



Solana Program Library ZK-Token Security Assessment

Solana Foundation
Version 1.2 – April 4, 2023

©2023 – NCC Group

Prepared by NCC Group Security Services, Inc. for Solana Foundation. Portions of this document and the templates used in its production are the property of NCC Group and cannot be copied (in full or in part) without NCC Group's permission.

While precautions have been taken in the preparation of this document, NCC Group the publisher, and the author(s) assume no responsibility for errors, omissions, or for damages resulting from the use of the information contained herein. Use of NCC Group's services does not guarantee the security of a system, or that computer intrusions will not occur.

Prepared By
Gérald Doussot
Thomas Pornin

Prepared For
Sam Kim

1 Executive Summary

Synopsis

In August 2022, Solana Foundation engaged NCC Group to conduct a security assessment of the ZK-Token SDK, a collection of open-source functions and types that implement the core cryptographic functionalities of the Solana Program Library (SPL) Confidential Token extension. These functionalities are homomorphic encryption and associated proofs used to demonstrate the consistency of elementary instructions that move tokens between accounts while keeping the involved amounts in an encrypted format that ensures that only the sender and recipient may learn any information about these amounts.

Scope

NCC Group's evaluation was over the contents of the [zk-token-sdk directory](#) of the main Solana repository on GitHub. The most recent commit at the time of the start of the engagement was used (commit 4e43aa6c18e6bb4d98559f80eb004de18bc6b418, dated 5 August 2022). The SDK uses the [curve25519-dalek library](#) for operations over the Ristretto255 group and its associated scalar field; that library was assumed to be correct and to properly implement the documented operations.

The evaluation target was verifying correctness of the implementation with regard to the specifications in [the protocol papers](#) (part1.pdf and part2.pdf files), in particular proper usage of the curve25519-dalek library. Timing-based side-channel attacks on private information (private keys, transaction amounts) were in scope.

Limitations

NCC Group's evaluation covered the functionality implemented by the ZK-Token SDK. The SDK implements zero-knowledge proofs of internal consistency of instructions. For example, the SDK will verify that any transfer instruction is for an amount that does not exceed the current balance of the source account. It is up to the private key owners to ensure that they generate proofs only for legitimate transactions, and that the smart contracts they invoke properly realize all the operations for which the account owners authorize the transfers.

The confidentiality model used by the Confidential Token extension has some inherent limitations, in that it can keep amounts secret for transfers, but a CloseAccount instruction necessarily reveals that the target account has an empty balance; similarly, a Withdraw operation always uses a non-secret amount and thereby reveals that the account balance was at least that high before the operation.

Key Findings and Strategic Recommendations

The assessment uncovered a high-severity issue through which invalid transaction data may trigger a panic and crash the instruction processor by making it attempt to read out-of-bounds data; see [finding "Parsing of SPL ZK-Token Protocol and Cryptographic Key Data May Crash System"](#).

Some operations on secret keys and amounts were found to potentially leak information through timing-based side-channels. Most can be fixed easily with negligible runtime overhead. Leaks related to the use of JSON are harder to mitigate unless JSON is abandoned as a storage format for private elements; this may require architectural changes in applications that use the ZK-Token library. No inexpensive solution is known about the leak inherent to the discrete logarithm used in amount decryption; however, that leak impacts only amounts, for which timing attacks are harder to leverage due to the non-repeated nature of individual transactions.



NCC Group recommends that the protocol papers be completed with specifications of the instructions and zero-knowledge proofs that they do not currently describe, such as the TransferWithFee instruction.

Retest Summary

Solana Foundation implemented a number of changes to address NCC Group's findings. NCC Group expended another five days to retest these changes at the end of February 2023. During the initial assessment, NCC Group identified:

- One (1) high-severity vulnerability,
- Seven (7) low-severity vulnerabilities, and
- Three (3) informational findings.

Upon completion of the assessment, all findings were reported to Solana Foundation, along with recommendations. After retesting, seven findings were found to be fully fixed. Of the remaining four findings:

- Three (3) low-severity vulnerabilities were considered an acceptable risk by Solana Foundation, and
- One (1) informational finding was considered an acceptable risk by Solana Foundation.



2 Dashboard


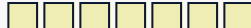

Target Data

Name	SPL Confidential Token Extension
Type	Library
Platforms	Rust

Engagement Data

Type	Implementation Review
Method	Code-assisted
Dates	2022-08-08 to 2022-08-19
Consultants	2
Level of Effort	20 person-days









Finding Breakdown

Critical issues	0	
High issues	1	
Medium issues	0	
Low issues	7	
Informational issues	3	
Total issues	11	

Category Breakdown

Cryptography	2	
Data Validation	1	
Uncategorized	8	

Component Breakdown

AuthenticatedEncryption	1	
DiscreteLog	2	
ElGamal	2	
ElGamal, AuthenticatedEncryption	1	
Instruction	1	
RangeProof	1	
SigmaProofs	1	
ZkTokenElGamal	2	

 Critical  High  Medium  Low  Informational



3 Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors.

Title	Status	ID	Risk
Parsing of SPL ZK-Token Protocol and Cryptographic Key Data May Crash System	Fixed	EY7	High
Non-Constant-Time Signature Comparison May Leak Some Information About Derived AES Key	Fixed	3PM	Low
Multiple Timing Side-Channels in Discrete Log Computation May Reveal Amount	Risk Accepted	WP9	Low
Private Key JSON Decoding/Encoding is not Constant-Time	Risk Accepted	QLJ	Low
Amount Value Split Discrepancy Between In-Sealevel and Out-of-Sealevel Code	Fixed	4FA	Low
Encoded Scalars are Malleable	Fixed	XL7	Low
Secret Amount May Leak Through Side Channels in Fee Calculation	Risk Accepted	77E	Low
Inner Product Computations are not Constant-Time	Fixed	4D4	Low
Fragile Key Derivation from Ed25519 Signatures	Risk Accepted	BE6	Info
Discrete Logarithm Failure with Many Threads	Fixed	T7P	Info
Timing Side-Channels May Reveal Which Transaction Amounts Have Large Fees	Fixed	R64	Info



4 Finding Details

High

Parsing of SPL ZK-Token Protocol and Cryptographic Key Data May Crash System

Overall Risk	High	Finding ID	NCC-E004944-EY7
Impact	Medium	Component	ElGamal
Exploitability	High	Category	Data Validation
		Status	Fixed

Impact

An attacker can submit invalid data, including invalid ElGamal encrypted ciphertexts, to a ZK-Token process and crash it, resulting in a denial-of-service condition for users of this process. The impact is contingent on the execution environment, be it a Solana node operating system or a Solana smart contract VM; for the former, it may bring down an affected Solana node process and cause substantial disruption to the service, while for the latter, it may abort the execution of one smart contract.

Description

The SPL ZK-Tokens protocol aims to provide confidentiality of SPL token transactions. It does so by augmenting transactions with encrypted data, to hide the actual transactions' values, and zero-knowledge proof data, to prove certain properties about the encrypted transactions without revealing their plaintext values.

Protocol data and supporting cryptographic key data are mapped from bytes to the relevant Rust data structures using their respective `from_bytes()` methods. When performing this mapping, the SPL ZK-Tokens application does not validate that there is sufficient data in the Rust byte slices passed as arguments to these methods in several locations. An attacker may craft data that will result in out-of-bound slice access during the mapping, cause a Rust `panic`, and crash of the SPL ZK-Tokens application.

NCC Group identified the following vulnerable locations. In the first code snippet, of method `from_bytes()` of structure `ElGamalCiphertext`, the `array_ref` macros extract a subset of the slice passed as an argument of the specified lengths. If there are not enough data, the process will crash.

```
impl ElGamalCiphertext {
    // SNIP
    pub fn from_bytes(bytes: &[u8]) -> Option<ElGamalCiphertext> {
        let bytes = array_ref![bytes, 0, 64];
        let (commitment, handle) = array_refs![bytes, 32, 32];
        let commitment = CompressedRistretto::from_slice(commitment).decompress()?;
        let handle = CompressedRistretto::from_slice(handle).decompress()?;

        Some(ElGamalCiphertext {
            commitment: PedersenCommitment(commitment),
            handle: DecryptHandle(handle),
        })
    }
}
```



Method `from_bytes()` of structure `ElGamalKeypair` attempts to operate on a subset of the `bytes` variable method argument, without checking that it has sufficient data:

```
impl ElGamalKeypair {
  // SNIP
  pub fn from_bytes(bytes: &[u8]) -> Option<Self> {
    Some(Self {
      public: ElGamalPubkey::from_bytes(bytes[..32].try_into().ok())?,
      secret: ElGamalSecretKey::from_bytes(bytes[32..].try_into().ok())?,
    })
  }
}
```

Recommendation

Ensure that the aforementioned `from_bytes()` methods validate that it has exactly 64 bytes of data before operating on it. Otherwise, the protocol should ignore the data.

Location

- Method `from_bytes()` of structure `ElGamalKeypair` in file `elgamal.rs`, line 197
- Method `from_bytes()` of structure `ElGamalCiphertext` in file `elgamal.rs`, line 434

Retest Results

2023-02-27 – Fixed

NCC Group reviewed the changes implemented in [PR #27389](#), and found that Solana Foundation implemented NCC Group's recommended fixes. The `from_bytes()` methods of the `ElGamalCiphertext` and `ElGamalKeypair` structs now validate that it has exactly 64 bytes of data before operating on it, thus preventing a process crash.

Furthermore, NCC Group notes that Solana Foundation has implemented additional input size checks on several methods of other structures, including `ElGamalPubkey`, `ElGamalSecretKey`, `DecryptHandle`, `PedersenCommitment`, and `proofs`. These additional changes do not appear to introduce further security issues.



Non-Constant-Time Signature Comparison May Leak Some Information About Derived AES Key

Overall Risk Low

Impact High

Exploitability Low

Finding ID NCC-E004944-3PM

Component AuthenticatedEncryption

Category Cryptography

Status Fixed

Impact

An attacker may be able to recover the first few bytes of the Token program AES key, if the generated key starts with bytes of value zero. This may in turn facilitate the recovery of the AES private key employed to encrypt the balance of an account, in the current implementation.

Description

An account's available balance is currently encrypted using Twisted ElGamal for processing of zero-knowledge proofs about the encrypted amount, and using AES-GCM-SIV for decryption by the Token program. The AES encryption key is generated from the Ed25519 signature of a message consisting of several dynamic and static parameters including the Token program public key, using the client signing private key.

As a result, the resulting signature can be considered secret data, which is usually not the case in typical deployment scenarios. This computed signature is compared with the default `Signature` structure value using the `==` comparison operator. This structure is a wrapper around a Rust generic array `GenericArray<u8, U64>` data structure, with its length parametrized with value `64`. The comparison operator `==` is not overloaded with another implementation, so the code will resort to a non-constant-time comparison of the computed signature (in effect the AES key), with an array filled with bytes of value zero, as illustrated below:

```
pub struct AeKey([u8; 16]);
impl AeKey {
    pub fn new(signer: &dyn Signer, address: &Pubkey) -> Result<Self, SignerError> {
        let message = Message::new(
            &[Instruction::new_with_bytes(*address, b"AeKey", vec![])],
            Some(&signer.try_pubkey()?),
        );
        let signature = signer.try_sign_message(&message.serialize())?;

        // Some `Signer` implementations return the default signature, which is not suitable
        // for
        // use as key material
        if signature == Signature::default() {
            Err(SignerError::Custom("Rejecting default signature".into()))
        } else {
            Ok(AeKey(signature.as_ref()[..16].try_into().unwrap()))
        }
    }
}
```



The comparison will complete faster if the signature's initial bytes do not match the default signature's initial bytes of value zero, and will be slower otherwise. The granularity of comparison is typically not in bytes, but in larger chunks, depending on the compiler and run-time environment; this may reduce the likelihood of a successful attack.

Recommendation

Implement and use a `Signature` structure comparison operator to run in constant-time, no matter what bytes differ. Note that overloading the existing `==` comparison operator may affect the performance of comparing public signature values in other areas of the system, and is not advised.

Location

Method `new()` of structure `AeKey` in file `auth_encryption.rs`, line 74

Retest Results

2023-02-27 – Fixed

NCC Group reviewed the changes implemented in [PR #27364](#), and found that Solana Foundation implemented NCC Group's recommended fix. The signature is now compared in constant-time using `crate::subtle::ConstantTimeEq`.

Furthermore, NCC Group notes that Solana Foundation has implemented another constant-time comparison of signatures in the `new()` method of struct `ElGamalKeypair`. The code changes do not appear to introduce further security issues.



Multiple Timing Side-Channels in Discrete Log Computation May Reveal Amount

Overall Risk Low

Impact High

Exploitability Low

Finding ID NCC-E004944-WP9

Component DiscreteLog

Category Cryptography

Status Risk Accepted

Impact

An attacker who can monitor the node memory cache may be able to determine the amount of a transaction being decrypted.

Description

The `decode_range()` function solves the discrete logarithm problem, as a final step of the Twisted ElGamal decryption process, to retrieve the confidential amount of a transaction. The computation can be distributed amongst a configurable number of threads. Precomputed results are stored in a Rust `HashMap`, which maps hash key $2^{16} * x_{hi} * G$ to hash value x_{hi} , where x_{hi} is the most significant 16 bits of the 32-bit amount value to uncover. `decode_range()` then iterates with x_{lo} , the least significant remaining bits of the amount value, from 0 to `range_bound`, which is 2^{16} divided by the number of threads. In each iteration, it checks if the hash table contains the value $C - x_{lo}$ where C is the value to solve the discrete logarithm for, and if it is the case, computes the amount value as $x_{lo} + 2^{16} * x_{hi}$. The function code is listed below:

```
fn decode_range(ristretto_iterator: RistrettoIterator, range_bound: usize) -> Option<u64> {
    let hashmap = &DECODE_PRECOMPUTATION_FOR_G;
    let mut decoded = None;
    for (point, x_lo) in ristretto_iterator.take(range_bound) {
        let key = point.compress().to_bytes();
        if hashmap.0.contains_key(&key) {
            let x_hi = hashmap.0[&key];
            decoded = Some(x_lo + TW016 * x_hi as u64);
        }
    }
    decoded
}
```

The code checks the hash table for the presence of the key with value $C - x_{lo}$, using an `if` conditional statement, which is not a constant-time operation, and is susceptible to side-channel attacks. Common processors use caches to speed up access to resources such as data and code. Attackers who can monitor the cache, for example and hypothetically from a virtual machine running on the same hypervisor as the virtual machine running the Solana ZK Tokens program, may observe changes in speed and cache behavior, when resources that depend on sensitive information are used. This attack can reveal the locations where the victim is accessing data (data flow), or in the case of this conditional statement, the code the victim is running and when (control flow).

Moreover, hash table operations are inherently non-constant-time (data-flow and control-flow, typically), and may reveal the secret hash key in case of a match (and/or by ruling out non-matches). Furthermore, if the hash table key being queried is present, then the hash



table is accessed again to read the actual `x_hi` value, based on the secret `C - x_lo` value. This operation amplifies the exploitability of the aforementioned control-flow side-channel.

Furthermore, the distribution of these non-constant-time operations across several threads may reveal what threads find or do not find the solution to the discrete logarithm problem.

Recommendation

Timing side-channels arising from conditional execution of code based on secret values can typically be addressed by performing branchless operations instead.

Side channels inherent to hash table structures may be addressed by performing a whole table scan, or using privacy techniques such as Path ORAM, which would be costly in this case.

Location

Function `decode_range()` in file `discrete_log.rs`, line 128

Retest Results

2023-02-27 – Not Fixed

Solana Foundation deems the risk of this finding to be acceptable, and did not fix this issue.



Private Key JSON Decoding/Encoding is not Constant-Time

Overall Risk Low

Impact Medium

Exploitability Low

Finding ID NCC-E004944-QLJ

Component ElGamal

Category Uncategorized

Status Risk Accepted

Impact

An attacker able to measure the precise timing of operations when a private key is decoded from a JSON object, or encoded to a JSON object, may learn partial information on the private key.

Description

ElGamal key pairs can be encoded to JSON objects, and decoded back from JSON files, using the functions in `encryption/elgamal.rs`, lines 204 and 218:

```
/// Reads a JSON-encoded keypair from a `Reader` implementor
pub fn read_json<R: Read>(reader: &mut R) -> Result<Self, Box<dyn std::error::Error>> {
    let bytes: Vec<u8> = serde_json::from_reader(reader)?;
    Self::from_bytes(&bytes).ok_or_else(|| {
        std::io::Error::new(std::io::ErrorKind::Other, "Invalid ElGamalKeypair").into()
    })
}

// ...

/// Writes to a `Write` implementer with JSON-encoding
pub fn write_json<W: Write>(
    &self,
    writer: &mut W,
) -> Result<String, Box<dyn std::error::Error>> {
    let bytes = self.to_bytes();
    let json = serde_json::to_string(&bytes.to_vec())?;
    writer.write_all(&json.clone().into_bytes())?;
    Ok(json)
}
```

The encoding/decoding backend is `serde_json`, which is invoked over the type `Vec<u8>`. The actual encoding format is a JSON array where each byte value is an integer, represented in decimal, e.g. as follows:

```
[207,131,225,53,126,239,184,189,241,84,40,80,214,109,128,7,214,32,228,5,11,87,21,220,131,244,16
↳ 9,33,211,108,233,206,71,208,209,60,93,133,242,176,255,131,24,210,135,126,236,47,99,185,49,189
↳ ,71,65,122,129,165,56,50,122,249,39,218,62]
```

The encoding process will then perform multiple conversions to decimal and output for each byte a number of digits that depends on the byte value; this is done with conditional jumps in the encoder. Similarly, upon decoding, the comma characters that separate successive values will be detected and treated differently, again using conditional jumps. This implies that the overall memory access pattern, both for instructions and for data, as well as the execution time of the decoding/encoding of each byte, will depend on the



values of the private key bytes. The overall execution time of the decoding or encoding of a key pair, and the resulting encoded key pair length, will also vary depending on the value of the private key bytes.

An attacker who is in position to make precise and repeated timing measurements may be able to obtain the length of the decimal encoding of each byte value. In particular, each byte that encodes to a single character (value between 0 and 9) is “worth” about 4.68 bits of information to the attacker. In total, such an attacker may hope for getting information mathematically equivalent to between 43.4 and 149.7 bits of entropy about the private key.

Recommendation

Private keys should be serialized and deserialized as opaque bytes. Text-based formats such as JSON make such handling challenging; the `ct-codecs` crate can be used to perform constant-time Base64 encoding and decoding of arbitrary sequences of bytes. The JSON encoder and decoder may still leak some information if they are using a look-up table mechanism to process string contents (e.g. to efficiently identify escape sequences or the end of the string); constant-time Base64 processing is only a partial mitigation. In general, it is best for the safety of secret data such as cryptographic keys if JSON or similar text formats are not used at all.

Location

Methods `read_json()` and `write_json()` of structure `ElGamalKeypair` in file `elgamal.rs`, [line 204](#) and [line 212](#)

Retest Results

2023-02-27 – Not Fixed

Solana Foundation deems the risk of this finding to be acceptable, and did not fix this issue.



Amount Value Split Discrepancy Between In-Sealevel and Out-of-Sealevel Code

Overall Risk	Low	Finding ID	NCC-E004944-4FA
Impact	Medium	Component	ZkTokenElGamal
Exploitability	Undetermined	Category	Uncategorized
		Status	Fixed

Impact

Some computations performed both inside the Sealevel parallel runtime (by smart contracts) and outside the runtime may disagree on how amount values are rebuilt from their two 32-bit halves.

Description

In `src/zk_token_elgamal/ops.rs`, the functions `add_with_lo_hi()` and `subtract_with_lo_hi()` compute additions and subtractions on ElGamal-encrypted amounts, leveraging the homomorphic encryption property. Namely, the functions add (or subtract) the encrypted amount `ct_1` to/from the encrypted amount `ct_0`. The amount `ct_1` is itself provided as two ciphertexts `ct_1_lo` and `ct_1_hi`, each encrypting the low and high 32 bits of the 64-bit amount, respectively. The functions, in the out-of-Sealevel section of the code, recompute `ct_1` from `ct_1_lo` and `ct_1_hi` using an appropriate linear combination (line 27):

```
#[cfg(not(target_os = "solana"))]
mod target_arch {
    // ...
    pub const TWO_32: u64 = 4294967296;

    // ...
    pub(crate) fn combine_lo_hi(
        ct_lo: &pod::ElGamalCiphertext,
        ct_hi: &pod::ElGamalCiphertext,
    ) -> Option<pod::ElGamalCiphertext> {
        add_ciphertexts(Scalar::one(), ct_lo, Scalar::from(TWO_32), ct_hi)
    }
}
```

In the in-Sealevel code, though, the linear coefficient is equal to 2^{16} instead of 2^{32} (lines 132 and 160):

```
#[cfg(target_os = "solana")]
#[allow(unused_variables)]
mod target_arch {
    // ...
    const SHIFT_BITS: usize = 16;

    // ...
    pub fn add_with_lo_hi(
        left_ciphertext: &pod::ElGamalCiphertext,
        right_ciphertext_lo: &pod::ElGamalCiphertext,
        right_ciphertext_hi: &pod::ElGamalCiphertext,
    ) -> Option<pod::ElGamalCiphertext> {
        let shift_scalar = to_scalar(1_u64 << SHIFT_BITS);
```



```
let shifted_right_ciphertext_hi = scalar_ciphertext(&shift_scalar,
↳ &right_ciphertext_hi?);
let combined_right_ciphertext = add(right_ciphertext_lo,
↳ &shifted_right_ciphertext_hi?);
add(left_ciphertext, &combined_right_ciphertext)
}
```

Thus, the two code versions do not compute the same final amount (except for very small amounts, up to 65535). The out-of-Sealevel code seems to be the correct version, since the specification of the protocol indeed calls for splitting 64-bit amounts into 32-bit halves.

Recommendation

The `SHIFT_BITS` value should be 32, to align with the out-of-Sealevel code and the specification.

Location

`src/zk_token_elgamal/ops.rs`, line 103

Retest Results

2023-02-27 – Fixed

NCC Group reviewed the changes implemented in [PR #28470](#), and found that Solana Foundation modified the code to be agnostic of the target (inside or outside of the Sealevel parallel runtime). The new common code now computes `left_ciphertext + (right_ciphertext_lo + 216 * right_ciphertext_hi)`. The function `combine_lo_hi()` was removed from the implementation.



Encoded Scalars are Malleable

Overall Risk	Low	Finding ID	NCC-E004944-XL7
Impact	Undetermined	Component	ZkTokenElGamal
Exploitability	Undetermined	Category	Uncategorized
		Status	Fixed

Impact

Scalar values accept several distinct encodings; this can induce some malleability of transactions, depending how the encoded scalars are used. Extra malleability has proven to induce weaknesses in some protocols.

Description

The `PodScalar` type is defined in `src/curve25519/scalar.rs` as a wrapper around 32 bytes, which nominally encode a scalar value (i.e., an integer modulo the prime order L of the Ristretto255 group). Conversion functions from that encoded format to the `Scalar` type of the `curve25519-dalek` library are provided in [src/curve25519/scalar.rs, line 17](#):

```
impl From<&PodScalar> for Scalar {
    fn from(pod: &PodScalar) -> Self {
        Scalar::from_bits(pod.0)
    }
}
```

and also in [src/zk_token_elgamal/convert.rs, line 84](#):

```
impl From<PodScalar> for Scalar {
    fn from(pod: PodScalar) -> Self {
        Scalar::from_bits(pod.0)
    }
}
```

The `Scalar::from_bits()` function in `curve25519-dalek` performs a *non-canonical decoding*: the provided bit pattern is used without checking whether it is a value already in the 0 to $L-1$ range. Since L is about 2^{252} , this means that the `from_bits()` function accepts up to 16 distinct encodings for a given scalar, that all yield *in fine* the same mathematical value.

This kind of malleability is usually not a problem, but it has occasionally led to trouble, especially in consensus protocols where different actors would disagree on the validity of a given non-canonical input. It also opens the possibility of making seemingly distinct sequences of bytes that are still, algebraically, the same object, which has also enabled attacks in some protocols. Note that for Ristretto points, the canonical encoding and decoding process is enforced, thereby avoiding any similar issue for group elements.

Recommendation

Decoding of a scalar from bytes should enforce a canonical encoding; this can be done by using the `Scalar::from_canonical_bytes()` function of `curve25519-dalek`.

Location

- [src/curve25519/scalar.rs, line 17](#)
- [src/zk_token_elgamal/convert.rs, line 84](#)



Retest Results

2023-02-27 – Fixed

NCC Group reviewed the changes implemented in [PR #28870](#), and found that Solana Foundation implemented NCC Group's recommended fix. The two aforementioned locations now decode a scalar from bytes using `Scalar::from_canonical_bytes()`, which enforces a canonical encoding.



Secret Amount May Leak Through Side Channels in Fee Calculation

Overall Risk Low

Impact Low

Exploitability Low

Finding ID NCC-E004944-77E

Component Instruction

Category Uncategorized

Status Risk Accepted

Impact

An attacker able to make precise timing measurements on the behavior of a target system may infer information about the amount in a transaction through side channels in the fee calculation.

Description

In a `TransferWithFee` instruction, the fee is computed by applying a configurable rate and a maximum value to the transaction amount; both the amount and the fee are secret values. The fee is computed in `src/instruction/transfer_with_fee.rs`, lines 697-709:

```
#[cfg(not(target_os = "solana"))]
fn calculate_fee(transfer_amount: u64, fee_rate_basis_points: u16) -> Option<(u64, u64)> {
    let numerator = (transfer_amount as u128).checked_mul(fee_rate_basis_points as u128)?;
    let mut fee = numerator.checked_div(ONE_IN_BASIS_POINTS)?;
    let mut delta_fee = 0_u128;

    let remainder = numerator.checked_rem(ONE_IN_BASIS_POINTS)?;
    if remainder > 0 {
        fee = fee.checked_add(1)?;

        let scaled_fee = fee.checked_mul(ONE_IN_BASIS_POINTS)?;
        delta_fee = scaled_fee.checked_sub(numerator)?;
    }

    let fee = u64::try_from(fee).ok()?;
    Some((fee as u64, delta_fee as u64))
}
```

Some operations in this function take a variable amount of time that depends on the source operands, or perform a memory access pattern at addresses that depend on the source operands:

- Integer divisions (calls to `checked_div()` and `checked_rem()`) involve CPU opcodes that may have a varying execution time, because they apply algorithms with a number of steps proportional to the size difference between the dividend and the divisor; they may also implement some optimizations when the divisor is a power of two. For instance, on Intel x86 CPUs of the “Skylake” series (up to and including “Coffee Lake” cores), the 64-bit unsigned division opcode `DIV` takes between 35 and 88 cycles to complete, depending on the input values.
- The conditional jump on the remainder value will lead to a different execution time (some instructions will be skipped) and a different memory access pattern (some instructions will not be fetched from RAM) if the value happens to be a multiple of the fee rate, leading to a remainder of value zero.



An attacker in position to perform precise timing measurements (e.g. controlling a virtual machine co-hosted on the same hardware as the target system) may leverage these leaks to obtain some information on the amount.

Recommendation

The test on the remainder can be avoided by adding `ONE_IN_BASIS_POINTS - 1` to the numerator prior to the division; with this change, a rounding-up division is performed, which is what the function aims at achieving. The computation of `delta_fee` would then be done systematically.

Since the divisor is non-secret (it is part of the public configuration), the division operation itself can be replaced with a multiplication followed by a shift, as described by [Grandlund and Montgomery](#). In this case, the dividend (numerator) may use up to 80 bits, and that method entails computing an $80 \times 80 \rightarrow 160$ multiplication, which exceeds the size of the largest unsigned integer type available in Rust. The multiplication thus would have to be performed using 64-bit limbs and manual carry propagation. The Rust compiler *may* already implement this method in the generated code (since the divisor, `ONE_IN_BASIS_POINTS`, is a constant known at compile-time) but this is not guaranteed and may depend on the target CPU architecture and model, the Rust compiler version, and the compilation flags.

Location

[src/instruction/transfer_with_fee.rs](#), lines 697-709

Retest Results

2023-02-27 – Partially Fixed

NCC Group reviewed the changes implemented in [PR #27356](#), and found that Solana Foundation implemented all but one of NCC Group's recommended fixes.

Solana has implemented the rounding-up division fix, but did not replace the division with a multiplication followed by a shift, leaving one potential side channel unaddressed. The remaining low side channel risk is considered acceptable by Solana Foundation.



Inner Product Computations are not Constant-Time

Overall Risk Low

Impact Low

Exploitability Low

Finding ID NCC-E004944-4D4

Component RangeProof

Category Uncategorized

Status Fixed

Impact

Attackers able to perform precise timing measurements on the behavior of a target system may infer information about private elements involved in range proofs, in particular transaction amounts.

Description

The `InnerProductProof::new()` function creates an inner-product proof; this is used as a core element of range proofs, involved in particular in demonstrating that encrypted amount values are in the proper range. The elements of some of the vectors on which inner product proofs are computed directly translate the bits of the binary representation of amount values.

The proof construction entails many linear combinations of elements of the Ristretto group; the linear coefficients are derived from the elements of the input vectors. To perform these linear combinations, the `curve25519-dalek RistrettoPoint::vartime_multiscalar_mul()` function is invoked in several places. However, that function is not constant-time; internally, it uses Straus's algorithm in combination with dynamically generated windows of point multiples, and a wNAF scalar representation. In Straus's algorithm, a common sequence of successive point doublings is performed, with occasional extra point additions; the exact places at which the point addition occurs in the sequence of point doublings depend on the position of non-zero digits in the wNAF representation, which itself depends on the exact value of the source scalars. Moreover, the points which are added are looked up from the windows at addresses that depend on the digit values in the wNAF scalar representation. Thus, the operation is not constant-time and will leak information detectable through timing-based side channels.

Note: `vartime_multiscalar_mul()` is also used in proof *verifications*; this is fine, since the verification uses only public data.

Recommendation

The *constant-time* implementation of Straus's algorithm should be used instead; it is implemented by `curve25519-dalek` as `RistrettoPoint::multiscalar_mul()`. That variant uses a non-wNAF scalar representation, so that point additions occur at times that do not depend on the source scalars; moreover, all window lookups are done in a constant-time way.

Location

[src/range_proof/inner_product.rs](#), lines 88, 102, 128, 132, 154, 160, 178, 179



Retest Results

2023-02-27 – Fixed

NCC Group reviewed the changes implemented in [PR #27355](#), and found that Solana Foundation implemented NCC Group's recommended fix. Solana Foundation replaced all aforementioned calls to `vartime_multiscalar_mul()` with `multiscalar_mul()`.



Fragile Key Derivation from Ed25519 Signatures

Overall Risk	Informational	Finding ID	NCC-E004944-BE6
Impact	Medium	Component	ElGamal, AuthenticatedEncryption
Exploitability	None	Category	Uncategorized
		Status	Risk Accepted

Impact

The derivation of secret keys from Ed25519 signatures may break, leading to loss of private keys and associated assets, if the signature generation engine ceases to use deterministic per-signature nonce generation.

Description

In `src/encryption/auth_encryption.rs`, a symmetric encryption key (for AES) is obtained by computing an Ed25519 signature over a conventional, fixed encoded message:

```
pub fn new(signer: &dyn Signer, address: &Pubkey) -> Result<Self, SignerError> {
    let message = Message::new(
        &[Instruction::new_with_bytes(*address, b"AeKey", vec![])],
        Some(&signer.try_pubkey()?),
    );
    let signature = signer.try_sign_message(&message.serialize());

    // Some `Signer` implementations return the default signature, which is not suitable
    ↳ for
    // use as key material
    if signature == Signature::default() {
        Err(SignerError::Custom("Rejecting default signature".into()))
    } else {
        Ok(AeKey(signature.as_ref()[..16].try_into().unwrap()))
    }
}
```

A similar mechanism is used to derive an ElGamal private key in `src/encryption/elgamal.rs`, [line 173](#) and [line 340](#).

The idea behind this construction is to try to obtain private keys for new cryptographic operations based on an existing private key storage system (e.g. a wallet) that supports only Ed25519 signature generation. Ed25519 signature generators *usually* employ a [deterministic signature generation process](#) in which the per-signature nonce value is obtained as a hash over the signature key pair and the message to sign. Ed25519 signatures are nominally a randomized signature scheme; the deterministic generation is known as *derandomization* and its main benefit is that it ensures cryptographic safety even if no high-quality cryptographic random number generator is available.

However, whether the per-signature nonce was chosen deterministically or randomly cannot be detected by verifiers; a signature generator may also use a randomized process and still output valid and interoperable signature values. Moreover, it has been argued that purely deterministic signatures [increase vulnerability to fault attacks](#); thus, an implementation of an Ed25519 signature generator may legitimately switch its behavior



from deterministic to randomized. In fact, using an extra random seed inside the derandomization process, to maintain the mathematical safety in case the random generator turns out not to be of high enough quality. This would still conform to the relevant standards (e.g. [RFC 8032](#)). In such a case, the key derivation process implemented by Solana would break, in that a new secret key would be obtained at each execution, even if working with the same Ed25519 private key. This occurrence would be equivalent to a private key loss. In that sense, this key derivation process is fragile.

Apart from the risk of key loss, the use of signature values as the source for private keys may raise some extra security concerns, depending on the usage context:

- Implementations usually take great care to protect private keys against eavesdropping by attackers, but may not be as careful about signature values, which are normally considered public data. For instance, in usage scenarios where ulterior RAM scanning is considered to be a practical threat, it is customary to wipe private elements after usage (any in-RAM copy of the private key, and of the per-signature nonce), but the same treatment would not be applied to the signature value itself.
- The resulting private key remains private only as long as the signer cannot be convinced by an attacker to sign the exact same conventional message. Solana is using a consistent serialized message format throughout its SDK, and the conventional message for key derivation includes an explicit tag, which avoids collisions with any other use of signatures *within Solana*. If the private key owner uses the same private key in a different system, then such guarantees no longer apply.

Recommendation

Clearly document the limitations and potential fragility of the construction, and recommend another mechanism when possible.

Location

- [src/encryption/auth_encryption.rs, line 77](#)
- [src/encryption/elgamal.rs, line 173 and line 340](#)

Retest Results

2023-02-27 – Not Fixed

Solana Foundation deems the risk of this finding to be acceptable, and did not fix this issue.



Discrete Logarithm Failure with Many Threads

Overall Risk	Informational	Finding ID	NCC-E004944-T7P
Impact	High	Component	DiscreteLog
Exploitability	None	Category	Uncategorized
		Status	Fixed

Impact

If using 131072 threads or more for solving discrete logarithms, the implementation no longer finds the solution, and it reports a failure for all inputs.

Description

The `decode_32()` function in `src/encryption/discrete_log.rs` recovers a value from its decrypted ciphertext representation using a baby-step/giant-step discrete logarithm algorithm that leverages the fact that the solution is known to be in the limited range (0 to $2^{32}-1$). This process entails performing 2^{16} point additions and lookups; this is expensive, therefore the implementation allows for splitting the work over a configurable number of threads. Each thread then performs a fraction of the 2^{16} operations.

The workload split is computed in the `DiscreteLog::num_threads()` function (line 79):

```
/// Adjusts number of threads in a discrete log instance.
pub fn num_threads(&mut self, num_threads: usize) -> Result<(), ProofError> {
    // number of threads must be a positive power-of-two integer
    if num_threads == 0 || (num_threads & (num_threads - 1)) != 0 {
        return Err(ProofError::DiscreteLogThreads);
    }

    self.num_threads = num_threads;
    self.range_bound = (TW016 as usize).checked_div(num_threads).unwrap();
    self.step_point = Scalar::from(num_threads as u64) * G;

    Ok(())
}
```

The code checks explicitly that the number of threads is not zero, and is a power of two. (The latter condition is intended to ensure an equal split of the workload among all threads.) However, it does not check any upper bound on the number of threads. If that number is larger than 2^{16} (i.e. at least $2^{17} = 131072$, since it is still checked to be a power of two), then the division will yield a value of `range_bound` equal to zero, and none of the threads will perform any work, since the upper range limit is exclusive in the `decode_range()` function that the threads run. In total, the 131072 threads will exit quickly, and `decode_u32()` will return `None`, as if the input were invalid.

Recommendation

Enforce in `num_threads()` an upper limit of 65536 on the number of threads.

Location

`src/encryption/discrete_log.rs`, line 79



Retest Results

2023-02-28 – Fixed

NCC Group reviewed the changes implemented in [PR #27412](#), and found that Solana Foundation imposed an upper limit of 65536 on the number of threads, thus ensuring that the implementation finds a solution for the restricted input domain.



Timing Side-Channels May Reveal Which Transaction Amounts Have Large Fees

Overall Risk Informational

Impact Low

Exploitability Low

Finding ID NCC-E004944-R64

Component SigmaProofs

Category Uncategorized

Status Fixed

Impact

An attacker able to perform precise timing measurements on the behaviour of the target system may infer which transactions use amounts large enough to make the fee reach the configured maximum bound.

Description

In a `TransferWithFee` instruction, the fee is computed by applying a given configurable rate to the transaction amount. However, a maximum value is applied to the fee; for transactions with large amounts, the fee will be equal to that bound. Two fee proofs are produced for the transaction, one showing that the fee is equal to the maximum value, and the other showing that the fee is the same inside two distinct ciphertexts, which covers the case of a fee below the maximum value. The two proofs cannot be true simultaneously, so one of them is simulated; the aggregated proof shows that one of the two proofs is real, but does not reveal which one was simulated. The production system entails making *two pairs* of proofs, to cover both cases (`src/sigma_proofs/fee_proof.rs`, line 58):

```
let mut transcript_fee_above_max = transcript.clone();
let mut transcript_fee_below_max = transcript.clone();

// compute proof for both cases `fee_amount` >= `max_fee` and `fee_amount` < `max_fee`
let proof_fee_above_max = Self::create_proof_fee_above_max(
    fee_opening,
    delta_commitment,
    claimed_commitment,
    &mut transcript_fee_above_max,
);

let proof_fee_below_max = Self::create_proof_fee_below_max(
    fee_commitment,
    (delta_fee, delta_opening),
    claimed_opening,
    max_fee,
    &mut transcript_fee_below_max,
);
```

The `create_proof_fee_above_max()` and `create_proof_fee_below_max()` functions return a pair of sigma proofs (one true, one simulated). These functions also use, and modify, a transcript, to inject the relevant proof elements and obtain the challenge values. The two functions are called on freshly created clones of the current transcript.



The right pair of proofs is then selected in a constant-time way, to avoid leaking through side-channels whether the fee was equal to the maximum (i.e. the transaction amount was large) or below the maximum:

```
let below_max = u64::ct_gt(&max_fee, &fee_amount);

// conditionally assign transcript; transcript is not conditionally selectable
if bool::from(below_max) {
    *transcript = transcript_fee_below_max;
} else {
    *transcript = transcript_fee_above_max;
}

Self {
    fee_max_proof: FeeMaxProof::conditional_select(
        &proof_fee_above_max.fee_max_proof,
        &proof_fee_below_max.fee_max_proof,
        below_max,
    ),
    fee_equality_proof: FeeEqualityProof::conditional_select(
        &proof_fee_above_max.fee_equality_proof,
        &proof_fee_below_max.fee_equality_proof,
        below_max,
    ),
}
```

While the selection of `fee_max_proof` and `fee_equality_proof` is done with constant-time conditional selection, the choice of the correct transcript clone is done in a non-constant-time way, with a conditional evaluation that leads to a specific memory access pattern: only one of the two cloned transcript states is copied into the current transcript object. Thus, whether the source amount was large or not still leaks through timing-based side-channels.

Recommendation

The side-channel leak can be avoided by modifying the implementation in the following way: instead of selecting one of the two cloned transcripts, the two clones may be discarded, and the original transcript instance updated *after* the constant-time selection of the right pair of proofs. Indeed, the updates to the transcript are the insertion of the `Y_max_proof`, `Y_delta`, and `Y_claimed` points, and the extraction of the `c` and `w` challenge scalars; the points to insert are part of the finally returned `FeeSigmaProof` instance. This would lead to the following code:

```
let below_max = u64::ct_gt(&max_fee, &fee_amount);

// we discard the two transcript clones

let rv = Self {
    fee_max_proof: FeeMaxProof::conditional_select(
        &proof_fee_above_max.fee_max_proof,
        &proof_fee_below_max.fee_max_proof,
        below_max,
    ),
    fee_equality_proof: FeeEqualityProof::conditional_select(
        &proof_fee_above_max.fee_equality_proof,
        &proof_fee_below_max.fee_equality_proof,
        below_max,
```



```
    ),  
};  
  
// Update the transcript just like the verifier will do  
transcript.append_point(b"Y_max_proof", &rv.fee_max_proof.Y_max_proof);  
transcript.append_point(b"Y_delta", &rv.fee_equality_proof.Y_delta);  
transcript.append_point(b"Y_claimed", &rv.fee_equality_proof.Y_claimed);  
transcript.challenge_scalar(b"c");  
transcript.challenge_scalar(b"w");  
  
rv
```

Since hashing data into a transcript has very low cost compared to the elliptic curve operations used in the sigma proofs, the overhead induced by this recommendation should be negligible.

Location

[src/sigma_proofs/fee_proof.rs](#), line 80

Retest Results

2023-02-20 – Fixed

NCC Group reviewed the changes implemented in [PR #27354](#), and found that Solana Foundation implemented NCC Group's provided code fix.



5 Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

Rating	Description
Critical	Implies an immediate, easily accessible threat of total compromise.
High	Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.
Medium	A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.
Low	Implies a relatively minor threat to the application.
Informational	No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

Rating	Description
High	Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.
Medium	Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.
Low	Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

Rating	Description
High	Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.
Medium	



Rating	Description
	Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.
Low	Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

Category Name	Description
Access Controls	Related to authorization of users, and assessment of rights.
Auditing and Logging	Related to auditing of actions, or logging of problems.
Authentication	Related to the identification of users.
Configuration	Related to security configurations of servers, devices, or software.
Cryptography	Related to mathematical protections for data.
Data Exposure	Related to unintended exposure of sensitive information.
Data Validation	Related to improper reliance on the structure or values of data.
Denial of Service	Related to causing system failure.
Error Reporting	Related to the reporting of error conditions in a secure fashion.
Patching	Related to keeping software up to date.
Session Management	Related to the identification of authenticated users.
Timing	Related to race conditions, locking, or order of operations.



6 Audit Notes

In this section, we include extra remarks which were deemed worth reporting, but are not security vulnerabilities by themselves.

Discrete logarithm inefficiency: The implementation of the limited-range discrete logarithm, in `src/encryption/discrete_log.rs`, recovers the value x from the input point $C = xG$ by computing all points $C - iG$ (for $i = 0$ to $2^{16}-1$) and looking for the presence of these points in the precomputed table of points $j(2^{16}G)$ (for $j = 0$ to $2^{16}-1$). The function `decode_range()` performs this operation:

```
fn decode_range(ristretto_iterator: RistrettoIterator, range_bound: usize) -> Option<u64> {
    let hashmap = &DECODE_PRECOMPUTATION_FOR_G;
    let mut decoded = None;
    for (point, x_lo) in ristretto_iterator.take(range_bound) {
        let key = point.compress().to_bytes();
        if hashmap.0.contains_key(&key) {
            let x_hi = hashmap.0[&key];
            decoded = Some(x_lo + TW016 * x_hi as u64);
        }
    }
    decoded
}
```

The Ristretto point compression function is invoked for each point; this is by far the most expensive step of this process. Ristretto point compression involves a square root computation in the field, with a cost roughly equivalent to 200 field multiplications, while a point addition uses only eight field multiplications, and a few additions and subtractions, leading to a per-addition cost equivalent to about 10 field multiplications. Thus, point compression is 20 times more expensive than point addition; the lookup (in a hash table) is also much faster than a point addition and adds negligible overhead.

Point compression can be made much more efficient by using batches, in which many points are compressed at the same time. Raw Ristretto compression cannot by itself benefit from batches, but it can if coupled with a point doubling. Indeed, when compressing a Ristretto point $2P$, knowledge of its half P allows replacing the square root with an inversion in the field. Thanks to a trick due to Montgomery, a batch of independent inversions can be all computed using a single inversion in the field, and a per-value overhead of only three multiplications. If `decode_range()` performed compression by batches of, for instance, 128 points, then the overall discrete logarithm process could be sped up by a factor of up to 10 or so.

A few caveats apply to this batching process:

- As explained above, the optimization can be applied only if combined with a point doubling, which must be taken into account: the actual lookup will be with point $2(C - iG)$; hence, the precomputed table should contain values $2j(2^{16}G) = j(2^{17}G)$.
- Batch doubling-and-compression is a functionality provided by `curve25519-dalek` as the `RistrettoPoint::double_and_compress_batch()` function, which already implements all the necessary formulas. However, there is a [known issue](#) in that implementation, in that it can fail (with a panic) if one of the points to compress happens to be the Ristretto identity point (the neutral element in the group). A [patch](#) is available; until the patch is merged into the library, the callers should take care not to include the identity point into their batches. (The `is_identity()` function on a `RistrettoPoint` is a relatively efficient way to check whether a point is indeed the group neutral or not.)
- When a thread has found the correct solution, it may stop right away; in particular, if a single thread is involved, then this would halve the total computation time (on average).



However, such halting may happen only with the granularity of batches, so that larger batches are at odds with that optimization; in practice, batch size should be set to some value benchmarked to yield the best results, probably around 100 points or so. It shall be noted that early thread exit yields only moderate gains when several threads are used (since only one thread will benefit from it), and increases the side-channel leaks inherent to the discrete logarithm process, as described in [finding "Multiple Timing Side-Channels in Discrete Log Computation May Reveal Amount"](#).

Note: During the retest of the findings, NCC Group noticed that Solana Foundation has implemented point compression optimization using batches, with a batch size of 32 points in [PR #27412](#).

Invalid amount split: The `split_u64()` function is defined in `src/instruction/mod.rs` (line 42); it splits a given amount (`u64` value) into “low” and “high” parts, whose lengths (in bits) are provided as parameters:

```
pub fn split_u64(
    amount: u64,
    lo_bit_length: usize,
    hi_bit_length: usize,
) -> Result<(u64, u64), ProofError> {
    assert!(lo_bit_length <= 64);
    assert!(hi_bit_length <= 64);

    if !bool::from((amount >> (lo_bit_length + hi_bit_length)).ct_eq(&0u64)) {
        return Err(ProofError::TransferAmount);
    }

    let lo = amount << (64 - lo_bit_length) >> (64 - lo_bit_length);
    let hi = amount >> lo_bit_length;

    Ok((lo, hi))
}
```

The two assertions check that the two lengths are in sensible ranges. However, the checks are incomplete:

- If `lo_bit_length` is zero, then the two shifts by `64 - lo_bit_length` overflow the shift counter.
- If `lo_bit_length` is 64, then the right shift by `lo_bit_length` overflows the shift counter.
- If `lo_bit_length + hi_bit_length` is 64 or more, even if both lengths are individually lower than 64, then the shift by `lo_bit_length + hi_bit_length` overflows the shift counter.

The shift counter normal range, for a 64-bit source operand, is 0 to 63; any shift count outside of this range is an overflow. Overflows trigger a panic with Rust code compiled in debug mode; in release mode, the shift count would be silently truncated to its low 6 bits.

In practice, the low and high bit lengths are constants (denoted `TRANSFER_AMOUNT_LOW_BITS` and `TRANSFER_AMOUNT_HIGH_BITS`, respectively, in `src/instruction/transfer.rs` and `src/instruction/transfer_with_fee.rs`); thus, any invalid value should be detected at compile time, when running the unit tests. The current values are 16 bits for the low part, 32 bits for the high part. We note that some parts of the code in `transfer.rs` and `transfer_with_fee.rs` are meant to cover the case of both lengths being 32



bits, which will trigger an overflow in `split_u64()`; thus, this case is probably not meant to be used anymore.

Note: During the retest of the findings, NCC Group noticed that Solana Foundation made a number of changes to this function in [PR #28472](#):

- It removed the two assertions.
- If `bit_length` is 64 bits:
 - The function now returns `(amount, 0)`,
- Otherwise:
 - It returns: `(bit_length` low bits of `amount` , `64 - bit_length` high bits of `amount`).

Therefore, it may overflow if bit length is 0; however, currently this value does not appear to be passed as an argument to the function.

Withdraw operation may reveal information about balance: The withdrawn amount reveals information about the current balance, specifically that the current balance is greater than or equal to the withdrawn amount. It is envisaged that an attacker who can observe encrypted balances before and after account withdraw operations, for example by capturing transactions on the network, can infer this information by:

- guessing an amount,
- encrypting this amount with a Pedersen opening of value zero,
- and then subtracting this guessed and encrypted value from the previous ciphertext, until the result is equal to the new ciphertext.

Since the instruction processor must know the plain amount for any Withdraw operation, it is NCC Group's understanding that this information is inherently public. This remark is mostly about pointing out that even if some additional protocol mechanism was added to somehow convey the withdrawn amount confidentially to the instruction processor, then the use of a non-secret Pedersen commitment opening (zero, in this case) would make such a scheme fail.

Note: Solana Foundation has reviewed this audit note, and deems that the risk that Withdraw operations may reveal information about balance is acceptable.

Errors in comments:

- The comment on the `ciphertext_lo` field of the `TransferData` structure (in [src/instruction/transfer.rs](#), line 45) says that it contains the encryption of "the low 32 bits of the transfer amount", which is incorrect since `TRANSFER_AMOUNT_LOW_BITS` is 16, not 32.
- The comment on the `WithdrawWithheldTokensData` structure (in [src/instruction/withdraw_withheld.rs](#), line 24) specifies that the pre-instruction should call `WithdrawData::verify_proof(&self)`, which is the wrong type name (it should be `WithdrawWithheldTokensData::verify_proof(&self)`).
- In [src/range_proof/inner_product.rs](#), line 36, it is documented that `InnerProductProof::new()` shall be called only with vectors whose length is either a power of two, or zero. However, the code explicitly checks (on line 68) that the (common) length of the vector parameters is a power of two (with the standard `usize::is_power_of_two()` function), which is not the case of zero. Thus, vectors with length zero are not actually supported; they would lead to a panic triggered by the failed assertion on line 68.

