mut r_complement = r_permu + r_lookup;
        // range
        if self.composer_config.enable_range {
            alpha_combinator *= alpha;
        }
        // substring
        if self.composer_config.enable_private_substring {
            r_complement += alpha_combinator * (-self.evaluations["z_substring_zeta_omega"]);

            alpha_combinator *= alpha_2 * alpha;
        }
        // pubmatch
        if self.composer_config.enable_pubmatch {
            alpha_combinator *= alpha;
        }
        // mimc
        if self.composer_config.enable_mimc {
            alpha_combinator *= alpha;
        }

        let pi_poly =
            Evaluations::from_vec_and_domain(self.public_input.clone(), self.domain).interpolate();
        log::trace!("check equality...");
        let lhs = {
            let v_zeta = self.domain.evaluate_vanishing_polynomial(zeta);
            self.evaluations["t4t_zeta"] * v_zeta
        };
        let rhs = {
            let pi_zeta = pi_poly.evaluate(&zeta);

            self.evaluations["r_zeta"] - r_complement - pi_zeta
        };

        if lhs != rhs {
            log::info!("equality check fail");
            return false;
        }
        log::trace!("check equality done");

        log::trace!("pc check...");
        let zeta_n = zeta.pow(&[self.domain.size() as u64]);

        //linear combine commitments at zeta. fixed order
        let cal_combine_comm_zeta = {
            let mut cal_Wz_comm = E::G1Projective::zero();
            let mut comb = F::one();

            //cal t4t comm
            cal_Wz_comm += self.commitments["t_0"].0.into_projective();
            let mut tmp = zeta_n;
            for i in 1..self.program_width {
                cal_Wz_comm += self.commitments[&format!("t_{}", i)]
                    .0
                    .into_projective()
                    .mul(tmp.into_repr());
                tmp *= zeta_n;
            }

            comb = comb * v;

            //cal r comm
            let cal_r_comm = {
                let mut acc = E::G1Projective::zero();
                let mut alpha_combinator = alpha;

                acc += self.commitments["q_m"]
                    .0
                    .into_projective()
                    .mul((self.evaluations["w_0_zeta"] * self.evaluations["w_1_zeta"]).into_repr());
                acc += self.commitments["q_c"].0.into_projective();
                if self.composer_config.enable_q0next {
                    acc += self.commitments["q0next"]
                        .0
                        .into_projective()
                        .mul((self.evaluations["w_0_zeta_omega"]).into_repr());
                }

                for i in 0..self.program_width {
                    acc += self.commitments[&format!("q_{}", i)]
                        .0
                        .into_projective()
                        .mul(self.evaluations[&format!("w_{}_zeta", i)].into_repr());
                }

                // z
                let mut tmp = self.evaluations["w_0_zeta"] + beta * zeta + gamma;
                for i in 1..self.program_width {
                    tmp *= self.evaluations[&format!("w_{}_zeta", i)]
                        + beta * zeta * coset_generator::(i)
                        + gamma;
                }
                acc += self.commitments["z"]
                    .0
                    .into_projective()
                    .mul((alpha * (tmp + alpha * lagrange_1_zeta)).into_repr());
                // last sigma!
                let mut tmp = beta * self.evaluations["z_zeta_omega"];
                for i in 0..self.program_width - 1 {
                    tmp *= self.evaluations[&format!("w_{}_zeta", i)]
                        + beta * self.evaluations[&format!("sigma_{}_zeta", i)]
                        + gamma;
                }
                acc += self.commitments[&format!("sigma_{}", self.program_width - 1)]
                    .0
                    .into_projective()
                    .mul((-alpha * (tmp)).into_repr());
                // update alpha_comb
                alpha_combinator *= alpha_2;

                // z_lookup
                let mut twi_zeta = vec![];
                for i in 0..self.program_width {
                    twi_zeta.push(self.evaluations[&format!("w_{}_zeta", i)]);
                }
                twi_zeta.push(self.evaluations["q_table_zeta"]);
                let tmp = combine(eta, twi_zeta);
                acc += self.commitments["z_lookup"].0.into_projective().mul(
                    (alpha_combinator
                        * (
                            (self.evaluations["q_lookup_zeta"] * (tmp) + gamma_1)
                                * ((self.evaluations["table_zeta"])
                                    + beta_1 * (self.evaluations["table_zeta_omega"])
                                    + (beta_1 + F::one()) * gamma_1)
                                + alpha * lagrange_1_zeta
                            // + alpha_2 * lagrange_n_zeta
                        ))
                        .into_repr(),
                );
                // s
                acc += self.commitments["s"].0.into_projective().mul(
                    (-alpha_combinator * (self.evaluations["z_lookup_zeta_omega"] * gamma_1))
                        .into_repr(),
                );
                // update alpha_comb
                alpha_combinator *= alpha_2;

                // q_range
                if self.composer_config.enable_range {
                    let quads = {
                        let mut quads: Vec<_> = (0..self.program_width - 1)
                            .into_iter()
                            .map(|j| {
                                quad(
                                    self.evaluations[&format!("w_{}_zeta", j)],
                                    self.evaluations[&format!("w_{}_zeta", j + 1)],
                                )
                            })
                            .collect();
                        quads.push(quad(
                            self.evaluations[&format!("w_{}_zeta", self.program_width - 1)],
                            self.evaluations["w_0_zeta_omega"],
                        ));

                        quads
                    };

                    acc += self.commitments["q_range"]
                        .0
                        .into_projective()
                        .mul((alpha_combinator * (combine(eta, quads.clone()))).into_repr());

                    alpha_combinator *= alpha;
                }

                // substring:
                if self.composer_config.enable_private_substring {
                    // q_substring
                    acc += self.commitments["q_substring"].0.into_projective().mul(
                        (alpha_combinator
                            * (self.evaluations["w_2_zeta"] * self.evaluations["w_3_zeta"]
                                - self.evaluations["w_0_zeta"] * self.evaluations["w_1_zeta"]))
                            .into_repr(),
                    );

                    // q_substring_r
                    acc += self.commitments["q_substring_r"].0.into_projective().mul(
                        (alpha_combinator
                            * (alpha
                                * (self.evaluations["w_0_zeta_omega"]
                                    * self.evaluations["w_3_zeta"]
                                    * (self.evaluations["w_2_zeta"]
                                        + self.evaluations["w_0_zeta_omega"]
                                        - self.evaluations["w_0_zeta"])
                                    - self.evaluations["w_2_zeta_omega"])))
                            .into_repr(),
                    );

                    // z_substring
                    acc += self.commitments["z_substring"].0.into_projective().mul(
                        (alpha_combinator * (alpha_2 * lagrange_1_zeta - F::one())).into_repr(),
                    );

                    // update alpha_comb
                    alpha_combinator *= alpha_2 * alpha;
                }

                // pub match
                if self.composer_config.enable_pubmatch {
                    // q_q_pubmatch
                    acc += self.commitments["q_pubmatch"].0.into_projective().mul(
                        (alpha_combinator
                            * (self.evaluations["w_1_zeta"]
                                * (self.evaluations["w_0_zeta"] - self.evaluations["w_1_zeta"])))
                            .into_repr(),
                    );

                    // update alpha_comb
                    alpha_combinator *= alpha;
                }

                // q_mimc
                if self.composer_config.enable_mimc {
                    let tmp1 = self.evaluations["w_0_zeta"] + self.evaluations["w_2_zeta"];
                    let part1 = self.evaluations["w_3_zeta"] - tmp1.square();

                    acc += self.commitments["q_mimc"].0.into_projective().mul(
                        (alpha_combinator
                            * (self.evaluations["w_0_zeta_omega"]
                                - self.evaluations["w_3_zeta"].square() * tmp1
                                - self.evaluations["w_1_zeta"]
                                + eta * part1))
                            .into_repr(),
                    );

                    // update alpha_comb
                    alpha_combinator *= alpha;
                }

                acc
            };

            cal_Wz_comm += cal_r_comm.mul(comb.into_repr());
            comb = comb * v;

            //cal table comm
            let mut cal_table_comm = self.commitments["table_0"].0.into_projective();
            let mut tmp = eta;
            for i in 1..self.program_width + 1 {
                cal_table_comm += self.commitments[&format!("table_{}", i)]
                    .0
                    .into_projective()
                    .mul(tmp.into_repr());
                tmp *= eta;
            }
            let cal_table_comm = Commitment:: {
                0: cal_table_comm.into_affine(),
            };
            self.commitments.insert("table".to_string(), cal_table_comm);

            for str in verify_open_zeta_labels.iter().skip(2) {
                let tmp = self.commitments[str.as_str()];
                cal_Wz_comm += tmp.0.into_projective().mul(comb.into_repr());
                comb = comb * v;
            }

            let c = Commitment:: {
                0: cal_Wz_comm.into_affine(),
            };
            c
        };
        //combined evaluations at zeta. fixed order
        let mut Wz_poly_eval = F::zero();
        let mut comb = F::one();
        for val in &proof.evaluations {
            Wz_poly_eval += comb * val;
            comb = comb * v;
        }

        //linear combine commitments at omega_zeta. fixed order
        let cal_combine_comm_omega_zeta = {
            let mut cal_wx_comm = E::G1Projective::zero();
            let mut comb = F::one();

            for str in &verify_open_zeta_omega_labels {
                let tmp = self.commitments[str.as_str()];
                cal_wx_comm += tmp.0.into_projective().mul(comb.into_repr());
                comb = comb * v;
            }

            let c = Commitment:: {
                0: cal_wx_comm.into_affine(),
            };
            c
        };
        //combined evaluations at omega_zeta. fixed order
        let mut Wzw_poly_eval = F::zero();
        let mut comb = F::one();
        for val in &proof.evaluations_alt_point {
            Wzw_poly_eval += comb * val;
            comb = comb * v;
        }

        // batch multi_point pc check
        let result = {
            let multi_point_pcres = pcvk.batch_verify_multi_point_open_pc(
                &[cal_combine_comm_zeta, cal_combine_comm_omega_zeta],
                &[zeta, zeta * self.domain.generator()],
                &[Wz_poly_eval, Wzw_poly_eval],
                &[proof.Wz_pi, proof.Wzw_pi],
                u,
            );
            log::trace!("multi_point PC? {}", multi_point_pcres);
            multi_point_pcres
        };

        log::trace!("verify time cost: {:?} ms", start.elapsed().as_millis()); // ms
        log::trace!("verify done");

        result
    }
```

Description

Ifzkoala : The Zero Bug Attack happened in Plonk verifier which accepts proofs containing multiple elements as per the Plonk protocol. However, by manually setting two of the elements to 0, the verifier will automatically accept that proof regardless of the other elements. This allows an attacker to successfully forge a proof.

The full description of this bug is quite math heavy and dives deep into the Plonk protocol. The finder of this bug, Nguyen Thoi Minh Quan, has a great detailed description of the bug here.

Elliptic curves have what is known as a point at infinity. Let O = point at infinity and P be any point on the curve. Then O + P = P. When implementing a cryptographic protocol in code, there are different ways to express the point at inifinity. For example, sometimes the number 0 is considered the point at infinity, but other times 0 is considered as the point (0, 0), which is not the point at infinity.

Plonk proofs require a group of elements and curve points, and then will check whether these elements and points satisfy certain equations. One of the main equations to check is an elliptic curve pairing. The curve points that are of importance for this bug are [Wz]_1 and [Wzw]_1.

In UniPass, since the verifier checks the different equation as provided in the reference doc, so in the verifier, the verify function should validate the input by checking `[Wz]_1 + u * [Wzw]_1 \neq 0` `z*[Wz]_1 + uzw * [W_zw]_1 + [F]_1 - [E]_1 \neq 0`

Recommendation

lfzkoala : Add validation in the verify function and check the equations `[Wz]_1 + u * [Wzw]_1 \neq 0` `z*[Wz]_1 + uzw * [W_zw]_1 + [F]_1 - [E]_1 \neq 0`

Client Response

Acknowledged,Infinity is a valid group element even though it is not on the curve. I do not understand why we need to exclude infinity in this case. The bug discovered by N.T.M. Quan is related to a faulty implementation of the inverse function, while we perform a zero check in our implementation of `inverse()` (in the contract). Could you please provide more details or attack vectors related to this finding?

UPC-6:Infinite loop and Denial of Service (DoS) attack in find_subsequence function

Category	Severity	Status	Contributor
Language Specific	**Medium**	**Fixed**	**Ifzkoala, BradMoonUESTC**

Code Reference

- code/UniPass-email-circuits/crates/email-parser/src/parser.rs#L9
- code/UniPass-email-circuits/crates/email-parser/src/parser.rs#L11-L16
- code/UniPass-email-circuits/crates/email-parser/src/parser.rs#L21
- code/UniPass-email-circuits/crates/email-parser/src/parser.rs#L26
- code/UniPass-email-circuits/crates/email-parser/src/parser.rs#L32

```
9:fn find_subsequence(haystack: &[u8], needle: &[u8]) -> Option {

11:        .windows(needle.len())
12:        .position(|window| window == needle)
13:}
14:
15:fn parse_header(
16:    dkim_msg: &[u8],

21:    let from_index = match find_subsequence(dkim_msg, b"from:") {

26:    let from_end_index = match find_subsequence(&dkim_msg[from_index..], b"\r\n") {

32:        match find_subsequence(&dkim_msg[from_index..], format!("<{}>", from).as_bytes()) {
```

Description

Ifzkoala : In the parse_header function, the find_subsequence function is called with a dynamically constructed format string: `format!("<{}>", from).as_bytes()` The from variable is derived from the email's "from" header, which could potentially be controlled by an attacker. If the "from" header contains characters that influence the behavior of the format! macro, it could lead to unexpected results.

Also, the find_subsequence function uses a sliding window approach to find the needle in the haystack. If the input data (email_raw_data) is very large, the performance of this function could degrade, possibly leading to a DoS attack.

BradMoonUESTC : The find_subsequence function in the given code is susceptible to an infinite loop when the length of needle is 0. An attacker can exploit this vulnerability to cause a Denial of Service (DoS) attack, leading to server resource exhaustion and service unavailability.

Recommendation

lfzkoala : To address this issue, you can sanitize the from variable before passing it to format! or use a different method for searching the substring without using a format string.

To address the DoS issue, consider implementing a more efficient search algorithm (e.g., Boyer-Moore or Knuth-Morris-Pratt) or putting a limit on the input size to prevent excessive resource consumption.

BradMoonUESTC : Add a check for the length of needle in the find_subsequence function to ensure it is greater than 0. This can be done by inserting the following code snippet at the beginning of the function:

```
if needle.is_empty() {
    return None;
}
```

Client Response

Fixed

UPC-7:Unsafe Use of Indexing

Category	Severity	Status	Contributor
Language Specific	**Medium**	**Acknowledged**	**Ifzkoala**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/arithmetic.rs#L9
- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/arithmetic.rs#L17
- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/arithmetic.rs#L34

```
9:        let var_o = self.alloc_variable(self.assignments[var_l.0] + self.assignments[var_r.0]);

17:        let var_o = self.alloc_variable(self.assignments[var_l.0] - self.assignments[var_r.0]);

34:        let var_o = self.alloc_variable(self.assignments[var_l.0] * self.assignments[var_r.0]);
```

Description

Ifzkoala : In several places throughout the code, indexing is performed on `self.assignments` and `self.selectors` without checking for out-of-bounds access. For example, in the `add`, `sub`, and `mul` methods, the code directly accesses elements in `self.assignments` using `self.assignments[var_l.0]` and `self.assignments[var_r.0]`. Out-of-bounds access can lead to undefined behavior or memory corruption.

Recommendation

Ifzkoala : To mitigate this issue, you could use `get()` or `get_mut()` methods to access the elements safely and handle the potential `None` values accordingly.

Client Response

Acknowledged,The parameters are determined internally by the algorithm, so if the algorithm is correct, it must be established, and additional checks will be considered in the future.

UPC-8:Weak Sources of Randomness risk in `mimc.rs` , `types.rs` and `parameters.rs`

Category	Severity	Status	Contributor
Weak sources of Randomness	**Medium**	**Fixed**	**Ifzkoala, newway55**

Code Reference

- code/UniPass-email-circuits/crates/prover/src/parameters.rs#L8
- code/UniPass-email-circuits/crates/email-parser/src/types.rs#L95
- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/mimc.rs#L253-L296

```rust
use plonk::ark_std::rand::Rng;

        let header_hash = Sha256::digest(&private.email_header).to_vec();

        let mut x_r_i = r_data;

        for i in 0..(ROUNDS - 1) {
            let w2 = F::from_str(ROUND_KEYS[i]).unwrap_or_default();
            assert_eq!(
                w2,
                F::from(BigUint::parse_bytes(ROUND_KEYS[i].as_bytes(), 10).unwrap())
            );
            let w3 = (self.assignments[x_l_i.0] + w2).square();
            let w3_var = self.alloc(w3);
            let w2_var = self.alloc(w2);

            let index = self.insert_gate(vec![x_l_i, x_r_i, w2_var, w3_var]);
            self.selectors.get_mut("q_mimc").unwrap()[index] = F::one();

            let x_lp_value: F =
                w3.square() * (self.assignments[x_l_i.0] + w2) + self.assignments[x_r_i.0];
            let x_l_ipp = self.alloc(x_lp_value);
            let x_r_ipp = x_l_i;

            //update tmp vars
            x_l_i = x_l_ipp;
            x_r_i = x_r_ipp;
        }

        //last round
        let xl2 = self.mul(x_l_i, x_l_i);
        let xl4 = self.mul(xl2, xl2);

        let xr_out =
            self.assignments[xl4.0] * self.assignments[x_l_i.0] + self.assignments[x_r_i.0];
        let xr_out_var = self.alloc(xr_out);
        self.poly_gate(
            vec![
                (xl4, F::zero()),
                (x_l_i, F::zero()),
                (xr_out_var, -F::one()),
                (x_r_i, F::one()),
            ],
            F::one(),
            F::zero(),
        );

        (x_l_i, xr_out_var)
```

Description

lfzkoala : Ensure that the RNG used in cryptographic operations is cryptographically secure. The rand::Rng trait is not guaranteed to be cryptographically secure. Consider using a cryptographically secure RNG, like rand::rngs::OsRng.

newway55 : Current implementation of from_hash in PublicInputs uses a fixed pepper value shared across all emails, which makes it easier for attackers to precompute hashes for common values of **From**. This could potentially allow attackers to forge email signatures and bypass DKIM verification. Generating a unique pepper value for each email would mitigate this issue by introducing randomness to the hash computation.

Consider below POC contract

-This test generates two different pepper values using thread_rng() from the rand crate and uses them to compute the from_hash field for two different emails with the same from value but different peppers. It then checks that the from_hash fields are different using assert_ne!().

```
#[cfg(test)]
mod tests {
    use super::*;
    use rand::{thread_rng, Rng};

    #[test]
    fn test_from_hash_random_pepper() {
        let email_header = b"from:test@example.com\r\nsubject:Test\r\n".to_vec();
        let from = "test@example.com";
        let subject_index = 15;
        let subject_right_index = 18;

        // Generate two different pepper values
        let pepper1 = thread_rng().gen::<u64>();
        let pepper2 = thread_rng().gen::<u64>();
        assert_ne!(pepper1, pepper2);

        // Compute public inputs for two emails with the same from value but different peppers
        let private1 = PrivateInputs {
            email_header: email_header.clone(),
            from_pepper: pepper1.to_le_bytes().to_vec(),
            from_index: 5,
            from_left_index: 5,
            from_right_index: 19,
            subject_index,
            subject_right_index,
            dkim_header_index: 0,
            selector_index: 0,
            selector_right_index: 0,
            sdid_index: 0,
            sdid_right_index: 0,
        };
        let public1 = PublicInputs::from(&private1);

        let private2 = PrivateInputs {
            email_header: email_header.clone(),
            from_pepper: pepper2.to_le_bytes().to_vec(),
            from_index: 5,
            from_left_index: 5,
            from_right_index: 19,
            subject_index,
            subject_right_index,
dkim_header_index: 0,
            selector_index: 0,
            selector_right_index: 0,
            sdid_index: 0,
            sdid_right_index: 0,
        };
        let public2 = PublicInputs::from(&private2);

        // Check that the from_hash fields are different
        assert_ne!(public1.from_hash, public2.from_hash);
    }
}
```

newway55 : We have a predicability issue here using same keys many times. This can lead to attacks.

- Using randomness to generate round keys adds an additional layer of security and unpredictability to the system. This reduces the chances of attacks that rely on guessing the round keys or exploiting weaknesses in hardcoded keys.

Consider below POC contract

```rust
use crate::Composer;

#[test]
fn test_mimc_round_keys_randomness() {
    let mut composer = Composer::<Bls12>::new();

    let l_data = composer.alloc(Fr::from(1u32));
    let r_data = composer.alloc(Fr::from(2u32));

    let (out_l, out_r) = composer.mimc_feistel(l_data, r_data);

    assert_eq!(composer.is_satisfied(), Ok(()));

    let round_keys: Vec<String> = composer.round_keys.iter().map(|rk| rk.to_string()).collect();

    // Check that all round keys are different
    for i in 0..round_keys.len() {
        for j in i + 1..round_keys.len() {
            assert_ne!(round_keys[i], round_keys[j]);
        }
    }
}
```

Recommendation

lfzkoala : Try to use cryptographically secure randomness resources such as rngs::OsRng. Also notice that OsRng is actually less efficient since it needs to fetch randomness from the operating system. You should balance efficiency and security and choose which randomness source should be used. From security perspective, this issue is critical
newway55 : Use the Checks-Effects-Interactions best practice and make all state changes before calling external contracts. Also, consider using function modifiers such as `nonReentrant` from Reentrancy Guard to prevent re-entrancy at the contract level.

Consider below fix in the `` function

- Generate a 32-byte random value using the rand crate's thread_rng() function, and use it as the pepper value.

```
use rand::Rng;

impl From<&PrivateInputs> for PublicInputs {
    fn from(private: &PrivateInputs) -> PublicInputs {
        let header_hash = Sha256::digest(&private.email_header).to_vec();
        let pepper = rand::thread_rng().gen::<[u8; 32]>();
        let from_hash = {
            let mut hasher = Sha256::default();
            let from_bytes = &private.email_header
                [private.from_left_index as usize..private.from_right_index as usize + 1];
            hasher.update(from_bytes);
            hasher.update(&pepper);
            hasher.finalize().to_vec()
        };
        PublicInputs {
            header_hash,
            from_hash,
            subject: String::from_utf8_lossy(
                &private.email_header
                    [private.subject_index as usize + 8..private.subject_right_index as usize],
            )
            .to_string(),
            selector: String::from_utf8_lossy(
                &private.email_header
                    [private.selector_index as usize..private.selector_right_index as usize],
            )
            .to_string(),
            sdid: String::from_utf8_lossy(
                &private.email_header
                    [private.sdid_index as usize..private.sdid_right_index as usize],
            )
            .to_string(),
        }
    }
}
```

newway55 : - Generate the round keys for the MiMC operation using randomness , rather than using hardcoded round keys.

Consider below fix in the `mimc_feistel()` function

```rust
use rand::Rng;

impl<F: Field> Composer<F> {
    const ROUNDS: usize = 220;
    const K: u64 = 0;

    /// MiMC (follow circomlib's MiMCsponge)
    /// rounds = 220
    /// k === 0
    /// x_l[i+1] = (k + x_l[i] + c[i])**5 + x_r[i]
    /// x_r[i+1] = x_l[i]
    /// if last round:
    /// x_l[i+1] = x_l[i]
    /// x_r[i+1] = x_r[i] + (k + x_l[i] + c[i])**5
    pub fn mimc_feistel(&mut self, l_data: Variable, r_data: Variable) -> (Variable, Variable) {
        if !self.switches.enable_mimc {
            self.switches.enable_mimc = true;
        }
        if !self.selectors.contains_key("q_mimc") {
            let current_index = self.size();
            self.selectors
                .insert("q_mimc".to_string(), vec![F::zero(); current_index]);
        }
        assert!(!self.is_finalized);
        assert!(self.program_width >= 4);

        let mut x_l_i = l_data;
        let mut x_r_i = r_data;

        let mut rng = rand::thread_rng();

        for i in 0..(Self::ROUNDS - 1) {
            let w2 = F::from(rng.gen::<u64>()) + F::from(Self::K);
            let w3 = (self.assignments[x_l_i.0] + w2).square();
            let w3_var = self.alloc(w3);
            let w2_var = self.alloc(w2);

            let index = self.insert_gate(vec![x_l_i, x_r_i, w2_var, w3_var]);
            self.selectors.get_mut("q_mimc").unwrap()[index] = F::one();

            let x_lp_value: F =
                w3.square() * (self.assignments[x_l_i.0] + w2) + self.assignments[x_r_i.
0];
            let x_l_ipp = self.alloc(x_lp_value);
            let x_r_ipp = x_l_i;

            //update tmp vars
            x_l_i = x_l_ipp;
            x_r_i = x_r_ipp;
        }

        //last round
        let xl2 = self.mul(x_l_i, x_l_i);
        let xl4 = self.mul(xl2, xl2);

        let xr_out =
            self.assignments[xl4.0] * self.assignments[x_l_i.0] + self.assignments[x_r_i.0];
        let xr_out_var = self.alloc(xr_out);
        self.poly_gate(
            vec![
                (xl4, F::zero()),
                (x_l_i, F::zero()),
                (xr_out_var, -F::one()),
                (x_r_i, F::one()),
            ],
            F::one(),
            F::zero(),
        );

        (x_l_i, xr_out_var)
    }
```

Client Response

Fixed,The unused MiMC circuit is removed.The same randomnesses are only used for testing and will be replaced with real random numbers in production.

UPC-9:Wires should be appended to the root position in `Composer::finalize`

Category	Severity	Status	Contributor
Logical	Medium	Fixed	alansh

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/mod.rs#L330

```
330:            self.epicycles[var.0].push(Wire::new(0, i));
```

Description

alansh :

```
        for (i, var) in self.public_input.iter().enumerate() {
            self.epicycles[var.0].push(Wire::new(0, i));
            for col in 1..self.program_width {
                // other witnesses just set 0 at PI' row
                self.epicycles[Self::null().0].push(Wire::new(col, i));
            }
        }
```

Here `Wire::new(0, i)` is directly appended into the slot for `var`. But in fact should be appended into the root slot. Or should at least ensure that the root slot of `var` is `var` itself by:

```
assert!(self.eq_constraints.find(var.0) == var.0);
```

Theoretically there's nothing stopping a public input to appear in a copy constraint.

A better way is to put `self.handle_eq_constraints();` behind the modification of `self.epicycles`.

Recommendation

alansh : Apply one of the above two fixes.

Client Response

Fixed

UPC-10:Avoid using assert! in production code

Category	Severity	Status	Contributor
Code Style	**Low**	**Mitigated**	**Ifzkoala**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/range.rs#L18

```
18:        assert!(!self.is_finalized);
```

Description

Ifzkoala : The code uses `assert!` to check various conditions, which will panic and potentially crash the program if the condition is not met. Consider using custom error types and returning a `Result` instead, which allows for more graceful error handling and provides better information about the error.

Recommendation

Ifzkoala : Double-check the code and consistently use custom error types or returning a `Result` instead. For locations below I just give an example path.

Client Response

Mitigated,Where assert is used, the algorithm is sure that there will be no errors, and this will not be affected by user input. If there is a problem with the algorithm level, it is better to panic directly.

UPC-11:Base58 is malleable

Category	Severity	Status	Contributor
Logical	**Low**	**Declined**	**zircon**

Code Reference

- code/UniPass-email-circuits/crates/prover/src/circuit/base64.rs#L60-L186

```
pub fn base64url_encode_gadget(
    cs: &mut Composer,
    input_messages: &[Variable],
    num_limit: usize,
) -> Result, Error> {
    assert_eq!(input_messages.len(), num_limit);
    assert!(num_limit % 3 == 0);
    let mut output = vec![];
    let base64url_index = cs.add_table(new_baseurl_chars_table());
    assert!(base64url_index != 0);

    let spread2_index = cs.get_table_index(format!("spread_2bits"));
    assert!(spread2_index != 0);
    let spread_4_index = cs.get_table_index(format!("spread_5bits_4bits"));
    assert!(spread_4_index != 0);
    let spread_6_index = cs.get_table_index(format!("spread_7bits_6bits"));
    assert!(spread_6_index != 0);

    for chars in input_messages.chunks(3) {
        let chars_value = cs.get_assignments(chars);
        let char1 = {
            let tmp = chars_value[0].into_repr();
            tmp.as_ref()[0].clone()
        };
        let char2 = {
            let tmp = chars_value[1].into_repr();
            tmp.as_ref()[0].clone()
        };
        let char3 = {
            let tmp = chars_value[2].into_repr();
            tmp.as_ref()[0].clone()
        };

        let char1_1 = char1 >> 2;
        let char1_1_var = cs.alloc(F::from(char1_1));
        let _ = cs.read_from_table(spread_6_index, vec![char1_1_var, Composer::::null()])?;

        let char1_2 = char1 & 0x3;
        let char1_2_var = cs.alloc(F::from(char1_2));
        let _ = cs.read_from_table(spread2_index, vec![char1_2_var])?;

        let char2_1 = char2 >> 4;
        let char2_1_var = cs.alloc(F::from(char2_1));
        let _ = cs.read_from_table(spread_4_index, vec![char2_1_var, Composer::::null()])?;

        let char2_2 = char2 & 0xf;
        let char2_2_var = cs.alloc(F::from(char2_2));
        let _ = cs.read_from_table(spread_4_index, vec![char2_2_var, Composer::::null()])?;

        let char3_1 = char3 >> 6;
        let char3_1_var = cs.alloc(F::from(char3_1));
        let _ = cs.read_from_table(spread2_index, vec![char3_1_var])?;

        let char3_2 = char3 & 0x3f;
        let char3_2_var = cs.alloc(F::from(char3_2));
        let _ = cs.read_from_table(spread_6_index, vec![char3_2_var, Composer::::null()])?;

        cs.poly_gate(
            vec![
                (chars[0], -F::one()),
                (char1_1_var, F::from(1u64 << 2)),
                (char1_2_var, F::one()),
            ],
            F::zero(),
            F::zero(),
        );

        cs.poly_gate(
            vec![
                (chars[1], -F::one()),
                (char2_1_var, F::from(1u64 << 4)),
                (char2_2_var, F::one()),
            ],
            F::zero(),
            F::zero(),
        );

        cs.poly_gate(
            vec![
                (chars[2], -F::one()),
                (char3_1_var, F::from(1u64 << 6)),
                (char3_2_var, F::one()),
            ],
            F::zero(),
            F::zero(),
        );

        let out1_var = char1_1_var;
        let out2_var = cs.alloc(F::from(char1_2 * (1 << 4) + char2_1));
        let out3_var = cs.alloc(F::from(char2_2 * (1 << 2) + char3_1));
        let out4_var = char3_2_var;

        cs.poly_gate(
            vec![
                (out2_var, -F::one()),
                (char1_2_var, F::from(1u64 << 4)),
                (char2_1_var, F::one()),
            ],
            F::zero(),
            F::zero(),
        );

        cs.poly_gate(
            vec![
                (out3_var, -F::one()),
                (char2_2_var, F::from(1u64 << 2)),
                (char3_1_var, F::one()),
            ],
            F::zero(),
            F::zero(),
        );

        let output_var1 = cs.read_from_table(base64url_index, vec![out1_var])?;
        output.push(output_var1[0]);

        let output_var2 = cs.read_from_table(base64url_index, vec![out2_var])?;
        output.push(output_var2[0]);

        let output_var3 = cs.read_from_table(base64url_index, vec![out3_var])?;
        output.push(output_var3[0]);

        let output_var4 = cs.read_from_table(base64url_index, vec![out4_var])?;
        output.push(output_var4[0]);
    }

    return Ok(output);
}
```

Description

zircon : In the OpenID circuit, the `base64url_encode_gadget` function is called to encode the header and payload, where base58 encoding is used, Base58 is malleable.

The impact is that base58 can lead to data distortion during decoding.

Consider below POC

```
import base64
s1=base64.b64decode("00==")
s2=base64.b64decode("03==")
s3=base64.b64decode("0w==")
b1 = base64.b64encode(s1)
b2 = base64.b64encode(s2)
b3 = base64.b64encode(s3)
print (s1,s2,s3)
print (b1,b2,b3)
print (s1==s3)
print (s2==s3)
```

The result is blow:

```
b'\xd3' b'\xd3' b'\xd3'
b'0w==' b'0w==' b'0w=='
True
True
```

Recommendation

zircon : During base58 encoding, check if extensible characters such as `00==, 03==, 0w==` are generated. If present, roll back the operation.

Client Response

Declined,Base64Url is used here to encode and decode the idToken of openID, which is provided by service providers (such as Google), and we cannot adjust it.

UPC-12:Denominator may not be always invertible

Category	Severity	Status	Contributor
Language Specific	**Low**	**Fixed**	**Ifzkoala**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/utils.rs#L152
- code/UniPass-email-circuits/crates/plookup-sha256/src/utils.rs#L343

```
152:            .inverse()

343:        let k2_inv = k2.inverse().unwrap();
```

Description

Ifzkoala : The `inverse()` method called in `evaluate_lagrange_polynomial` assumes that the denominator is invertible, which may not be true for all inputs. An attacker could potentially provide a point that results in a non-invertible denominator and cause the program to panic or return incorrect results. It would be better to handle this case explicitly and return an error or other appropriate behavior.

Recommendation

Ifzkoala : Do input validation to handle this case explicitly and return an error or other appropriate behavior.

Client Response

Fixed

UPC-13:Insecure Deserialization

Category	Severity	Status	Contributor
Logical	**Low**	**Acknowledged**	**Ifzkoala, BradMoonUESTC**

Code Reference

- code/UniPass-email-circuits/crates/prover/src/types.rs#L1
- code/UniPass-email-circuits/crates/prover/src/parameters/iden3.rs#L321-L327

```
1:use email_parser::types::{deserialize_hex_string, serialize_hex_string};

321:fn deserialize_g1_vec(reader: &mut R, n_vars: u32) -> IoResult> {
322:    (0..n_vars).map(|_| deserialize_g1(reader)).collect()
323:}
324:
325:fn deserialize_g2_vec(reader: &mut R, n_vars: u32) -> IoResult> {
326:    (0..n_vars).map(|_| deserialize_g2(reader)).collect()
327:}
```

Description

Ifzkoala : The `deserialize_hex_string` function deserializes a hex string into a `Vec<u8>`. This function trusts that the input string is a valid hex string, and if it is not, an error is returned. While the error is caught and handled, it's important to be cautious when deserializing untrusted data as this can lead to potential security vulnerabilities.
BradMoonUESTC : In the deserialize_g1_vec and deserialize_g2_vec functions, the n_vars parameter is not validated. If the value of n_vars is extremely large, it could lead to memory exhaustion or other issues.

Recommendation

Ifzkoala : To mitigate this issue, consider using a secure deserialization mechanism or validate the input string before processing it.
BradMoonUESTC : To avoid this situation, add validation for the n_vars parameter in these functions, such as ensuring its value is within a reasonable range.

Client Response

Acknowledged,This check is located in the process of importing CRS parameters in snarkJs. This process is only operated once when the circuit deployment is started, so the impact is very small.

UPC-14:Integer Overflow and Underflow risk in `arithmetic.rs`

Category	Severity	Status	Contributor
Integer Overflow and Underflow	**Low**	**Declined**	**newway55**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/arithmetic.rs#L8
- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/arithmetic.rs#L78

```
8:    pub fn add(&mut self, var_l: Variable, var_r: Variable) -> Variable {

78:        assert!(self.selectors.contains_key("q0next"));
```

Description

newway55 : The lack of bounds checks and overflow/underflow checks can lead to incorrect assignments of values to Variables, which can in turn lead to incorrect results and potentially even vulnerabilities in a cryptographic context.

- Impact : an attacker may be able to manipulate the input values which will make overflow or underflow, potentially allowing them to bypass security checks or gain unauthorized access to sensitive information.

Consider below POC contract

```rust
#[test]
fn test_add_overflow() {
    let mut composer = Composer::<u32>::new(10);
    let a = composer.alloc_variable(4294967295); // max value for u32
    let b = composer.alloc_variable(1);
    let c = composer.add(a, b); // this should overflow
    let result = composer.solve();
    assert_eq!(result.get(c), Some(0));
}

#[test]
fn test_sub_underflow() {
    let mut composer = Composer::<u32>::new(10);
    let a = composer.alloc_variable(0);
    let b = composer.alloc_variable(1);
    let c = composer.sub(a, b); // this should underflow
    let result = composer.solve();
    assert_eq!(result.get(c), Some(4294967295)); // max value for u32
}
```

newway55 : The problem in poly_gate_with_next is that it does not check the length of the next_wire vector, assuming that it only contains one tuple. This can result in unexpected behavior and incorrect results when the vector contains more than one tuple, potentially leading to vulnerabilities in a cryptographic context.

Consider below POC contract

```
#[test]
fn test_poly_gate_with_next_wire_vulnerability() {
    use ark_ff::Zero;
    use ark_relations::{
        lc,
        r1cs::{ConstraintSynthesizer, ConstraintSystem, SynthesisError},
    };

    // Define a struct for the R1CS example
    #[derive(Clone)]
    struct TestCircuit<F: Field> {
        a: Option<F>,
        b: Option<F>,
    }

    // Implement the `ConstraintSynthesizer` trait for the struct
    impl<F: Field> ConstraintSynthesizer<F> for TestCircuit<F> {
        fn generate_constraints(self, cs: ConstraintSystem<F>) -> Result<(), SynthesisError> {
            let mut composer = Composer::new(cs);

            let a_var = composer.alloc(|| self.a.ok_or(SynthesisError::AssignmentMissing))?;
            let b_var = composer.alloc(|| self.b.ok_or(SynthesisError::AssignmentMissing))?;
            let c_var = composer.alloc(|| self.a.ok_or(SynthesisError::AssignmentMissing))?;

            composer.poly_gate_with_next(
                vec![
                    (a_var, F::one()),
                    (b_var, F::one()),
                    (c_var, F::one())
                ],
                F::one(),
                F::zero(),
                vec![
                    // This wire has an index of 2, but it's not present in the list of wires
                    // passed to `poly_gate_with_next`, so it will be ignored.
                    (composer.alloc(|| Some(F::one()))?, F::one()),
                    (composer.alloc(|| Some(F::zero()))?, F::zero())
]
            );

            // Enforce constraints
            composer.constrain(c_var);

            Ok(())
        }
    }

    // Create a new `ConstraintSystem`
    let cs = ConstraintSystem::<Fr>::new_ref();

    // Assign values to the inputs
    let input = TestCircuit {
        a: Some(Fr::one()),
        b: Some(Fr::one()),
    };

    // Synthesize the constraints for the circuit
    let _ = input.generate_constraints(cs.clone()).unwrap();

    // Check if the constraint system is satisfied
    assert!(cs.is_satisfied().unwrap());
}
```

Recommendation

newway55 : - This code uses the get method to safely access the assignments vector and return a default value (in this case, zero) if the index is out of bounds.

- You can do the same for `mul and sub` functions.

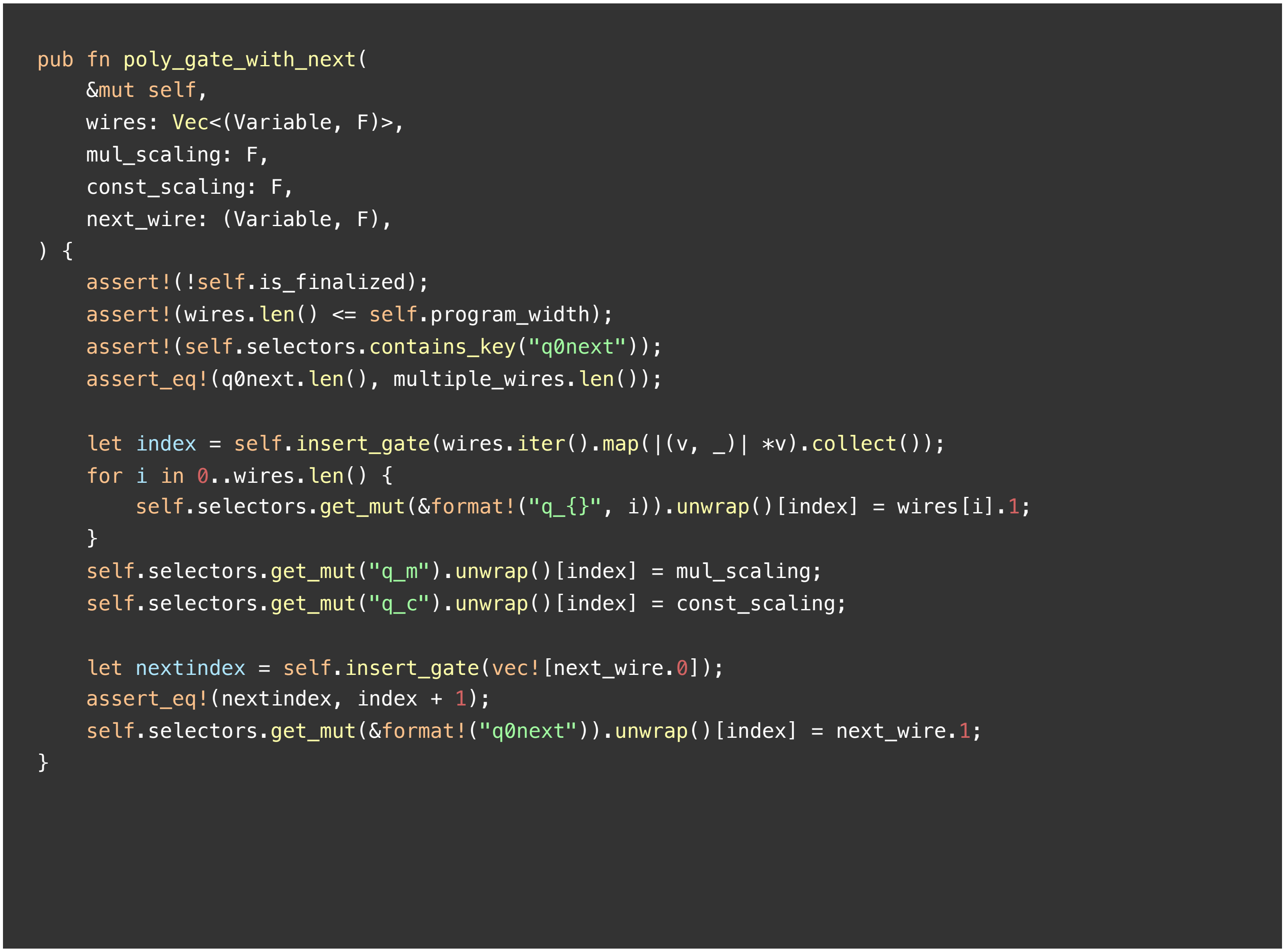
Consider below fix in the `add` function of `Composer` implementation

```
Copy code
pub fn add(&mut self, var_l: Variable, var_r: Variable) -> Variable {
    let l = self.assignments.get(var_l.0).cloned().unwrap_or_default();
    let r = self.assignments.get(var_r.0).cloned().unwrap_or_default();
    let var_o = self.alloc_variable(l + r);
    self.add_gate(var_l, var_r, var_o);

    var_o
}
```

newway55 : Adding this : `assert_eq!(q0next.len(), multiple_wires.len());`

Consider below fix in the `poly_gate_with_next` function



```rust
pub fn poly_gate_with_next(
    &mut self,
    wires: Vec<(Variable, F)>,
    mul_scaling: F,
    const_scaling: F,
    next_wire: (Variable, F),
) {
    assert!(!self.is_finalized);
    assert!(wires.len() <= self.program_width);
    assert!(self.selectors.contains_key("q0next"));
    assert_eq!(q0next.len(), multiple_wires.len());

    let index = self.insert_gate(wires.iter().map(|(v, _)| *v).collect());
    for i in 0..wires.len() {
        self.selectors.get_mut(&format!("q_{}", i)).unwrap()[index] = wires[i].1;
    }
    self.selectors.get_mut("q_m").unwrap()[index] = mul_scaling;
    self.selectors.get_mut("q_c").unwrap()[index] = const_scaling;

    let nextindex = self.insert_gate(vec![next_wire.0]);
    assert_eq!(nextindex, index + 1);
    self.selectors.get_mut(&format!("q0next")).unwrap()[index] = next_wire.1;
}
```

Client Response

Declined,Composer only accepts finite fields, there is no overflow issue.

UPC-15:Lack Input Validation

Category	Severity	Status	Contributor
Code Style	Low	Fixed	Ifzkoala, BradMoonUESTC

Code Reference

- code/UniPass-email-circuits/crates/prover/src/circuit/circuit_1024.rs#L28-L82
- code/UniPass-email-circuits/crates/plookup-sha256/src/sha256.rs#L151

```
            [private_inputs.from_left_index..private_inputs.from_right_index + 1]
            .to_vec();
        let mut email_addr_pepper_bytes = email_addr_bytes.clone();
        email_addr_pepper_bytes.append(&mut private_inputs.from_pepper);

        let email_header_bytes = private_inputs.email_header.clone();

        // set any byte of "pub match string" to "0" is OK.
        // (you can only remain bytes for format checking, set all other bytes to 0)
        let mut email_header_pub_match = email_header_bytes.clone();

        for i in 0..email_header_pub_match.len() {
            if private_inputs.from_index == 0 {
                if i >= private_inputs.from_index && i < private_inputs.from_left_index {
                    continue;
                }
            } else {
                if i >= private_inputs.from_index - 2 && i < private_inputs.from_left_index {
                    continue;
                }
            }

            if i == private_inputs.from_right_index + 1 {
                continue;
            }

            if private_inputs.subject_index == 0 {
                if i >= private_inputs.subject_index && i < private_inputs.subject_right_index {
                    continue;
                }
            } else {
                if i >= private_inputs.subject_index - 2 && i < private_inputs.subject_right_index {
                    continue;
                }
            }

            if i == private_inputs.dkim_header_index - 2 {
                break;
            }

            email_header_pub_match[i] = 0;
        }

        let from_len =
            (private_inputs.from_right_index - private_inputs.from_left_index + 1) as u32;
        let from_left_index = private_inputs.from_left_index as u32;

        Ok(Self {
            email_header_bytes,
            email_addr_pepper_bytes,
            email_header_pub_match,
            from_left_index,
            from_len,
        })
    }

    pub fn new_from_32bits_var(
```

Description

lfzkoala : In various functions such as `new_from_32bits_var`, `new_from_vars`, and `new_from_8bits`, .etc, there is no validation for input arguments like `var`, `hvar`, `lvar`, `char1`, `char2`, `char3`, and `char4`. (Just take examples here, actually lots of functions don't have input validation). These variables should be validated for their type and range to ensure they do not cause unexpected behavior or security vulnerabilities in the code.

BradMoonUESTC : In the Email1024CircuitInput::new() function, there is a potential risk for an index out-of-bounds error when slicing the email header:

```
let email_addr_bytes = private_inputs.email_header
    [private_inputs.from_left_index..private_inputs.from_right_index + 1]
    .to_vec();
```

If private_inputs.from_left_index or private_inputs.from_right_index are not properly validated and are outside the bounds of the email_header vector, this code may panic or cause unintended behavior.

BradMoonUESTC : The Email1024CircuitInput::new() function does not perform sufficient validation on its input parameters. This can lead to potential issues, such as index out-of-bounds errors, incorrect processing of input data, or other unintended behavior.

For example, the function does not validate the relationship between private_inputs.from_left_index and private_inputs.from_right_index, which can lead to an incorrect range being used when slicing the email header:

```
let email_addr_bytes = private_inputs.email_header
    [private_inputs.from_left_index..private_inputs.from_right_index + 1]
    .to_vec();
```

Recommendation

lfzkoala : I recommend double-checking the code, especially each function to guarantee that the inputs are validated. Some functions may really don't need input validation but the code writer should double-check. For the locations I just give a path for example, actually lots of functions lack input validation.
BradMoonUESTC : Before performing the slice operation, add checks to ensure that private_inputs.from_left_index and private_inputs.from_right_index are within the bounds of the email_header vector. You can use the checked_add and checked_sub methods to perform safe arithmetic operations and avoid potential panics:

```
if private_inputs.from_left_index < private_inputs.email_header.len()
    && private_inputs.from_right_index < private_inputs.email_header.len()
{
    let email_addr_bytes = private_inputs.email_header
        [private_inputs.from_left_index..private_inputs.from_right_index + 1]
        .to_vec();
} else {
    // Handle the error case, e.g., return an Err with a descriptive error message.
}
```

BradMoonUESTC : To ensure correct and safe handling of inputs, add input validation checks in the Email1024CircuitInput::new() function. For instance, you can check if private_inputs.from_left_index is less than or equal to private_inputs.from_right_index to guarantee a valid range:

```
if private_inputs.from_left_index <= private_inputs.from_right_index {
    // Proceed with the rest of the function
} else {
    // Handle the error case, e.g., return an Err with a descriptive error message.
}
```

Client Response

Fixed

UPC-16:Logical risk in `Prover implementation` for `insert` functions

Category	Severity	Status	Contributor
Logical	Low	Mitigated	newway55

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/prover/mod.rs#L60-L70

```
    pub fn insert(&mut self, label: &str, domain_values: Vec) {
        let (domain_values, coset_values, polynomial) =
            interpolate_and_coset_fft(domain_values, self.domain, self.coset);

        self.domain_values.insert(label.to_string(), domain_values);
        self.coset_values.insert(label.to_string(), coset_values);
        self.polynomials.insert(label.to_string(), polynomial);
    }

    // will blind the poly according to open_num
    pub fn insert_with_blind(
```

Description

newway55 : The insert() and insert_with_blind() methods in the given code are used to preprocess input vectors into polynomials, coset FFT, etc. However, these methods do not validate the size of the domain_values vector to ensure that it has the correct size for the domain field.

This can have a potential logical vulnerability, where the polynomial interpolation process may fail due to the size of the domain_values vector being different from the domain field size. This can cause the prover to generate incorrect or invalid proofs, This can result in the verification of invalid transactions or system operations.

Consider below POC contract

Example : if the domain field size is expected to be 2^16 but the domain_values vector has a size of 2^15, then the polynomial interpolation process may fail, leading to invalid proofs.

Test to make sure that the validation correctly detects this issue and prevents the invalid proof.

```
use rand::Rng;
use crate::{Field, Domain, Prover, padding_and_interpolate};

// Test that the prover generates valid proof even when the domain_values vector has a size smaller than the domain field size
#[test]
fn test_insert_with_small_domain_values() {
    // Setup domain and coset
    let domain_size = 65536;
    let domain = Domain::new_for_size(domain_size).unwrap();
    let coset = domain.coset().unwrap();

    // Setup prover
    let mut prover = Prover::new(domain, coset, 5);

    // Generate random domain values with size 2^15
    let mut rng = rand::thread_rng();
    let domain_values = (0..32768).map(|_| Field::random(&mut rng)).collect();

    // Call insert_with_blind with domain_values vector
    prover.insert_with_blind("witness", domain_values.clone(), 0, &mut rng);

    // Check that the polynomial interpolation process works and the prover generates valid proofs
    let result = padding_and_interpolate(domain_values, prover.domain);
    assert!(result.is_ok());
}

}
```

Recommendation

newway55 : - Validate the size of the domain_values vector to ensure that it matches the domain field size before processing it. This will ensure that the polynomial interpolation process works correctly, and the generated proofs are valid.

Consider below fix in the `Prover implementation`

```
pub struct Prover<F: Field, D: Domain<F>, E: PairingEngine> {
    domain_values: Map<String, Vec<F>>, //already padded
    coset_values: Map<String, Vec<F>>,  //coset fft
    polynomials: Map<String, DensePolynomial<F>>,

    challenges: Map<String, F>,
    pub evaluations: Map<String, F>,
    pub commitments: Map<String, Commitment<E>>,

    pub domain: D,
    pub coset: D,
    pub program_width: usize,

    pub composer_config: ComposerConfig,
}

impl<'a, F: Field, D: Domain<F>, E: PairingEngine> Prover<F, D, E> {
    pub fn domain_size(&self) -> usize {
        self.domain.size()
    }

    pub fn coset_size(&self) -> usize {
        self.coset.size()
    }

    // Preprocessing vectors into polynomials, etc
    pub fn insert(&mut self, label: &str, domain_values: Vec<F>) {
        assert_eq!(domain_values.len(), self.domain_size(), "Domain values vector has incorrect size
for the domain field");
        let (domain_values, coset_values, polynomial) =
            interpolate_and_coset_fft(domain_values, self.domain, self.coset);

        self.domain_values.insert(label.to_string(), domain_values);
        self.coset_values.insert(label.to_string(), coset_values);
        self.polynomials.insert(label.to_string(), polynomial);
    }

    // will blind the poly according to open_num
    pub fn insert_with_blind<R: RngCore>(
        &mut self,
        label: &str,
        domain_values: Vec<F>,
        open_num: usize,
rng: &mut R,
    ) {
        assert_eq!(domain_values.len(), self.domain_size(), "Domain values vector has incorrect size for the domain field");
        let (domainvalues, poly) = padding_and_interpolate(domain_values, self.domain);
        let (coset, blindpoly) = blind_and_coset_fft(poly, self.domain, self.coset, open_num, rng);

        self.domain_values.insert(label.to_string(), domainvalues);
        self.coset_values.insert(label.to_string(), coset);
        self.polynomials.insert(label.to_string(), blindpoly);
    }
}
```

Client Response

Mitigated,The algorithm is guaranteed to be consistent, and Composer will check the user-operable input.

UPC-17:Public Mutable References

Category	Severity	Status	Contributor
Logical	**Low**	**Acknowledged**	**BradMoonUESTC**

Code Reference

- code/UniPass-email-circuits/crates/prover/src/parameters/iden3.rs#L42-L52

```
42:pub struct BinFile<'a, R> {
43:    #[allow(dead_code)]
44:    ftype: String,
45:    #[allow(dead_code)]
46:    version: u32,
47:    sections: HashMap>,
48:    #[serde(skip)]
49:    reader: &'a mut R,
50:}
51:
52:impl<'a, R: Read + Seek> BinFile<'a, R> {
```

Description

BradMoonUESTC : The BinFile struct contains a public mutable reference:

```
pub struct BinFile<'a> {
    pub reader: &'a mut dyn Read,
}
```

Having a public mutable reference could lead to data races and other safety issues. In a concurrent environment, multiple threads might access and modify the shared mutable reference, causing unpredictable behavior or crashes.

Recommendation

BradMoonUESTC : Consider changing the mutable reference to internal mutability (e.g., using RefCell) or controlling access to the reader in another way. For example, you can encapsulate the mutable reference within the BinFile struct and provide methods to access it safely:

```
use std::io::Read;
use std::cell::RefCell;

pub struct BinFile<'a> {
    reader: RefCell<&'a mut dyn Read>,
}

impl<'a> BinFile<'a> {
    pub fn new(reader: &'a mut dyn Read) -> Self {
        BinFile {
            reader: RefCell::new(reader),
        }
    }

    // Provide methods to access the reader safely
    // ...
}
```

By using RefCell, you can ensure that only one mutable reference to the reader exists at a time, preventing data races and improving overall safety.

Client Response

Acknowledged,This check is located in the process of importing CRS parameters in snarkJs. This process is only operated once when the circuit deployment is started, so the impact is very small.

UPC-18:Unsafe type conversion in prepare_generic_params

Category	Severity	Status	Contributor
Integer Overflow and Underflow	**Low**	**Acknowledged**	**BradMoonUESTC**

Code Reference

- code/UniPass-email-circuits/crates/prover/src/parameters.rs#L37-L39

```
37:pub fn prepare_generic_params(max_degree: usize, rng: &mut impl Rng) -> PCKey {
38:    PCKey::::setup(max_degree, rng)
39:}
```

Description

BradMoonUESTC : In the `prepare_generic_params` function, the `max_degree` parameter has a usize type. However, when calling this function, a value that is too large might be passed, which could lead to overflow issues.

Recommendation

BradMoonUESTC : Ensure that the max_degree value is within a reasonable range to avoid overflow. You can add a check to validate the value of max_degree before proceeding with the rest of the function:

```
if max_degree > MAX_ALLOWED_DEGREE {
    return Err("The max_degree value is too large.");
}
```

Define a constant MAX_ALLOWED_DEGREE that represents the maximum safe value for max_degree.

Client Response

Acknowledged,This method may only be used when initially generating parameters, so it has little impact.

UPC-19:Use of String::from_utf8_lossy for potentially non-UTF8 data

Category	Severity	Status	Contributor
Language Specific	Low	Fixed	Ifzkoala

Code Reference

- code/UniPass-email-circuits/crates/email-parser/src/types.rs#L110
- code/UniPass-email-circuits/crates/email-parser/src/types.rs#L115
- code/UniPass-email-circuits/crates/email-parser/src/types.rs#L120

```
110:            subject: String::from_utf8_lossy(

115:            selector: String::from_utf8_lossy(

120:            sdid: String::from_utf8_lossy(
```

Description

Ifzkoala : The String::from_utf8_lossy function is used to convert byte slices into strings in multiple places within the impl From<&PrivateInputs> for PublicInputs block. This function replaces invalid UTF-8 sequences with the Unicode replacement character (U+FFFD), which might lead to unexpected results or misinterpretation of the data.

Recommendation

Ifzkoala : To mitigate this issue, consider using a proper validation mechanism for the input data to ensure it's valid UTF-8.

Client Response

Fixed

UPC-20:min size of coset not accurate in `ProverKey::new`

Category	Severity	Status	Contributor
Logical	**Low**	**Fixed**	**alansh**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/prover/prover_key.rs#L36

```
36:        let coset = D::new((domain.size() + 1) * program_width + 2)
```

Description

alansh : The size of coset is decided by the grand product polynomial of permutation, which is a product of `program_width` degree `N+1` polynomial and degree `N+2` `z` polynomial, minus the degree of vanishing polynomial `N`, the total degree is `program_width*(N+1) + N+2 - N=program_width*(N+1) + 2`, so the number of coefficients should be `program_width*(N+1) + 3`.

Recommendation

alansh :

```
let coset = D::new((domain.size() + 1) * program_width + 2)
```

should be

```
let coset = D::new((domain.size() + 1) * program_width + 3)
```

Client Response

Fixed

UPC-21:Code Style in `sha256.rs` , prover `mod.rs` and `parser.rs`

Category	Severity	Status	Contributor
Code Style	**Informational**	**Mitigated**	**newway55**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/sha256.rs#L15-L112
- code/UniPass-email-circuits/crates/plookup-sha256/src/prover/mod.rs#L34-L47
- code/UniPass-email-circuits/crates/plookup-sha256/src/sha256.rs#L714

```
    "983e5152", "a831c66d", "b00327c8", "bf597fc7", "c6e00bf3", "d5a79147", "06ca6351", "14292967",
    "27b70a85", "2e1b2138", "4d2c6dfc", "53380d13", "650a7354", "766a0abb", "81c2c92e", "92722c85",
    "a2bfe8a1", "a81a664b", "c24b8b70", "c76c51a3", "d192e819", "d6990624", "f40e3585", "106aa070",
    "19a4c116", "1e376c08", "2748774c", "34b0bcb5", "391c0cb3", "4ed8aa4a", "5b9cca4f", "682e6ff3",
    "748f82ee", "78a5636f", "84c87814", "8cc70208", "90befffa", "a4506ceb", "bef9a3f7", "c67178f2",
];

impl Table {
    /// used for sha256. also can be used as range constraint.
    /// reference https://zcash.github.io/halo2/design/gadgets/sha256/table16.html
    pub fn spread_table(bits: usize) -> Self {
        let size = 1 << bits;
        let width = 2;
        let key_width = 1; //key is num

        let mut columns = vec![Vec::with_capacity(size); width];
        let mut key_map = Map::new();
        let mut row = 0;
        for key in 0..size {
            let mut key_spread = 0u64;

            let mut tmp = key;
            for i in 0..bits {
                if tmp == 0 {
                    break;
                }

                let this_bit = tmp % 2;
                tmp >>= 1;

                if this_bit == 1 {
                    key_spread += 1 << (2 * i);
                }
            }

            for (i, v) in vec![F::from(key as u64), F::from(key_spread)]
                .into_iter()
                .enumerate()
            {
                columns[i].push(v);
            }

            key_map.insert(vec![F::from(key as u64)], row);
            row += 1;
        }

        Table {
            id: format!("spread_{}bits", bits),
            index: 0,
            size,
            width,
            key_width,
            columns,
            lookups: Vec::new(),

            key_map,
        }
    }

    /// reference https://zcash.github.io/halo2/design/gadgets/sha256/table16.html
    /// contain another smaller spread table, which 'lookup key' should add an extra '0'.
    pub fn spread_table_2in1(bits: usize, small_bits: usize) -> Self {
        assert!(bits > small_bits);
        let size = 1 << bits;
        let small_size = 1 << small_bits;
        let width = 3;
        let key_width = 2; //key is (num, sign)

        let mut columns = vec![Vec::with_capacity(size); width];
        let mut key_map = Map::new();
        let mut row = 0;
        for key in 0..size {
            let mut key_spread = 0u64;

            let mut tmp = key;
            for i in 0..bits {
                if tmp == 0 {
                    break;
                }

                let this_bit = tmp % 2;
                tmp >>= 1;

                if this_bit == 1 {
                    key_spread += 1 << (2 * i);
                }
            }

            if key < small_size {
                for (i, v) in vec![F::from(key as u64), F::zero(), F::from(key_spread as u64)]
                    .into_iter()
                    .enumerate()
                {
                    columns[i].push(v);
                }
                key_map.insert(vec![F::from(key as u64), F::zero()], row);
            } else {
                for (i, v) in vec![F::from(key as u64), F::one(), F::from(key_spread as u64)]

pub struct Prover, E: PairingEngine> {
    domain_values: Map>, //already padded
    coset_values: Map>,  //coset fft
    polynomials: Map>,

    challenges: Map,
    pub evaluations: Map,
    pub commitments: Map>,

    pub domain: D,
    pub coset: D,
    pub program_width: usize,

    pub composer_config: ComposerConfig,

        assert_eq!(value_u64, 0);
```

Description

newway55 : Using ok_or is more idiomatic Rust and makes the code more readable by clearly communicating the error handling logic.

Consider below POC contract

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_parse_header() {
        let dkim_msg = b"from: <example@example.com>\r\nsubject: Test email\r\n";
        let dkim_header = Header {
            sdid: "example.com".to_string(),
            selector: "selector".to_string(),
        };
        let from_pepper = vec![0u8; 16];
        let from = "example@example.com".to_string();

        let result = parse_header(dkim_msg, &dkim_header, from_pepper, from);

        assert!(result.is_ok());

        let (public_inputs, private_inputs) = result.unwrap();

        assert_eq!(public_inputs.from_pepper, from_pepper);
        assert_eq!(private_inputs.from_left_index, 6);
        assert_eq!(private_inputs.from_right_index, 26);
        assert_eq!(private_inputs.subject_index, 30);
        assert_eq!(private_inputs.subject_right_index, 41);
        assert_eq!(private_inputs.dkim_header_index, 0);
        assert_eq!(private_inputs.selector_index, 31);
        assert_eq!(private_inputs.selector_right_index, 39);
        assert_eq!(private_inputs.sdid_index, 42);
        assert_eq!(private_inputs.sdid_right_index, 52);
    }
}


}
```

newway55 : Declaring public many fields will lead to possible repercussions in any code that uses the `Prover` struct, because this code will be able to access and modify the fields.

- Impact : An attacker who can modify these public values in commitment fields for example can create invalid proofs which will be validated by verifier.

Consider below POC contract

This test checks that the private modifier is ok (implemented correctly) by verifying that it's not possible to access the private fields.

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_private_fields() {
        let prover = Prover::<Fr, EvaluationDomain<Fr>, Bls12>::new(
            EvaluationDomain::<Fr>::new(16).unwrap(),
            EvaluationDomain::<Fr>::new(32).unwrap(),
            ComposerConfig::new(5),
        );

        // check if the fields are private
        // these should fail to compile
        // prover.domain_values; // uncommenting should not compile
        // prover.coset_values; // uncommenting should not compile
        // prover.polynomials; // uncommenting should not compile
        // prover.challenges; // uncommenting should not compile
        // prover.evaluations; // uncommenting should not compile
        // prover.commitments; // uncommenting should not compile
        // prover.domain; // uncommenting should not compile
        // prover.coset; // uncommenting should not compile
        // prover.program_width; // uncommenting should not compile
    }
}
}
```

newway55 : - The assertion assert_eq!(value_u64, 0); is redundant because the bitwise shifts performed earlier ensure that value_u64 is zero beyond the bits that have been extracted.

- The assertion is unnecessary and can be safely removed without affecting the correctness of the code.

Consider below POC contract

```rust
#[cfg(test)]
mod tests {
    use super::*;
    use crate::composer::Composer;

    #[test]
    fn test_new_sha256sigma0form() {
        // Create a new Composer
        let mut composer = Composer::new();

        // Initialize a Sha256sigma0form with a value of 1234567890
        let word = Sha256Word::new_constant(1234567890u64.into(), "test_word");
        let _ = Sha256sigma0form::new(&mut composer, &word).unwrap();

        // Check that the assertion in the Sha256sigma0form constructor does not fail
        assert!(true);
    }
}
```

Recommendation

newway55 : ok_or method instead of unwrap_or_else.

**More concise way of handling the Option type and produces a cleaner code.

The ok_or method returns the Ok value if it exists, and otherwise returns an Err value with the given argument. In this case, we use it to replace the unwrap_or_else method that was previously being used to return an error if the find_subsequence function returns None.

Consider below fix in the `parse_header` function

```
fn parse_header(
    dkim_msg: &[u8],
    dkim_header: &Header,
    from_pepper: Vec<u8>,
    from: String,
) -> ParserResult<(PublicInputs, PrivateInputs)> {
    let from_index = find_subsequence(dkim_msg, b"from:")
        .ok_or(ParserError::HeaderFormatError)?;

    let from_end_index = find_subsequence(&dkim_msg[from_index..], b"\r\n")
        .ok_or(ParserError::HeaderFormatError)? + from_index;

    let from_left_index = find_subsequence(&dkim_msg[from_index..], format!("<{}>", from).as_bytes
())
        .map_or_else(|| find_subsequence(dkim_msg, from.as_bytes()), |index| {
            if index < from_end_index {
                Some(from_index + index + 1)
            } else {
                find_subsequence(dkim_msg, from.as_bytes())
            }
        }).ok_or(ParserError::HeaderFormatError)?;

    let from_right_index = from_left_index + from.len() - 1;

    let subject_index = find_subsequence(dkim_msg, b"subject:")
        .ok_or(ParserError::HeaderFormatError)?;

    let subject_right_index = find_subsequence(&dkim_msg[subject_index..], b"\r\n")
        .ok_or(ParserError::HeaderFormatError)? + subject_index;

    let dkim_header_index = find_subsequence(dkim_msg, b"dkim-signature:")
        .ok_or(ParserError::HeaderFormatError)?;

    let sdid_index = find_subsequence(&dkim_msg[dkim_header_index..], b"d=")
        .map_or(Err(ParserError::HeaderFormatError), |d_index| {
            find_subsequence(&dkim_msg[d_index..], dkim_header.sdid.as_bytes())
                .map(|index| d_index + index)
                .ok_or(ParserError::HeaderFormatError)
        })?;

    let sdid_right_index = sdid_index + dkim_header.sdid.len();

    let selector_index = find_subsequence(&dkim_msg[dkim_header_index..], b"s=")
.map_or(Err(ParserError::HeaderFormatError), |s_index| {
            find_subsequence(&dkim_msg[s_index..], dkim_header.selector.as_bytes())
                .map(|index| s_index + index)
                .ok_or(ParserError::HeaderFormatError)
        })?;

    let selector_right_index = selector_index + dkim_header.selector.len();

    let private_input = PrivateInputs {
        email_header: dkim_msg.to_vec(),
        from_pepper,
        from_index,
        from_left_index,
        from_right_index,
        subject_index,
        subject_right_index,
        dkim_header_index,
        selector_index,
        selector_right_index,
        sdid_index,
        sdid_right_index,
    };

    let public_input: PublicInputs = (&private_input).into();

    Ok((public_input, private_input))
}
```

newway55 : - Add private modifier to the fields. Access is restricted to Prover struct only. Prevents external code from access and modifying.

Consider below fix in the `sample.test()` function

```
pub struct Prover<F: Field, D: Domain<F>, E: PairingEngine> {
    private domain_values: Map<String, Vec<F>>, //already padded
    private coset_values: Map<String, Vec<F>>,  //coset fft
    private polynomials: Map<String, DensePolynomial<F>>,

    private challenges: Map<String, F>,
    private evaluations: Map<String, F>,
    private commitments: Map<String, Commitment<E>>,

    private domain: D,
    private coset: D,
    private program_width: usize,

    pub composer_config: ComposerConfig,
}
```

newway55 : - Remove `assert_eq!(value_u64, 0);`

Consider below fix in the `` function

```
impl Sha256sigma0form {
    /// split the word into a|b|c|d (3,4,11,14)bits
    fn new<F: Field>(cs: &mut Composer<F>, word: &Sha256Word) -> Result<Self, Error> {
        let value = cs.get_assignment(word.var);

        let tmp = value.into_repr();
        let value_u64 = tmp.as_ref();
        let mut value_u64 = value_u64[0].clone();

        let a_value = value_u64 % 8;
        value_u64 >>= 3;
        let b_value = value_u64 % 16;
        value_u64 >>= 4;
        let c_value = value_u64 % 2048;
        value_u64 >>= 11;
        let d_value = value_u64;
        value_u64 >>= 14;

        let a = cs.alloc(F::from(a_value));
        let b = cs.alloc(F::from(b_value));
        let c = cs.alloc(F::from(c_value));
        let d = cs.alloc(F::from(d_value));

   .....
```

Client Response

Mitigated,Just like UPC-1

UPC-22:Gas Optimization in `arithmetic.rs` in `Composer` implementation in `add` function

Category	Severity	Status	Contributor
Gas Optimization	**Informational**	**Declined**	**newway55**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/arithmetic.rs#L8

```
8:     pub fn add(&mut self, var_l: Variable, var_r: Variable) -> Variable {
```

Description

newway55 : Batching can be applied to the add function, which is called many times in the code. Instead of performing each addition one by one, we can batch them together and perform them all at once. This significantly reduces the gas cost because it requires only a single EVM instruction to perform multiple additions, rather than performing each addition separately and incurring gas costs for each operation.

Consider below POC contract

```rust
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_batch_add() {
        let mut composer = Composer::new();

        // Create an array of random numbers
        let nums: Vec<F> = (0..10).map(|_| F::rand()).collect();

        // Allocate variables for the numbers
        let vars: Vec<Variable> = nums.iter().map(|n| composer.alloc(*n)).collect();

        // Compute the sum of the numbers using batch_add
        let sum = composer.batch_add(&vars);

        // Compute the expected sum
        let expected_sum = nums.iter().sum();

        // Assert that the computed sum is equal to the expected sum
        assert_eq!(composer.eval(sum), expected_sum);
    }
}
```

Recommendation

newway55 : - Modify the function to accept a list of (left, right) pairs to add together

Consider below fix in the `` function

```
fn add_batch(&mut self, inputs: &[(Variable, Variable)]) -> Vec<Variable> {
    let results = inputs.iter().map(|_| self.alloc(F::zero())).collect::<Vec<_>>();
    let index = self.insert_gate(
        inputs
            .iter()
            .flat_map(|(left, right, result)| vec![*left, *right, *result])
            .collect(),
    );
    self.selectors
        .get_mut("q_add")
        .unwrap()
        .iter_mut()
        .skip(index)
        .take(inputs.len())
        .for_each(|x| *x = F::one());
    results
}
```

Client Response

Declined,I can't understand the purpose of this suggestion, and the circuit implementation has little to do with Gas consumption.

UPC-23:Inconsistent Hash Function Used

Category	Severity	Status	Contributor
Logical	Informational	Declined	Ifzkoala

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/transcript.rs#L4

```
4:use sha3::{Digest, Keccak256};
```

Description

Ifzkoala : You are using Keccak256 from the sha3 crate to get challenge, which is a widely used cryptographic hash function. However, if you are working with pairing-based cryptography, consider using a hash function that is more commonly used in this domain, such as Blake2s or Poseidon. Also for Plonk you're using Sha256 which is inconsistent.

Recommendation

Ifzkoala : Consider using consistent hash function or more pairing-friendly hash functions. Using Sha3 may not lead very serious issues but using consistent hash function will improve the code to be more efficient.

Client Response

Declined,We didn't implement Keccak256 in the circuit, just used it as CRH to generate random challenges, because it is cheap to use in Solidity.

UPC-24:Missing deserialization result validation

Category	Severity	Status	Contributor
Logical	Informational	Acknowledged	BradMoonUESTC

Code Reference

- code/UniPass-email-circuits/crates/prover/src/parameters.rs#L49-L55
- code/UniPass-email-circuits/crates/prover/src/parameters.rs#L79-L87
- code/UniPass-email-circuits/crates/prover/src/parameters.rs#L100-L107

```
49:pub fn load_params(p: &str) -> Result, SerializationError> {
50:    let pckey_file = File::open(p)?;
51:    let pckey_file = io::BufReader::new(pckey_file);
52:    let pckey = PCKey::deserialize_unchecked(pckey_file)?;
53:
54:    return Ok(pckey);
55:}

79:pub fn load_prover_key>(
80:    p: &str,
81:) -> Result, SerializationError> {
82:    let pk_file = File::open(p)?;
83:    let pk_file = io::BufReader::new(pk_file);
84:    let pk = ProverKey::deserialize_unchecked(pk_file)?;
85:
86:    return Ok(pk);
87:}

100:pub fn load_verifier_comms(
101:    p: &str,
102:) -> Result>, SerializationError> {
103:    let vcomms_file = File::open(p)?;
104:    let vcomms_file = io::BufReader::new(vcomms_file);
105:    let vcomms = Vec::deserialize_unchecked(vcomms_file)?;
106:    return Ok(vcomms);
107:}
```

Description

BradMoonUESTC : In the load_params, load_prover_key, and load_verifier_comms functions, the deserialization results are not validated. It's important to verify the deserialized data to ensure its integrity and validity.

Recommendation

BradMoonUESTC : Add validation checks for the deserialization results in the respective functions. For example, in the load_verifier_comms function, you can add the following check to ensure the vcomms vector is not empty:

```
if vcomms.is_empty() {
    return Err(SerializationError::InvalidData("The verifier_comms vector is empty."));
}
```

Client Response

Acknowledged,This check is located in the process of importing CRS parameters in snarkJs. This process is only operated once when the circuit deployment is started, so the impact is very small.

UPC-25:No error handling for empty email_header_bytes and email_addr_pepper_bytes

Category	Severity	Status	Contributor
Privilege Related	**Informational**	**Mitigated**	**BradMoonUESTC**

Code Reference

- code/UniPass-email-circuits/crates/prover/src/circuit/circuit_1024.rs#L27

```
27:        let email_addr_bytes = private_inputs.email_header
```

Description

BradMoonUESTC : In the current implementation, there is no error handling for empty email_header_bytes. This may lead to unexpected behaviors when processing email headers.

```
let email_header_bytes = &email_header[..];
let email_addr_pepper_bytes = &email_addr_pepper[..];
```

Recommendation

BradMoonUESTC : Add error handling for empty email_header_bytes and return an appropriate error message.

```
if email_header_bytes.is_empty() {
    return Err("Empty email_header_bytes");
}
```

Client Response

Mitigated,When processing emails into email_headers, checks are made to avoid empty email_headers.

UPC-26:Padding bytes may introduce unnecessary computations

Category	Severity	Status	Contributor
Privilege Related	**Informational**	**Declined**	**BradMoonUESTC**

Code Reference

- code/UniPass-email-circuits/crates/prover/src/circuit/circuit_1024.rs#L64-L66

```
64:             if i == private_inputs.dkim_header_index - 2 {
65:                 break;
66:             }
```

Description

BradMoonUESTC : The padding bytes (email_header_bytes_padding, email_addr_pepper_bytes_padding, email_header_pub_match_padding) may introduce unnecessary computations in the circuit, increasing the complexity.

```
let email_header_bytes_padding = vec![0u8; 32 - email_header_bytes.len()];
let email_addr_pepper_bytes_padding = vec![0u8; 32 - email_addr_pepper_bytes.len()];
let email_header_pub_match_padding = vec![0u8; 32 - email_header_pub_match.len()];
```

Recommendation

BradMoonUESTC : Consider optimizing the padding byte handling to reduce unnecessary computations within the circuit.

```
let email_header_bytes_padding = vec![0u8; PADDING_LENGTH - email_header_bytes.len()];
let email_addr_pepper_bytes_padding = vec![0u8; PADDING_LENGTH - email_addr_pepper_bytes.len()];
let email_header_pub_match_padding = vec![0u8; PADDING_LENGTH - email_header_pub_match.len()];
```

Client Response

Declined,The padding format needs to be consistent with the Sha256 hash and cannot be changed to other modes.

UPC-27:Publicly Exposed Constant

Category	Severity	Status	Contributor
Privilege Related	Informational	Mitigated	Ifzkoala

Code Reference

- code/UniPass-email-circuits/crates/prover/src/circuit/base64.rs#L9-L10

```
9:pub const BASE64URL_ENCODE_CHARS: &[u8; 64] =
10:    b"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_";
```

Description

Ifzkoala : The constant BASE64URL_ENCODE_CHARS is public, which could potentially allow unauthorized modification.

Recommendation

Ifzkoala : Consider making it private or using a public getter function if needed.

Client Response

Mitigated,const variables are immutable.

UPC-28:Unnecessary cloning of the chunk_messages

Category	Severity	Status	Contributor
Language Specific	**Informational**	**Fixed**	**Ifzkoala**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/sha256.rs#L329

```
329:    Words.append(&mut chunk_messages.clone().to_vec());
```

Description

Ifzkoala : The line `Words.append(&mut chunk_messages.clone().to_vec())`; can be changed to `Words.extend_from_slice(chunk_messages)`; to avoid unnecessary cloning and make the code more efficient.

Recommendation

Ifzkoala : Change `Words.append(&mut chunk_messages.clone().to_vec())` to `Words.extend_from_slice(chunk_messages)`

Client Response

Fixed

UPC-29:Unnecessary mut in the get_value_by_key function

Category	Severity	Status	Contributor
Code Style	**Informational**	**Acknowledged**	**Ifzkoala**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/lookup.rs#L27

```
27:    fn get_value_by_key(&mut self, key: &[F]) -> Result, Error> {
```

Description

Ifzkoala : The `&mut self` parameter in the `get_value_by_key` function may not be necessary, as it doesn't appear to modify the struct in any way except for appending to `self.lookups`. If the purpose of appending to `self.lookups` is to record that a lookup occurred, this could be done in a more explicit way, for example, by using a separate method or passing a mutable reference to a container for recording lookups as an argument.

Recommendation

Ifzkoala : Using a separate method or passing a mutable reference to a container for recording lookups as an argument.

Client Response

Acknowledged

UPC-30:Unused code

Category	Severity	Status	Contributor
Code Style	Informational	Fixed	alansh

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/mod.rs#L52-L61
- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/mod.rs#L154
- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/mod.rs#L198-L201

```
52:    pub fn new(initsize: usize) -> Self {
53:        let mut disjoint = vec![0usize; initsize];
54:        let rank = vec![1usize; initsize];
55:
56:        for i in 0..disjoint.len() {
57:            disjoint[i] = i;
58:        }
59:
60:        SimpleUnionFind { disjoint, rank }
61:    }

154:        let _ = cs.alloc(F::zero());

198:    /// any value should be OK
199:    pub fn any() -> Variable {
200:        Variable(0)
201:    }
```

Description

alansh : The variable "any" is never used, and it's assigned a fixed value 0 instead of a more random value.
alansh : `SimpleUnionFind::new` is not used and should be removed.

Recommendation

alansh : Remove unused variable, adjust `null()` correspondingly.
alansh : Remove `SimpleUnionFind::new`

Client Response

Fixed

UPC-31:Using `Cargo Clippy` Specification Code

Category	Severity	Status	Contributor
Code Style	**Informational**	**Fixed**	**Hellobloc**

Code Reference

- code/UniPass-email-circuits/src/main.rs#L1

```
1:use plonk::ark_serialize::SerializationError;
```

Description

Hellobloc : There is a large amount of code in the current code base that can be further standardized, and these specifications can help reduce useless operations and increase code readability.

Specifically, we can use the static analysis of `cargo clippy` to get the following code specification suggestions.

- unneeded `return` statement
- if-then-else expression returns a bool literal
- unnecessary `>= y + 1` or `x - 1 >=`
- needlessly taken reference of left operand
- question mark operator is useless here
- operator precedence can trip the unwary
- redundant field names in struct initialization
- the loop variable `i` `col` `j` is used to index `wires`
- length comparison to zero
- writing `&Vec` instead of `&[_]` involves a new object where a slice will do
- this expression creates a reference which is immediately dereferenced by the compiler
- returning the result of a `let` binding from a block
- used a field initializer for a tuple struct
- manual implementation of an assign operation
- this lifetime isn't used in the impl
- redundant clone
- casting integer literal to `u64` is unnecessary
- using `clone` on the type which implements the `Copy` trait

Given the overwhelming amount of content in the above recommendations but the specific information is all available at cargo clippy.

Therefore, in order to improve the readability of the audit report, we will not expand on the details here.

Recommendation

Hellobloc : We recommend adding Clippy's lint check to Github Action and adopting the reasonable suggestions therein, replacing code that is unnecessary or makes it difficult to read.

Client Response

Fixed

UPC-32: `PCKey` has both `pub max_degree` and `pub fn max_degree`

Category	Severity	Status	Contributor
Code Style	**Informational**	**Fixed**	**alansh**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/kzg10.rs#L23
- code/UniPass-email-circuits/crates/plookup-sha256/src/kzg10.rs#L217-L219

```
23:    pub max_degree: usize,

217:    pub fn max_degree(&self) -> usize {
218:        self.max_degree
219:    }
```

Description

alansh : Just keeping one of them is enough.

Recommendation

alansh : Remove one of them

Client Response

Fixed

UPC-33: `blind_and_coset_fft` should ensure the highest blinding coefficient is not zero

Category	Severity	Status	Contributor
Logical	**Informational**	**Fixed**	**alansh**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/utils.rs#L222

```
222:        blind_coeffs.push(F::rand(rng));
```

Description

alansh :

```
    for _ in 0..open_num {
        blind_coeffs.push(F::rand(rng));
    }
```

To ensure perfect hiding the highest blinding coefficient should not be zero, and since here `rng` is purely local, it's cheap to throw zero and generate another random `F`.

Recommendation

alansh : Ensure the last element of `blind_coeffs` to be non-zero.

Client Response

Fixed

UPC-34: `epicycles` / `wires` could be updated in place in `Composer::finalize`

Category	Severity	Status	Contributor
Gas Optimization	Informational	Fixed	alansh

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/mod.rs#L322-L327
- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/mod.rs#L337-L348
- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/mod.rs#L350-L360

```
        for epicycle in self.epicycles.iter_mut() {
            *epicycle = cfg_iter_mut!(epicycle)
                //Move all wires down 'input size' lines
                .map(|w| Wire::new(w.col, w.row + input_size))
                .collect()
        }

        let mut wires = Map::new();
        //put PI at the front of w_0, q0 must be 1 and other 'q' must be 0.
        for (label, wire) in self.wires.iter_mut() {
            let mut vars = if label == "w_0" {
                self.public_input.clone()
            } else {
                vec![Self::null(); input_size]
            };
            vars.append(wire);
            wires.insert(label.to_string(), vars);
        }
        self.wires = wires;

        let mut selectors = Map::new();
        for (label, selector) in self.selectors.iter_mut() {
            let mut values = if label == "q_0" {
                vec![F::one(); input_size]
            } else {
                vec![F::zero(); input_size]
            };
            values.append(selector);
            selectors.insert(label.to_string(), values);
        }
        self.selectors = selectors;
```

Description

alansh :

```
        for epicycle in self.epicycles.iter_mut() {
            *epicycle = cfg_iter_mut!(epicycle)
                //Move all wires down 'input size' lines
                .map(|w| Wire::new(w.col, w.row + input_size))
                .collect()
        }
```

The above code doesn't make use of `iter_mut`. Similar for `wires`/`selectors`

Recommendation

alansh : Update `epicycle`/`wires`/`selectors` in place instead of returning new ones.

Client Response

Fixed

UPC-35: `if` and `while` are using the same condition in `Prover::new`

Category	Severity	Status	Contributor
Code Style	Informational	Fixed	alansh

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/prover/mod.rs#L485-L487

```
485:        if t_chunks.len() > self.program_width {
486:            //put the extra in the last
487:            while t_chunks.len() > self.program_width {
```

Description

alansh :

```
        // high probability because of blind
        if t_chunks.len() > self.program_width {
            //put the extra in the last
            while t_chunks.len() > self.program_width {
              .....
            }
        }
```

can be shortened as

```
        // high probability because of blind
        //put the extra in the last
        while t_chunks.len() > self.program_width {
          .....
        }
```

Recommendation

alansh : Don't repeat yourself

Client Response

Fixed

UPC-36:comment wrong for coset_generator

Category	Severity	Status	Contributor
Logical	Informational	Fixed	alansh

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/utils.rs#L110

```
110:/// saying that $Hi \cup Hj \neq \emptyset$.
```

Description

alansh : `saying that $Hi \cup Hj \neq \emptyset$` should be `saying that $Hi \cup Hj \eq \emptyset$`

Recommendation

alansh : `\neq` => `\eq`

Client Response

Fixed

UPC-37:coset for `sigma_4` is not removed in `PermutationWidget::compute_quotient_contribution`

Category	Severity	Status	Contributor
Logical	**Informational**	**Fixed**	**alansh**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/prover/widget/permutation.rs#L153

```
153:        prover.remove_coset_values("sigma_3");
```

Description

alansh : Should add:

```
prover.remove_coset_values("sigma_4");
```

Recommendation

alansh :

```
prover.remove_coset_values("sigma_4");
```

Client Response

Fixed

UPC-38:dangling branch in padding_bytes

Category	Severity	Status	Contributor
Logical	Informational	Fixed	alansh

Code Reference

- code/UniPass-email-circuits/crates/prover/src/utils.rs#L33-L39

```
33:    let padding_count = if input_remainder < 448 {
34:        (448 - input_remainder) / 8
35:    } else if input_remainder >= 448 {
36:        (448 + 512 - input_remainder) / 8
37:    } else {
38:        64
39:    };
```

Description

alansh : The `else` branch will never be possible:

```
    let padding_count = if input_remainder < 448 {
        (448 - input_remainder) / 8
    } else if input_remainder >= 448 {
        (448 + 512 - input_remainder) / 8
    } else {
        64
    };
```

Recommendation

alansh : remove the `else` branch.

Client Response

Fixed

UPC-39:duplicate code in Table::spread_table and Table::spread_table_2in1

Category	Severity	Status	Contributor
Code Style	Informational	Fixed	alansh

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/sha256.rs#L34-L48
- code/UniPass-email-circuits/crates/plookup-sha256/src/sha256.rs#L87-L101

```
            let mut key_spread = 0u64;

            let mut tmp = key;
            for i in 0..bits {
                if tmp == 0 {
                    break;
                }

                let this_bit = tmp % 2;
                tmp >>= 1;

                if this_bit == 1 {
                    key_spread += 1 << (2 * i);
                }
            }

            let mut key_spread = 0u64;

            let mut tmp = key;
            for i in 0..bits {
                if tmp == 0 {
                    break;
                }

                let this_bit = tmp % 2;
                tmp >>= 1;

                if this_bit == 1 {
                    key_spread += 1 << (2 * i);
                }
            }
```

Description

alansh : The code below appears both in `Table::spread_table` and `Table::spread_table_2in1`:

```
            let mut key_spread = 0u64;

            let mut tmp = key;
            for i in 0..bits {
                if tmp == 0 {
                    break;
                }

                let this_bit = tmp % 2;
                tmp >>= 1;

                if this_bit == 1 {
                    key_spread += 1 << (2 * i);
                }
            }
```

Recommendation

alansh : Put the code into a `func key_to_spread` for better reuse.

Client Response

Fixed

UPC-40:loop code can be more breviate in `Prover::new`

Category	Severity	Status	Contributor
Code Style	**Informational**	**Fixed**	**alansh**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/prover/mod.rs#L562-L569

```
562:        let mut tmp = zeta_n;
563:        let mut t_LC_terms = vec![(F::one(), format!("t_{}", 0))];
564:        for i in 1..self.program_width {
565:            let str = format!("t_{}", i);
566:            t_LC_terms.push((tmp, str));
567:
568:            tmp *= zeta_n;
569:        }
```

Description

alansh :

```
        let mut tmp = zeta_n;
        let mut t_LC_terms = vec![(F::one(), format!("t_{}", 0))];
        for i in 1..self.program_width {
            let str = format!("t_{}", i);
            t_LC_terms.push((tmp, str));

            tmp *= zeta_n;
        }
```

can be more breviate:

```
        let mut tmp = F::one();
        let mut t_LC_terms = Vec::with_capacity(self.program_width);
        for i in 0..self.program_width {
            let str = format!("t_{}", i);
            t_LC_terms.push((tmp, str));

            tmp *= zeta_n;
        }
```

Recommendation

alansh : Don't repeat yourself.

Client Response

Fixed

UPC-41:missing assert! in test_setup function

Category	Severity	Status	Contributor
Logical	Informational	Fixed	alansh

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/kzg10.rs#L394

```
394:        pckey.check();
```

Description

alansh : The check result is not asserted:

```
    fn test_setup() -> Result<(), Error> {
        // let rng = &mut thread_rng();
        let rng = &mut test_rng();
        let pckey = PCKey::<ark_bn254::Bn254>::setup(16, rng);

        pckey.check();

        Ok(())
    }
```

Recommendation

alansh :

```
    fn test_setup() -> Result<(), Error> {
        // let rng = &mut thread_rng();
        let rng = &mut test_rng();
        let pckey = PCKey::<ark_bn254::Bn254>::setup(16, rng);

        assert!(pckey.check());

        Ok(())
    }
```

Client Response

Fixed

UPC-42:performance issue in `blind_t`

Category	Severity	Status	Contributor
Gas Optimization	Informational	Fixed	alansh

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/utils.rs#L251-L252
- code/UniPass-email-circuits/crates/plookup-sha256/src/utils.rs#L256-L257

```
251:    let mut coeff0 = tmp_coeff.clone();
252:    coeff0.push(blind_rand[0]);

256:        let mut coeff = tmp_coeff.clone();
257:        coeff.push(blind_rand[i]);
```

Description

alansh :

```
let tmp_coeff = vec![F::zero(); domain.size()];

    let mut coeff0 = tmp_coeff.clone();
    coeff0.push(blind_rand[0]);
```

The above code first allocates a `vec` of size `domain.size()` then push 1 element into it.This will cause realloc.

Recommendation

alansh : Consider below fix

```
let mut coeff0 = vec![F::zero(); domain.size()+1];
coeff0[domain.size()] = blind_rand[0];
```

Similar for `coeff`.

Client Response

Fixed

UPC-43:q_arith selector is not actually used in Composer::fully_costomizable_poly_gates

Category	Severity	Status	Contributor
Code Style	**Informational**	**Fixed**	**alansh**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/arithmetic.rs#L99-L106

```
99:        q_arith: Vec,
100:        q_m: Vec,
101:        q_c: Vec,
102:        q0next: Vec,
103:    ) {
104:        assert!(!self.is_finalized);
105:        let n = multiple_wires.len();
106:        assert_eq!(n, q_arith.len());
```

Description

alansh : There's no `q_arith` selector in the circuit, so here `q_arith` is confusing. Its content is not actually used, only length check.

Recommendation

alansh : If `q_arith` selector is needed, actually implement it. Otherwise remove it. Or if only length is important, pass `q_arith_length` instead.

Client Response

Fixed

UPC-44:typo in Composer::fully_costomizable_poly_gates

Category	Severity	Status	Contributor
Code Style	**Informational**	**Fixed**	**alansh**

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/composer/arithmetic.rs#L96

```
96:     pub fn fully_costomizable_poly_gates(
```

Description

alansh : Should be `Composer::fully_customizable_poly_gates`

Recommendation

alansh : Replace `fully_costomizable_poly_gates` with `fully_customizable_poly_gates`.

Client Response

Fixed

UPC-45:using both ark_std and std

Category	Severity	Status	Contributor
Code Style	Informational	Fixed	alansh

Code Reference

- code/UniPass-email-circuits/crates/plookup-sha256/src/prover/widget/mod.rs#L3
- code/UniPass-email-circuits/crates/plookup-sha256/src/kzg10.rs#L13

```
3:use rand_core::RngCore;

13:use std::ops::Div;
```

Description

alansh : `ark-std` is a library that serves as a compatibility layer for `no_std` use cases, so should not use `std` directly, otherwise the whole point of `ark-std` is lost.

Recommendation

alansh : Use `ark-std` consistently, search `std::` to see all the locations.
Similarly the project is using both `rand_core::RngCore` and `ark_std::rand::RngCore`, should use one consistently.

Client Response

Fixed

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3's prior written consent in each instance.

This report is not an "endorsement" or "disapproval" of any particular project or team. This report is not an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3's position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.