zk-token-sdk

Audit

Presented by:

OtterSec contact@osec.io

Akash Thota 0x4ka5h@osec.io

Tuyết Dương tuyet@osec.io

Robert Chen r@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Solana Foundation engaged OtterSec to perform an assessment of the `zk-token-sdk` program. This assessment was conducted between September 1st and September 26th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 9 findings in total.

In particular, we have identified several significant fund loss concerns during the transfer of amounts with associated fees. These encompass several edge cases in transferring an amount with a fee, potentially resulting in a depletion of funds from the recipient's account (OS-ZKT-ADV-00). Additionally, we have noted a lack of validation regarding the maximum fee value users may set during transfers, which may also result in financial losses (OS-ZKT-ADV-01).

Furthermore, we have drawn attention to an inconsistency in the verification process, which ensures that the number of commitments does not surpass the maximum allowable commitments (OS-ZKT-ADV-02), and also highlighted an overflow due to a right shift operation, resulting in a panic when dealing with u128/u256 range proofs where a bit length exceeds 64 (OS-ZKT-ADV-03).

We also made a recommendation around the implementation of proper error handling (OS-ZKT-SUG-01) and to include a limit on the maximum number of usable generators to prevent a possible overflow scenario (OS-ZKT-SUG-00). We further suggested implementing a check to ensure bit lengths are greater than zero to avoid the representation of zero bit lengths (OS-ZKT-SUG-02).

02 | Scope

The source code was delivered to us in a git repository at github.com/solana-labs/solana/tree/master/zk-token-sdk. This audit was performed against commit 9e703f8.

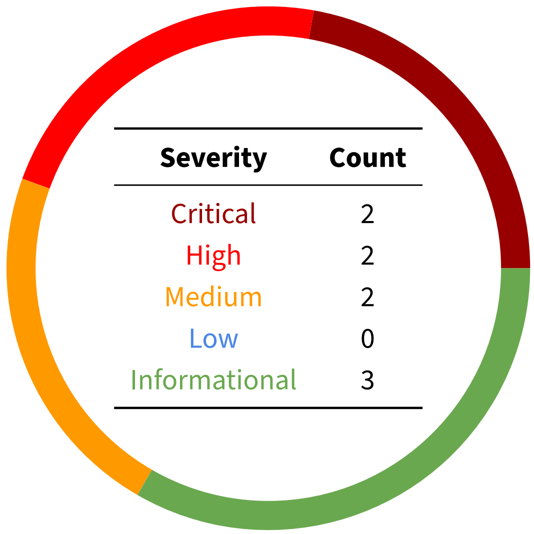
A brief description of the programs is as follows:

Name	Description
zk-token-sdk	This software development kit contains code that is utilized by the token-2022 extension.

03 | Findings

Overall, we reported 9 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
Critical	2
High	2
Medium	2
Low	0
Informational	3

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-ZKT-ADV-00	Critical	Resolved	While transferring amounts with a fee, edge cases may result in a loss of funds from the receiver's account.
OS-ZKT-ADV-01	Critical	Resolved	Lack of validation on the value of the maximum fee the user may set while performing a transfer results in a loss of funds.
OS-ZKT-ADV-02	High	Resolved	The check to ensure the number of commitments does not exceed `MAX_COMMITMENTS` is inconsistent.
OS-ZKT-ADV-03	High	Resolved	Panic due to a right shift operation, which overflows when dealing with u128/u256 range proofs.
OS-ZKT-ADV-04	Medium	Resolved	When `compression_batch_size` is set to zero, `decode_u32` panics.
OS-ZKT-ADV-05	Medium	Resolved	Possibility of panic in code while generating an ElGamal secret key from a high seed length.

OS-ZKT-ADV-00 [crit] | Edge Cases While Transferring With Fees

Description

The vulnerabilities pertain to specific edge cases when `delta_fee` is computed, where malicious provers may manipulate the parameters to create valid proofs that, when processed, result in a loss of funds from the destination account. This occurs as in both edge cases, proofs may be verified even though the value of `fee_to_encrypt` is greater than that of `transfer_amount`, thus resulting in the loss of funds scenario.

with_fee.rs.rs RUST

```rust
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)?;

    // Warning: Division may involve CPU opcodes that have variable execution times.
        ↪ This
    // non-constant-time execution of the fee calculation can theoretically reveal
        ↪ information
    // about the transfer amount. For transfers that involve extremely sensitive
        ↪ data, additional
    // care should be put into how the fees are calculated.
    let fee = numerator
        .checked_add(ONE_IN_BASIS_POINTS)?
        .checked_sub(1)?
        .checked_div(ONE_IN_BASIS_POINTS)?;

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

    Some((fee as u64, delta_fee as u64))
}
```

Proof of Concept

When a new `TransferWithFeeData` is generated, the following are the two edge cases' parameters that malicious provers may set and be verified successfully.

- **CASE 1:**
  - `transfer_amount` is set to zero.
  - `fee_to_encrypt` is set to one.
  - `delta_fee` is set to 10,000.
  - `fee_rate_basis_points` may be any value.

- **CASE 2:**
  - transfer_amount is set to x (where $x > 0$).
  - fee_to_encrypt is set to $x + 1$.
  - delta_fee is set to 10,000.
  - fee_rate_basis_points is set to 10,000.

Thus, in both cases, the set value of `delta_fee` is in a valid range, rendering the proof valid.

Remediation

Implement a range proof of `transfer_amount - fee_to_encrypt` or change the constant `MAX_FEE_BASIS_POINTS` to 9999.

Patch

Fixed in 3414.

OS-ZKT-ADV-01 [crit] | Missing Validation For Fee Amount

Description

In `process_set_transfer_fee`, `fee_to_encrypt` represents the fee to encrypt in a token confidential transfer, and it is meant to be deducted from the `transfer_amount` after it is successfully received at the destination account. The `maximum_fee` parameter the user sets defines the upper limit of the transfer fee.

transfer_fee/processor.rs RUST

```rust
fn process_set_transfer_fee(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    transfer_fee_basis_points: u16,
    maximum_fee: u64,
) -> ProgramResult {
    [...]
    // When setting the transfer fee, we have two situations:
    // * newer transfer fee epoch <= current epoch:
    //     newer transfer fee is the active one, so overwrite the older transfer
    ↪   fee with the newer one, then overwrite the newer transfer fee
    // * newer transfer fee epoch >= next epoch:
    //It was never used, so just overwrite the next transfer fee
    let epoch = Clock::get()?.epoch;
    if u64::from(extension.newer_transfer_fee.epoch) <= epoch {
        extension.older_transfer_fee = extension.newer_transfer_fee;
    }
    //Set two epochs ahead to avoid rug pulls at the end of an epoch
    and let newer_fee_start_epoch = epoch.saturating_add(2);
    let transfer_fee = TransferFee {
        epoch: newer_fee_start_epoch.into(),
        transfer_fee_basis_points: transfer_fee_basis_points.into(),
        maximum_fee: maximum_fee.into(),
    };
    extension.newer_transfer_fee = transfer_fee;

    Ok(())
}
```

The vulnerability arises as malicious provers may manipulate `fee_to_encrypt` to exceed `transfer_amount` in the proof. In the instance where `fee_to_encrypt` equals the `maximum_fee`, it only proves that `fee_to_encrypt = maximum_fee`, there is no inherent verification that `maximum_fee` is less than or equal to `transfer_amount`. This allows users to set any value for `maximum_fee`, potentially exceeding the `transfer_amount`.

Therefore, when a prover configures `fee_to_encrypt` to be equal to `maximum_fee`, and if `maximum_fee` exceeds the `transfer_amount`, the destination account will need to cover for `fee_to_encrypt`, resulting in a depletion of funds for the destination account.

Proof of Concept

1. User A initiates a token confidential transfer with the following parameters:
   - `transfer_amount`: 100 tokens.
   - `fee_to_encrypt`: 200 tokens.
   - `maximum_fee`: 200 tokens.
2. The transfer is processed, and `fee_to_encrypt` is set equal to `maximum_fee` as specified by the malicious prover.
3. The system then attempts to deduct the `fee_to_encrypt` from the destination account, but the account only possesses 100 tokens (the `transfer_amount`).
4. As a result, the remaining amount of 100 is deducted from the destination account, resulting in that account losing 100 tokens.

Here is an example unit test of the vulnerability:

fee_proof.rs RUST

```rust
fn test_fee_above_max_proof() {
        let transfer_amount: u64 = 1;
        let max_fee: u64 = 100;

        let fee_rate: u16 = 555; // 5.55%
        let fee_amount: u64 = 100;
        let delta: u64 = 998790; // 4*10000 - 55*555

        let (transfer_commitment, transfer_opening) =
            ↪  Pedersen::new(transfer_amount);
        let (fee_commitment, fee_opening) = Pedersen::new(max_fee);

        let scalar_rate = Scalar::from(fee_rate);
        let scalar_delta = Scalar::from(998790_u64);
        let (delta_commitment, delta_opening) = Pedersen::new(scalar_delta);
        let (claimed_commitment, claimed_opening) = Pedersen::new(0_u64);

        let mut prover_transcript = Transcript::new(b"test");
        let mut verifier_transcript = Transcript::new(b"test");

        let proof = FeeSigmaProof::new(
            (fee_amount, &fee_commitment, &fee_opening),
            (delta, &delta_commitment, &delta_opening),
            (&claimed_commitment, &claimed_opening),
            max_fee,
            &mut prover_transcript,
        );

        assert!(proof
            .verify(
                &fee_commitment,
                &delta_commitment,
                &claimed_commitment,
```



```
                    max_fee,
                    &mut verifier_transcript,
                )
                .is_ok());
        }
    }
```

Remediation

Implement a range proof of `transfer_amount - fee_to_encrypt`.

Patch

Fixed in 34314.

OS-ZKT-ADV-02 [high] | Inconsistent Check On Maximum Commitments

Description

The vulnerability pertains to the potential risk of a dishonest prover creating a crafted proof containing more commitments (bit-lengths) than the maximum permissible limit, resulting in incorrect verification without raising any error. This situation may occur as the number of commitments is checked before creating a proof to ensure they do not exceed the limit specified in `MAX_COMMITMENTS`. However, the verifier does not adequately check the length of the `bit_lengths` vector to ensure it is less than or equal to `MAX_COMMITMENTS` before verification.

batched_range_proof/mod.rs RUST

```rust
fn new(
    commitments: &Vec<&PedersenCommitment>,
    amounts: &Vec<u64>,
    bit_lengths: &Vec<usize>,
    openings: &Vec<&PedersenOpening>,
) -> Result<Self, ProofError> {
    // the number of commitments is capped at 8
    let num_commitments = commitments.len();
    if num_commitments > MAX_COMMITMENTS
        || num_commitments != amounts.len()
        || num_commitments != bit_lengths.len()
        || num_commitments != openings.len()
    {
        return Err(ProofError::Generation);
    }
```

Thus, the verifier then proceeds to verify the proof, believing it is a valid proof for the commitments provided. However, they verify additional commitments that are not part of the original commitment set.

Proof of Concept

1. The prover starts with a set of valid commitments [C_1, C_2, C_3] and their corresponding `bit_lengths` [32, 32, 64], where the total number of commitments is three, which is less than `MAX_COMMITMENTS` (8).
2. The dishonest prover creates a crafted proof where they include additional commitments [C_4, C_5, C_6, C_7, C_8, C_9] along with corresponding `bit_lengths`: [16, 16, 16, 16, 16]. Now, the total number of commitments is nine, more than `MAX_COMMITMENTS`.
3. When the verifier receives this crafted proof and attempts to verify it, there is no check to validate that the number of commitments (9) is less than `MAX_COMMITMENTS` (8), and successfully verifies the proof.

Remediation

Ensure to verify the length of `bit_lengths`, such that it is not greater than `MAX_COMMITMENTS`.

Patch

Fixed in 8bb153b.

OS-ZKT-ADV-03 [high]| Right Shift Overflow Panic

Description

`BatchedRangeProofU128Data::new` creates a new `BatchedRangeProofU128Data` instance, which includes generating a batched range proof for a set of commitments, values, bit lengths, and openings. It performs validation to ensure that the sum of bit lengths is 128 (128 bits in a 128-bit value), and it checks for potential overflows during bit length calculations. It is similar in `BatchedRangeProofU256Data::new`.

batched_range_proof_u128.rs RUST

```rust
#[cfg(not(target_os = "solana"))]
impl BatchedRangeProofU128Data {
    pub fn new(
        commitments: Vec<&PedersenCommitment>,
        amounts: Vec<u64>,
        bit_lengths: Vec<usize>,
        openings: Vec<&PedersenOpening>,
    ) -> Result<Self, ProofError> {
        // the sum of the bit lengths must be 64
        let batched_bit_length = bit_lengths
            .iter()
            .try_fold(0_usize, |acc, &x| acc.checked_add(x))
            .ok_or(ProofError::Generation)?;

        // `u64::BITS` is 128, which fits in a single byte and should not overflow
            ↪ to `usize` for
        // an overwhelming number of platforms. However, to be extra cautious, use
            ↪ `try_from` and
        // `unwrap` here. A simple case `u128::BITS as usize` can silently
            ↪ overflow.
        let expected_bit_length = usize::try_from(u128::BITS).unwrap();
        if batched_bit_length != expected_bit_length {
            return Err(ProofError::Generation);
        }
        [...]
    }
}
```

In `BatchedRangeProofU128Data::new`, a calculation of `batched_bit_length` represents the total number of bits required to represent the batched range of values. This calculation is performed by iterating through the `the bit_lengths` vector and summing all the bit lengths. The issue is related to the potential of right shift operations (>>) causing an overflow panic when dealing with u128 and u256 range proofs, especially when elements in the `bit_lengths` vector are greater than 64. This occurs as bit lengths are not validated, as shown in `BatchedRangeProofU128Data::new` above.

Proof of Concept

1. The protocol accepts a list of commitments that is less than or equal to `MAX_COMMITMENTS`.
2. Suppose we need to create a batched u128 range proof for a set of 3 commitments where one commitment has a bit length greater than 64.
3. Here is a simplified example of such data:

```
let bit_lengths = vec![70, 30, 28];
```

4. In this case, the `bit_lengths` vector contains three elements: 70, 30, and 28.
5. When the `batched_bit_length` is calculated, it will sum up these values:

```
let batched_bit_length = 70 + 30 + 28;
```

6. The value of `batched_bit_length` will be 128, which satisfies the u128 type, but the bit length for the first commitment is 70, which is greater than 64, which will panic.
7. Attempting to perform a right shift operation (>>) on this element would result in an overflow panic as u128 may only represent values up to $2^{128} - 1$.

Remediation

Ensure that a right shift operation will not overflow for bit lengths exceeding 64 bits.

Patch

Fixed in c155a20.

OS-ZKT-ADV-04 [med]| Panic On Setting Zero Batch Size

Description

In `discrete_log`, `set_compression_batch_size` enables users to set the compression batch size utilized during the discrete logarithm computation. The `compression_batch_size` parameter determines how many `Ristretto` points are processed together in each batch during the discrete logarithm computation. Currently, there is no check to ensure if `compression_batch_size` is set to zero, resulting in no points being processed in a batch.

handler_update_lending_market.rs RUST

```rust
/// Adjusts inversion batch size in a discrete log instance.
pub fn set_compression_batch_size(
    &mut self,
    compression_batch_size: usize,
) -> Result<(), DiscreteLogError> {
    if compression_batch_size >= TWO16 as usize {
        return Err(DiscreteLogError::DiscreteLogBatchSize);
    }
    self.compression_batch_size = compression_batch_size;
    Ok(())
}
```

The issue arises in `decode_u32`, designed to iterate through the `Ristretto` points in batches. Consequently, when the batch size is set to zero, this iteration logic breaks down, as there are no points to process in each batch. The code in `decode_u32` expects to process a non-empty batch of points and is unable to handle the scenario of an empty batch properly, resulting in a panic.

Remediation

Ensure that `compression_batch_size` is always set to a positive value greater than zero. Add a validation check in `set_compression_batch_size` to prevent setting it to zero:

discrete_log.rs RUST

```rust
/// Adjusts inversion batch size in a discrete log instance.
pub fn set_compression_batch_size(
    &mut self,
    compression_batch_size: usize,
) -> Result<(), DiscreteLogError> {
    if compression_batch_size == 0 {
        return Err(DiscreteLogError::DiscreteLogBatchSize);
    }
    [...]
}
```

Patch

Fixed in 4c0dc00.

OS-ZKT-ADV-05 [med]| Lack Of Restriction On Seed Length

Description

The issue arises due to a lack of explicit checks or restrictions on the length of the seed input, which may result in issues, including a stack overflow or panic if the seed length is very high. The code recursively calls `Self::from_seed(seed)?`, resulting in a panic.

elgamal.rs RUST

```rust
impl SeedDerivable for ElGamalSecretKey {
    fn from_seed(seed: &[u8]) -> Result<Self, Box<dyn error::Error>> {
        let key = Self::from_seed(seed)?;
        Ok(key)
    }
    [...]
}
```

Remediation

Implement a check that ensures the seed length does not exceed a preset value.

Patch

Fixed in 33700.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-ZKT-SUG-00	Possibility of overflow if the value of `gens_capacity` becomes exceedingly large.
OS-ZKT-SUG-01	Ensure proper error handling instead of panicking when vector lengths are unequal in `inner_product`.
OS-ZKT-SUG-02	Unnecessary representation of zero-bit numbers in `bit-lengths` vector.

OS-ZKT-SUG-00 | Panic Due To Overflow

Description

In the `BulletproofGens` structure, the `gens_capacity` field represents the maximum number of usable generators. However, if `gens_capacity` becomes too large, i.e., it holds `usize::MAX`, it may result in an overflow issue due to the limited capacity of the underlying data structures.

range_proof/generators.rs RUST

```rust
/// Increases the generators' capacity to the amount specified.
/// If less than or equal to the current capacity, it does nothing.
pub fn increase_capacity(&mut self, new_capacity: usize) {
    if self.gens_capacity >= new_capacity {
        return;
    }

    let label = [b'G'];
    self.G_vec.extend(
        &mut GeneratorsChain::new(&[label, [b'G']].concat())
            .fast_forward(self.gens_capacity)
            .take(new_capacity - self.gens_capacity),
    );

    self.H_vec.extend(
        &mut GeneratorsChain::new(&[label, [b'H']].concat())
            .fast_forward(self.gens_capacity)
            .take(new_capacity - self.gens_capacity),
    );

    self.gens_capacity = new_capacity;
}
```

In `increase_capacity`, attempting to increase the capacity to a value that is larger than `usize::MAX`, which is the maximum value that a `usize` type may hold, will result in an overflow. Specifically, the addition operation `self.gens_capacity + new_capacity` may overflow, resulting in the capacity wrapping around to a smaller value.

Remediation

Limit the value of `gens_capacity` by introducing a maximum limit.

Patch

Fixed in 34166.

OS-ZKT-SUG-01 | Error Handling

Description

In `range_proof/util`, `inner_product`, computes the inner product of two vectors by multiplying their corresponding elements and summing up the results. It ensures that the input vectors have the same length and returns the inner product as a scalar value.

range_proof/util.rs RUST

```rust
/// Computes an inner product of two vectors
/// Panics if the lengths of a and b are not equal.
pub fn inner_product(a: &[Scalar], b: &[Scalar]) -> Scalar {
    let mut out = Scalar::zero();
    if a.len() != b.len() {
        panic!("inner_product(a,b): lengths of vectors do not match");
    }
    for i in 0..a.len() {
        out += a[i] * b[i];
    }
    out
}
```

However, its current implementation panics instead of returning an error, crashing the program execution.

Remediation

Ensure an error is returned instead of panicking.

Patch

Fixed in 34065.

OS-ZKT-SUG-02 | Representation Of Zero Bit Numbers

Description

The issue is regarding how values with zero-bit representations. There are no zero-bit length numbers except zero. Passing zero-bit length elements in range proofs is possible, but there is no representation of a zero-bit number.

Even if the prover tries to pass a vector of bit-lengths: [16,16,16,16, 0, 0, 0, 0], to a range proof, `log(bit-lengths)` remains the same for the vectors without zeroes. While the program does not explicitly represent zero-bit numbers, it may effectively manage them within the protocol by optimizing the handling of values with zero bits.

Remediation

Implement a check to validate `bit-length > 0`.

Patch

Fixed in 34166.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings section.

Critical	Vulnerabilities that immediately result in a loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation. • Improperly designed economic incentives leading to loss of funds.
High	Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions. • Exploitation involving high capital requirement with respect to payout.
Medium	Vulnerabilities that may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
Low	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions.
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants. • Improved input validation.

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.