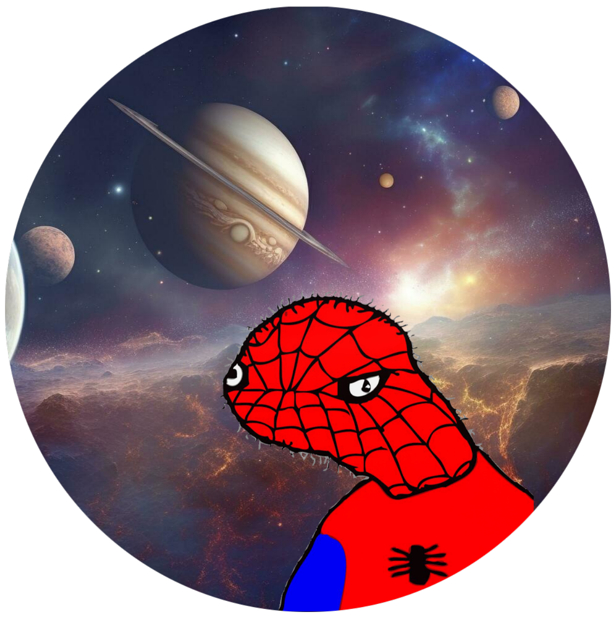

Privacy Cash - Privacy Cash Audit Report

Version 1.5

Zigtur

October 9, 2025

Privacy Cash - Privacy Cash - Audit Report

Zigtur

October 09, 2025

Prepared by: Zigtur

Table of Contents

Introduction

Disclaimer

A smart contract security review cannot guarantee the complete absence of vulnerabilities. This effort, bound by time, resources, and expertise, aims to identify as many security issues as possible. However, there is no assurance of 100% security post-review, nor is there a guarantee that the review will uncover all potential problems in the smart contracts. It is highly recommended to conduct subsequent security reviews, implement bug bounty programs, and perform on-chain monitoring.

About Zigtur

Zigtur is an independent blockchain security researcher dedicated to enhancing the security of the blockchain ecosystem. With a history of identifying numerous security vulnerabilities across various protocols in public audit contests and private audits, **Zigtur** strives to contribute to the safety and reliability of blockchain projects through meticulous security research and reviews. Explore previous work here or reach out on X @zigtur.

About Privacy Cash

PrivacyCash is a decentralized protocol for secure and anonymous transactions built on Solana.

Using zero-knowledge proofs, it lets users deposit assets and withdraw them to different addresses, breaking the link between sender and receiver. By combining advanced cryptography with decentralized infrastructure, PrivacyCash restores financial privacy and freedom on public blockchains.

Security Assessment Summary

Review commit hash - 549686066e81c5434182f9f85b9296bb636b07e9

Fixes review commit hash - None

Additional review commit hash - 2ac0e8942e4e806dc6d2188991576bc05837ef8f

Deployment chains

- Solana

Scope

The following code is in scope of the review:

- anchor/programs/*
- circuits/*

Risk Classification

	Impact: High	**Impact:** Medium	**Impact:** Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Issues

INFO-01 - `is_known_root` loops through zero roots after launch

Scope:

- merkle_tree.rs#L79-L95

Description

The `is_known_root` function loops through the entire root history array to find matching roots. After the protocol launches, most of the root history will initially contain zero values, causing unnecessary iterations through empty slots.

```
pub fn is_known_root(tree_account: &MerkleTreeAccount, root: [u8; 32]) -> bool {
    if root == [0u8; 32] {
        return false;
    }

    let root_history_size = tree_account.root_history_size as usize; // 100
    let current_root_index = tree_account.root_index as usize;
    let mut i = current_root_index;

    loop {
        if root == tree_account.root_history[i] {
            return true;
        }
        // ... continues looping through all 100 slots
    }
}
```

This results in inefficient root lookups, especially in the early stages when only a few roots have been stored.

Recommendation

The `is_known_root` function could stop when it finds an empty merkle root.

Note: The protocol is already live, so this issue can remain unfixed.

Privacy Cash

Acknowledged.

Zigtur

Acknowledged.

INFO-02 - `size_of` is not appropriate for account size

Scope:

- lib.rs#L324
- lib.rs#L337
- lib.rs#L358
- lib.rs#L367
- lib.rs#L406
- lib.rs#L415
- lib.rs#L424

Description

The various accounts used in the codebase are initialized with a space defined through a call to `std::mem::size_of()`.

However, this function indicates the memory layout size which may differ than the Anchor serialized layout. This may lead to defining accounts with an inaccurate space.

Recommendation

Anchor documentation about space indicates that the `InitSpace` macro can be derived to determine the correct space. Then, the `INIT_SPACE` constant can be used.

Note: this macro does not apply to `zero_copy` accounts.

Privacy Cash

Acknowledged.

Zigtur

Acknowledged.

INFO-03 - Storing data in `tree_token_account` is useless

Scope:

- lib.rs#L465-L468

Description

The `TreeTokenAccount` struct only stores an authority pubkey and bump value, but this data is never accessed or used by the program logic.

```
#[account]
pub struct TreeTokenAccount {
    pub authority: Pubkey,
    pub bump: u8,
}
```

The account is created during initialization but serves no functional purpose other than existing as a PDA. The stored authority and bump values are never read or validated in subsequent operations.

Recommendation

Consider removing the `TreeTokenAccount` structure, and use the `tree_token_account` as a simple system account.

Privacy Cash

Acknowledged.

Zigtur

Acknowledged.

INFO-04 - Authority permission is not transferable

Scope:

- lib.rs#L472
- lib.rs#L496

Description

The `GlobalConfig` and `MerkleTreeAccount` structures store an authority field, but there are no functions implemented to transfer or update the authority. This creates a permanent centralization point where the original authority cannot delegate or transfer control.

```rust
#[account]
pub struct GlobalConfig {
    pub authority: Pubkey,  // Cannot be changed
    // ... other fields
}

#[account(zero_copy)]
pub struct MerkleTreeAccount {
    pub authority: Pubkey,  // Cannot be changed
    // ... other fields
}
```

If the authority private key is compromised or lost, there is no mechanism to recover or transfer control to a new authority.

Recommendation

Implement authority transfer functions for both structures:

```rust
pub fn transfer_global_authority(
    ctx: Context<TransferGlobalAuthority>,
    new_authority: Pubkey,
) -> Result<()> {
    let global_config = &mut ctx.accounts.global_config;
    require!(
        ctx.accounts.current_authority.key() == global_config.authority,
        ErrorCode::UnauthorizedAccess
    );
    global_config.authority = new_authority;
    Ok(())
}
```

This would enable proper governance and reduce single points of failure.

Privacy Cash

Acknowledged.

Zigtur

Acknowledged.

INFO-05 - `i64::MIN` check can be removed

Scope:

- utils.rs#L93-L96

Description

The `check_public_amount` function includes an explicit check for `i64::MIN` which is redundant given the existing arithmetic operations:

```rust
pub fn check_public_amount(ext_amount: i64, fee: u64, public_amount_bytes: [u8;
↪  32]) -> bool {
    if ext_amount == i64::MIN {
        msg!("can't use i64::MIN as ext_amount");
        return false;
    }
    // ... later in the code
    let ext_amount_fr = if ext_amount >= 0 {
        Fr::from(ext_amount as u64)
    } else {
        Fr::from((-ext_amount) as u64).neg()  // This would panic on i64::MIN
        ↪  anyway
    };
}
```

The explicit `i64::MIN` check is unnecessary because the negation operation `(-ext_amount)` would panic on `i64::MIN` due to two's complement arithmetic overflow, providing the same protection.

Recommendation

The check can be safely removed since the negation operation already provides overflow protection:

```rust
pub fn check_public_amount(ext_amount: i64, fee: u64, public_amount_bytes: [u8;
↪  32]) -> bool {
    // Remove this check - the negation below will handle it
    // if ext_amount == i64::MIN {
    //     msg!("can't use i64::MIN as ext_amount");
    //     return false;
    // }

    let ext_amount_fr = if ext_amount >= 0 {
        Fr::from(ext_amount as u64)
    } else {
        Fr::from((-ext_amount) as u64).neg()  // Will panic on i64::MIN naturally
    };
}
```

Alternatively, use `checked_neg()` for more explicit error handling.

Privacy Cash

Acknowledged.

Zigtur

Acknowledged.

INFO-06 - `g1_point.neg()` is executed twice

Scope:

- utils.rs#L235-L249

Description

In the `verify_proof` function, `g1_point.neg()` is called twice unnecessarily, which is inefficient:

```
let mut proof_a_neg = [0u8; 65];
if g1_point
    .neg()  // First call
    .x
    .serialize_with_mode(&mut proof_a_neg[..32], Compress::No)
    .is_err() {
    return false;
}
if g1_point
    .neg()  // Second call - redundant computation
    .y
    .serialize_with_mode(&mut proof_a_neg[32..], Compress::No)
    .is_err() {
    return false;
}
```

Each call to `.neg()` performs the elliptic curve point negation operation, so calling it twice is computationally wasteful.

Recommendation

`g1_point.neg()` can be executed only once.

```
diff --git a/anchor/programs/zkcash/src/utils.rs
↪  b/anchor/programs/zkcash/src/utils.rs
index 36f8250..815c6e8 100644
--- a/anchor/programs/zkcash/src/utils.rs
+++ b/anchor/programs/zkcash/src/utils.rs
@@ -233,15 +233,14 @@ pub fn verify_proof(proof: Proof, verifying_key:
↪  Groth16Verifyingkey) -> bool {
     };

     let mut proof_a_neg = [0u8; 65];
-    if g1_point
-        .neg()
+    let g1_point_neg = g1_point.neg();
+    if g1_point_neg
         .x
```

```diff
            .serialize_with_mode(&mut proof_a_neg[..32], Compress::No)
            .is_err() {
            return false;
        }
-       if g1_point
-           .neg()
+       if g1_point_neg
            .y
            .serialize_with_mode(&mut proof_a_neg[32..], Compress::No)
            .is_err() {
```

Privacy Cash

Acknowledged.

Zigtur

Acknowledged.

INFO-07 - Fee payment can be bypassed

Scope:

- lib.rs#L390-L392

Description

The Solana program enforces fees to be paid. However, the fee recipient can be any address.

This allows a user to pay fees to himself.

Recommendation

Consider ensuring that fees are paid to Privacy Cash team.

Privacy Cash

Acknowledged. We intentionally made this choice, so the protocol can one day be fully decentralized.

Zigtur

Acknowledged.

Appendix

Groth16 specific risk assessment

Privacy Cash relies on Groth16 zero-knowledge proofs to ensure transaction privacy and validity. While Groth16 provides excellent proof size and verification efficiency, it introduces specific risks related to the trusted setup ceremony that are critical to understand.

Trusted Setup Requirements

Groth16 is a **non-universal** zero-knowledge proof system, meaning it requires a circuit-specific trusted setup ceremony. This ceremony generates two sets of parameters:

- **Proving Key (PK)**: Used to generate proofs
- **Verifying Key (VK)**: Used to verify proofs on-chain

The security of the entire Privacy Cash protocol depends on the integrity of this trusted setup.

Critical Security Assumptions

1. Toxic Waste Destruction

During the trusted setup, intermediate values called "toxic waste" are generated. If an entity is able to rebuild **all intermediate values**, they can:

- Forge valid proofs for invalid transactions
- Create fake deposits without actual token transfers
- Generate unlimited tokens from nothing
- Completely break the protocol's integrity

When at least one of the trusted setup participants properly deletes his intermediate value, the trusted setup can be considered secure.

2. Multi-Party Computation (MPC) Requirements

The trusted setup must be conducted as a multi-party ceremony where:

- Multiple independent participants contribute randomness
- **Only ONE participant needs to be honest** for security
- All participants must provably destroy their toxic waste
- The ceremony must be publicly verifiable

Consequences of Compromised Setup

If the trusted setup is compromised, an attacker with access to toxic waste could:

Immediate Threats:

- Mint unlimited tokens by creating fake deposit proofs
- Steal all deposited funds by forging withdrawal proofs
- Break transaction unlinkability by correlating inputs/outputs
- Drain the protocol’s token pools without detection

Long-term Impact:

- Complete loss of user funds
- Irreversible protocol compromise
- No way to detect malicious proofs until damage is done
- Requirement for complete redeployment with new trusted setup

Risk Mitigation Strategies

1. Ceremony Participation

- Maximize the number of independent participants
- Include participants from different jurisdictions and organizations
- Require public commitments from participants
- Document the entire ceremony process

2. Transparency Measures

- Publish all ceremony artifacts and transcripts
- Enable independent verification of the setup
- Provide cryptographic proofs of proper execution
- Maintain detailed audit trails

3. Ongoing Vigilance

- Monitor for unusual transaction patterns
- Implement circuit-level constraints and checks
- Consider migration paths to universal proof systems
- Plan for potential trusted setup renewal

Limitations of This Review

Important Note: This security review **did not assess the trusted setup ceremony** conducted for Privacy Cash. The review focused on:

- Circuit logic and constraints
- Smart contract implementation

Recommended Additional Assessments:

1. **Trusted Setup Audit**: Independent verification of ceremony execution
2. **Ceremony Participant Verification**: Validation of participant independence
3. **Toxic Waste Destruction Proof**: Cryptographic evidence of proper cleanup
4. **Parameter Verification**: Mathematical verification of generated parameters