Solana Core

Audit

Presented by:

OtterSec contact@osec.io

Blas Kojusner blas@osec.io

Alec Petridis alec@osec.io

Harrison Green hgarrereyn@osec.io

Robert Chen notdeghost@osec.io

Contents

Appendices

01 | Executive Summary

Overview

OtterSec performed an assessment of the Versioned Transaction implementation and Address Lookup Table program. This assessment was conducted between August 1st and August 19th, 2022.

The audit scope consisted of a review of the pull requests outlined in the Core Implementation in issue #26391.

After delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation.

We delivered the final report August 23rd, 2022.

Key Findings

The following is a summary of the major findings in this audit.

- 4 findings total

02 | Scope

The code was delivered to us in several pull requests connected to issue #26391.

There were several components included in this audit. A brief description is as follows:

Name	Description
versioned transactions	A new upgradeable transaction format intended to replace the current (legacy) format. V0 of this format also supports address table references.
address-lookup-table	A new native on-chain program that contains instructions for managing address tables.
runtime	Modifications to runtime code to add support for address tables at the banking stage.

03 | Architecture

Address Lookup Table Program

Solana's networking stack uses an MTU size of 1280 bytes, which leaves 1232 bytes for packet data like serialized transactions. This constraint has led developers building applications on Solana to store state temporarily on-chain to consume it in a later transaction.

This workaround does not scale well when developers compose many on-chain programs in a single atomic transaction since more composition means more account inputs, each of which take up 32 bytes.

The address lookup table program has been proposed to increase the account limit in a single transaction. This program allows a protocol developer or an end user to create collections of related addresses on-chain for concise use in a transaction's account inputs.

Once the addresses are stored on-chain, they can be succinctly referenced in a transaction header using a 1-byte u8 index rather than the full 32-byte address. The address lookup tables are rent-exempt when initialized and after each time new addresses are appended.

Once an address lookup table is no longer needed, it can be deactivated and closed to have its rent balance reclaimed. Address lookup tables can be deactivated at any time but can continue to be used by transactions until the deactivation slot is no longer present in the slot hashes sysvar.

Versioned Transactions

There is a new transaction format which supports the use of on-chain address lookup tables to efficiently load more accounts into a single transaction. Considering the transaction types may need to be upgraded in the future, the format is an upgradeable versioned-transaction format.

The new format (which supports address table lookups) is `VersionedMessage::V0` and the old format is `VersionedMessage::legacy`.

The `v0` transaction format can be distinguished from the `legacy` transaction format by setting the upper bit of the u8 message header `num_required_signatures`. While the legacy format was not designed to be upgradeable, those transactions should never have this bit set normally and therefore it can be used to distinguish between the two formats.

Security Considerations

Lookup Table Immutability

While traditional account references contain an explicit account public key, address table lookups perform account retrieval indirectly using an address table account and a u8 account index.

In this second format, clients need to be assured that the contents of the address table at that particular index have not changed in the time between when the transaction was sent to the network and when the transaction was executed and included in the blockchain.

Therefore, an address at a particular index in an address table should be immutable for the duration of its lifetime. Additionally, it should not be possible to close and initialize a new address table at the same address (which may contain different values).

In Solana, this consideration is implemented by requiring address tables accounts to be PDAs using a recent slot hash as part of the seed. Address tables can not be closed until the deactivation slot hash is outside of recent history at which point it is not possible to re-initialize a new address table at the same address. Additionally, address tables are append-only and addresses can not be modified.

Resource Consumption

Enabling more account inputs in a transaction will require more resource consumption. Before address tables are enabled, transaction-wide compute limits on data reads and write locks are required.

Front Running

If the addresses listed within an address lookup table are mutable, front running attacks could modify which addresses are resolved for a later transaction. As a result, address lookup tables are append-only and may only be closed if it is no longer possible to create a new lookup table at the same derived address.

Duplicate Accounts

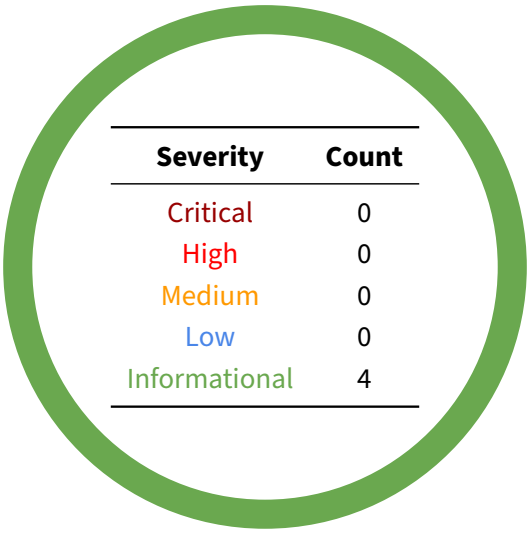
Transactions may not load an account more than once whether directly through `account_keys` or indirectly through `address_table_lookups`.

04 | Findings

Overall, we report 4 findings.

General findings don’t have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



Severity	Count
Critical	0
High	0
Medium	0
Low	0
Informational	4

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 do represent antipatterns and could introduce a vulnerability in the future.

ID	Status	Description
OS-SAT-SUG-00	TODO	Unnecessary padding field in lookup table
OS-SAT-SUG-01	TODO	Use checked arithmetic instead of saturating arithmetic
OS-SAT-SUG-02	TODO	Transactions cannot verify addresses loaded from lookup tables
OS-SAT-SUG-03	TODO	Serialize metadata instead of the whole account data

OS-SAT-SUG-00 | Unnecessary Padding Field

Description

The `padding` field in the `LookupTableMeta` struct is not necessary since data is bincode serialized. Enforcing pubkey data to start at `LOOKUP_TABLE_META_SIZE` already achieves 8-byte alignment.

programs/address-lookup-table/src/state.rs RUST

```
40      // Padding to keep addresses 8-byte aligned
41      pub _padding: u16,
42      // Raw list of addresses follows this serialized structure in
43      // the account's data, starting from `LOOKUP_TABLE_META_SIZE`.
```

Remediation

It is suggested to remove the parameter.

OS-SAT-SUG-01 | Replace Saturating Arithmetic

Description

Use of saturating arithmetic (`saturating_add`, `saturating_sub`, `saturating_mul`, etc...) causes values to “saturate” at the maximum or minimum possible representations.

Saturating arithmetic in the program analyzed is used to calculate lookup table size and the required lamports used for rent on an address table. During our analysis, we have found that all saturating arithmetic cases explored were enforced by auxiliary code before or after the arithmetic.

For example, in `src/processor.rs`, the value for `new_table_addresses_len` is calculated by adding the length of the new addresses in the table with the length of the current addresses in the table. This value is then checked against the constant `LOOKUP_TABLE_MAX_ADDRESSES`, however there should not be a case where it will overflow due to these checks.

programs/address-lookup-table/src/processor.rs RUST

```
260 let new_table_addresses_len = lookup_table
261     .addresses
262     .len()
263     .saturating_add(new_addresses.len());
```

Remediation

Replace saturating math with checked operations or plain math.

OS-SAT-SUG-02 | Transaction Address Verification

Description

The current transaction format adds support for lookup tables so that more addresses can be loaded in a single transaction without serializing the 32-byte address for each account. While traditional account references contain explicit account pubkeys, transactions which use adderess-tables perform account lookup indirectly.

An end-user may wish to explore which addresses were used in a particular instruction. For example, information about transactions is cached and made available through several off-chain resources (such as Solana Explorer). These tools allow end-users to gain insight into the activity on the blockchain and perform various types of analyses. It is particularly important to be able to observe which accounts were referenced in a specific transaction.

While observing account references is straightforward with the legacy transaction format (since this data is included in the header), it becomes more difficult with address tables. Specifically there are two cases:

1. While an address table is still open, a user can simply observe the state in the address table (which is immutable) in order to figure out which account was used.
2. After an address table has been closed, a user will need to replay all the transactions which constructed such an address table in order to determine its state at the time of the target transaction.

For case 2, this may be impractical or difficult for “light-node” users who may simply want to retrieve cached information from validators and validate portions of the data offline.

For example, if a user knows the transaction hash for a particular transaction, they can validate the contents of the transaction (which may have been provided by an untrusted party) by simply re-computing the hash. This user can subsequently be assured that the contents of the transaction header and embedded instructions are correct.

Currently, however there is no mechanism to validate the contents of an address-table. The only information included in the actual transaction is the address of the address-table and the indexes used by the transaction. Critically, there is a piece of information missing that could be used to validate the contents of such a table.

One easy approach could be to include both the address table pubkey and a hash of the address-table state inside the transaction header. Using the address table hash, an end user could validate which particular accounts were referenced if that information was provided by a separate party.

Remediation

Introduce a new transaction version format that includes a hash on the `_contents_` for each address table lookup so that the loaded addresses can be verified by the signer even after the lookup table is closed by simply comparing the hashes in question. The issue is being tracked at #26989.

OS-SAT-SUG-03 | Metadata Deserialization

Description

Under `AddressLookupTable::deserialize`, the whole account data is treated as serialized metadata when `bincode::deserialize` is invoked. This will treat the address data as serialized metadata which could potentially take up more space than is allocated. This is currently not a vulnerability since the types are included in the address-table metadata to prevent an overflow.

programs/address-lookup-table/src/state.rs RUST

```
190 /// Efficiently deserialize an address table without allocating
191 /// for stored addresses.
192 pub fn deserialize(data: &'a [u8]) -> Result<AddressLookupTable<'a>,
        ↪ InstructionError> {
193     let program_state: ProgramState =
194         bincode::deserialize(data).map_err(|_|
        ↪ InstructionError::InvalidAccountData)?;
```

Remediation

Only the metadata slice should be changed when `bincode::deserialize` is invoked to prevent accidentally treating the address data as serialized metadata.

A | Program Files

Below are the files in scope for this audit and their corresponding SHA256 hashes.

```
programs
  address-lookup-table
    Cargo.toml                                          963b340668da608ecb4bfcc0b1891327
    build.rs                                            f62785b6e61c3862bd701f4f6abe6241
    src
      error.rs                                          c95336aa2dc43b71f8cb8d843e50bc46
      instruction.rs                                    b4f0c05d634717f9a7b8e97a494c0961
      lib.rs                                            d5918a8ce2d6ac9640f671c235b208ce
      processor.rs                                      f8acecc60b8c517d0772f9a5d5cbdc59
      state.rs                                          a152d8def77bc9ffe45785cf79d75e26
runtime
  src
    accounts.rs                                         fce2f79927f41addcd73699e25895456
    bank
      address_lookup_table.rs                           da185e2367c632ceb6b862f263edbdf8
sdk
  program
    src
      message
        versions
          v0
            mod.rs                                      7eb4a97ad27b353d5dc7424a83f90ca0
```

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 onchain program. In other words, there is no way to steal tokens or deny service, ignoring any Solana specific quirks such as account ownership issues. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of Solana's execution model. Some common implementation vulnerabilities include account ownership issues, arithmetic overflows, and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix C.

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 get a comprehensive understanding of the program first. In our audits, we always approach any target in a team of two. 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.

C | Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust as conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Account security

Account Ownership	Account ownership should be properly checked to avoid type confusion attacks. For Anchor, the safety of unchecked accounts should be clearly justified and immediately obvious.
Accounts	For non-Anchor programs, the type of the account should be explicitly validated to avoid type confusion attacks.
Signer Checks	Privileged operations should ensure that the operation is signed by the correct accounts.
PDA Seeds	PDA seeds are uniquely chosen to differentiate between different object classes, avoiding collision.

Input validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so.
Numbers	Sane limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have sane size restrictions to prevent denial of service conditions
Internal State	If there is internal state, ensure that there is explicit validation on the input account's state before engaging in any state transitions. For example, only open accounts should be eligible for closing.

Miscellaneous

Libraries	Out of date libraries should not include any publicly disclosed vulnerabilities
Clippy	cargo clippy is an effective linter to detect potential anti-patterns.

D | Vulnerability Rating Scale

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

Critical	Vulnerabilities which immediately lead to loss of user funds with minimal preconditions Examples: • Misconfigured authority/token account validation • Rounding errors on token transfers
High	Vulnerabilities which could lead to 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 which could lead to denial of service scenarios or degraded usability. Examples: • Malicious input cause computation limit exhaustion • Forced exceptions preventing normal use
Low	Low probability vulnerabilities which could still be 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 • Uncaught Rust errors (vector out of bounds indexing)