Cashmere

Audit

Presented by:

OtterSec contact@osec.io

Robert Chen notdeghost@osec.io

Rajvardhan Agarwal raj@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Cashmere engaged OtterSec to perform an assessment of the `multisig` program.

This assessment was conducted between June 13th and June 17th, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation.

We delivered final confirmation of the patches on **[not yet delivered]**.

Key Findings

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

- 9 findings total
- Two notable findings
  - OS-CSH-ADV-00: Fee collection bypass due to missing check.
  - OS-CSH-SUG-00: Possible account buffer overflow due to realloc.

02 | Scope

The source code was delivered to us in a git repository at github.com/cashmere-inc/multisig. This audit was performed against commit `3b9c29b`.

Only the `multisig` program was in scope.

A brief description of the program follows. A full list of program files and hashes can be found in Appendix A.

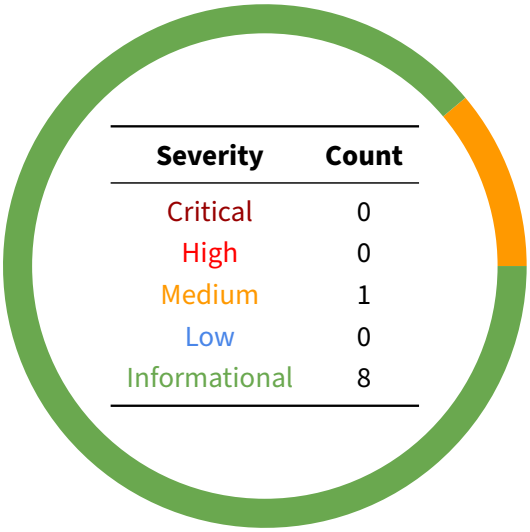
Name	Description
multisig	The central program for managing multi-sig wallets. The functionality of this program includes: • Creating, voting, and executing transactions on behalf of the wallet. • Invoking derived wallet signatures for multiple purposes.

03 | Findings

Overall, we report 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 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	1
Low	0
Informational	8

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 D.

ID	Severity	Status	Description
OS-CSH-ADV-00	Medium	Resolved	Fee collection can be bypassed when creating a multi-sig wallet due to a missing check on the treasury account.

OS-CSH-ADV-00 [med] [resolved] | Wallet Creation Fee Bypass

Description

The `create_multisig` instruction does not properly validate the treasury account. This could allow users to bypass the fee when creating a wallet by passing their own fee account into the instruction.

The following code snippets show the affected code.

multisig/src/lib.rs RUST

```rust
pub struct CreateMultisig<'info> {
    #[account(init, payer = payer, space =
    ↪ Multisig::space(max_owners))]
    multisig: Account<'info, Multisig>,
    #[account(mut)]
    /// CHECK:
    pub treasury: UncheckedAccount<'info>,
```

multisig/src/lib.rs RUST

```rust
anchor_lang::solana_program::program::invoke(
    &anchor_lang::solana_program::system_instruction::transfer(
        &ctx.accounts.payer.key(),
        &ctx.accounts.treasury.key(),
        100000000,
```

Proof of Concept

1. Invoke the `create_multisig` instruction with a self-owned account as the treasury account.
2. The wallet is created. However, the fee for wallet creation is transferred back to the user.

Remediation

The program can include as a constant the `Pubkey` for the treasury account, which can be checked when creating the wallet via an Anchor constraint.

Patch

The fee collection code was moved to the frontend. Updated in #235.

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-CSH-SUG-00	Resolved	Potential buffer overflow when updating the owner set due to unsafe use of realloc.
OS-CSH-SUG-01	Resolved	The size of structures is calculated incorrectly when reserving space.
OS-CSH-SUG-02	Resolved	The discriminator size used during wallet creation is incorrect.
OS-CSH-SUG-03	Resolved	The limit for maximum owners is inconsistent between wallet creation and wallet update.
OS-CSH-SUG-04	Resolved	It is necessary to validate the length of the owners parameter while creating a wallet.
OS-CSH-SUG-05	Resolved	There is the potential for integer overflow when updating the owner sequence number.
OS-CSH-SUG-06	Resolved	Error matching is not required when invoking cross program instructions.
OS-CSH-SUG-07	Resolved	Redundant code is used when re-calculating space for the MultiSig structure.

OS-CSH-SUG-00 [resolved] | Possible Realloc Account Overflow

Description

The `solana-program`'s `realloc` implementation for account data is missing a size check. It fails to check the new size against `MAX_PERMITTED_DATA_INCREASE` before resizing the buffer.

As Anchor allows passing in trailing accounts, this could allow an attacker to overflow into subsequent accounts and modify arbitrary accounts owned by the program.

The following snippets show the affected code.

multisig/src/lib.rs RUST

```
    let new_max_owners = owners.len() + 4;
    let new_size =  4 // Anchor discriminator
    + std::mem::size_of::<Multisig>()
    + 4 // 4 = the Vec discriminator
    + std::mem::size_of::<Pubkey>() * (new_max_owners as usize)
    + 64;  // fudge - extra space for fun

    [...]
    // perform the realloc
    multisig.to_account_info().realloc(new_size, false)?;
```

solana-program-1.9.28/src/account_info.rs RUST

```
pub fn realloc(&self, new_len: usize, zero_init: bool) -> Result<(),
    ↪  ProgramError> {
    let orig_len = self.data_len();
    // realloc
    unsafe {
        // First set new length in the serialized data
        let ptr = self.try_borrow_mut_data()?.as_mut_ptr().offset(-8)
    ↪  as *mut u64;
        *ptr = new_len as u64;

        // Then set the new length in the local slice
        let ptr = &mut *(((self.data.as_ptr() as *const u64).offset(1)
    ↪  as u64) as *mut u64);
        *ptr = new_len as u64;
    }
```

This vulnerability is not currently exploitable due to transaction size limits. It's not possible to encode the additional number of accounts required for overflow to occur.

Furthermore, because the length increases more than `MAX_PERMITTED_DATA_INCREASE`, the Solana runtime will abort the transaction.

solana/programs/bpf_loader/src/serialization.rs RUST

```rust
if post_len != *pre_len
    && (post_len.saturating_sub(*pre_len)) <=
    ↪  MAX_PERMITTED_DATA_INCREASE
{
    data_end = start + post_len;
}
```

However, it is still recommended to patch this vulnerability to avoid future issues. For more information, please refer to the code snippet below.

RUST

```rust
require!(new_size < old_size + MAX_PERMITTED_DATA_INCREASE, ...);
multisig.to_account_info().realloc(new_size, false)?;
```

Patch

Check size increase prior to reallocation, implemented in #256.

multisig/src/lib.rs RUST

```rust
require!(new_size < mutlsig_account_info.data_len() +
    ↪  MAX_PERMITTED_DATA_INCREASE, InvalidReallocAccountSize);
```

OS-CSH-SUG-01 [resolved] | Incorrect Struct Size Calculcations

Description

It was found that `std::mem::size_of` is used for calculating the size of structures. However, `std::mem::size_of` returns the in-memory size of a structure, and has no relation to the size of its Borsh-serialized representation.

multisig/src/state.rs RUST

```
    pub fn space(max_owners: u8) -> usize {
        4 // Anchor discriminator
            + std::mem::size_of::<Multisig>()
            + 4 // 4 = the Vec discriminator
            + std::mem::size_of::<Pubkey>() * (max_owners as usize)
            + 64 // fudge - extra space for fun
```

Moreover, it was noticed that no memory gets reserved for the `signers` vector present in the `Proposal` structure. It can be seen in the code snippet below that space is only reserved for the instructions vector.

multisig/src/state.rs RUST

```
impl Proposal {
    // computes the amount of space to allocate
    pub fn space(ixs: Vec<ProposalIx>) -> usize {
            4 // Anchor discriminator
            + std::mem::size_of::<Proposal>()
            + 4 // 4 = the Vec discriminator
            + (ixs.iter().map(|ix| ix.space()).sum::<usize>())
    }
}
```

Remediation

The size for each structure must be calculated manually and added to the program as several constants. Refer to Anchor documentation for more information. It is also recommended to add tests that ensure that the serialized size is what was expected.

In addition, memory must be reserved for the `signers` vector when the space for the `Proposal` structure is calculated.

Patch

Resolved in #278.

multisig/src/state.rs RUST

```
    pub fn space(&self) -> usize {
        8 // Anchor discriminator
        + 32 // multisig (Pubkey)
        + 4 // ixs (Vec discriminator)
        + (self.ixs.iter().map(|ix| ix.space()).sum::<usize>()) //
↪   ixs (ProposalIx * len) &ProposalIx
        + 4 // signers (Vec discriminator)
        + (self.signers.len() as usize) * 1 // signers (bool *
↪   len)
        + 8 // num_executed_ixs (u64)
        + 4 // owner_set_seqno (u32)
        + 32 // proposer (Pubkey)
    }
```

OS-CSH-SUG-02 [resolved] | Incorrect Discriminator Size

Description

The discriminator size used while creating the `MultiSig` wallet is incorrect [0]. The actual size of Anchor's discriminator is 8 bytes.

multisig/src/state.rs RUST

```
impl Multisig {
    // computes the amount of space to allocate
    pub fn space(max_owners: u8) -> usize {
        4 // Anchor discriminator [0]
            + std::mem::size_of::<Multisig>()
            + 4 // 4 = the Vec discriminator
            + std::mem::size_of::<Pubkey>() * (max_owners as usize)
            + 64 // fudge - extra space for fun [1]
```

This is not a problem in the current state of the program due to the extra space reserved [1]. However, it is recommended to update the discriminator size to avoid any future issues.

Remediation

Update the Anchor discriminator size to 8 bytes.

Patch

Resolved in #278.

OS-CSH-SUG-03 [resolved] | Inconsistency in maximum number of owners

Description

It was noticed that in the `create_multisig` instruction u8 type is used for `max_owners`. This limits the number of maximum owners to 256 during wallet creation.

multisig/src/lib.rs RUST

```
#[instruction(max_owners: u8)]
pub struct CreateMultisig<'info> {
    #[account(init, payer = payer,
        space = Multisig::space(max_owners))]
    multisig: Account<'info, Multisig>,
```

However, no such limit exists in the `set_owners` instruction and any number of owners can be added.

multisig/src/lib.rs RUST

```
    pub fn set_owners(ctx: Context<Auth>, owners: Vec<Pubkey>) ->
    ↪  Result<()>{
        assert_unique_owners(&owners)?;
        require!(!owners.is_empty(), InvalidOwnersLen);

        let multisig = &mut ctx.accounts.multisig;
        if(owners.len() as u64) < multisig.threshold {
            multisig.threshold = owners.len() as u64;
        }

        if owners.len() >= multisig.owners.len() {
            let new_max_owners = owners.len() + 4;
```

Remediation

The `create_multisig` instruction should use u64 type for `max_owners`. Alternatively, a limit for the maximum number of owners should be enforced in the `set_owners` instruction.

Patch

Resolved in #278.

OS-CSH-SUG-04 [resolved] | Missing Owners Length Check

Description

The `create_multisig` instruction does not validate the length of the `owners` parameter. This can cause the program to panic if the length is larger than `max_owners`, which is used to calculate the size for the `MultiSig` structure.

multisig/src/lib.rs RUST

```
    pub fn create_multisig(
        ctx: Context<CreateMultisig>,
        owners: Vec<Pubkey>,
        max_owners: u8,
        threshold: u64,
        nonce: u8,
    ) -> Result<()> {
        assert_unique_owners(&owners)?;
        require!(threshold > 0 && threshold <= owners.len() as u64,
    ↪   InvalidThreshold);
        require!(!owners.is_empty(), InvalidOwnersLen);

        let multisig = &mut ctx.accounts.multisig;
        multisig.owners = owners;
```

Remediation

The length of the `owners` parameter should be checked against `max_owners`.

RUST

```
require!((max_owners as usize) >= owners.len(), ...);
```

Patch

Resolved in #278.

multisig/src/lib.rs RUST

```rust
        assert_unique_owners(&owners)?;
        require!(threshold > 0 && threshold <= owners.len() as u64,
    ↪   InvalidThreshold);
        require!(!owners.is_empty(), InvalidOwnersLen);
        require!(owners.len() <= (max_owners as usize),
    ↪   InvalidMaxOwnersSize);
```

OS-CSH-SUG-05 [resolved] | Sequence Number Integer Overflow

Description

It was found that it is possible for `multisig.owner_set_seqno` to overflow when updated. This could allow users to propose and execute transactions from an owner set that was previously in use.

A sample of the affected code is shown in the snippet below.

multisig/src/lib.rs RUST

```rust
    pub fn cancel_all_proposals(ctx: Context<Auth>) -> Result<()> {
        let multisig = &mut ctx.accounts.multisig;
        // increment the sequence number
        // so that the pending proposals
        // can no longer be executed
        multisig.owner_set_seqno+=1;
        return Ok(());
    }
```

Remediation

Use `checked_add` when updating `multisig.owner_set_seqno`. For more information, refer to the code snippet below.

multisig/src/lib.rs RUST

```rust
    pub fn cancel_all_proposals(ctx: Context<Auth>) -> Result<()> {
        let multisig = &mut ctx.accounts.multisig;
        // increment the sequence number
        // so that the pending proposals
        // can no longer be executed
        multisig.owner_set_seqno =
    ↪   multisig.owner_set_seqno.checked_add(1);
        return Ok(());
    }
```

Patch

Resolved in #278 by using `checked_add`.

OS-CSH-SUG-06 [resolved] | Dead Error Matching Code

Description

The return value for cross program invocations do not need to be checked as `solana_program::program::invoke_signed` will never return an error. In case of an error during the invocation, the whole transaction aborts.

As seen here, the syscall implementation can only return `Ok(SUCCESS)`.

multisig/src/lib.rs RUST

```
    match solana_program::program::invoke_signed(&ix, accounts,
    ↪  signer) {
            Ok(_result) => {
            [...]
                ctx.accounts.proposal.num_executed_ixs += 1;
            }
            Err(err) => {
```

Remediation

The program does not need to do error matching on cross program invocations. For more information, refer to the code snippet below.

RUST

```
    solana_program::program::invoke_signed(&ix, accounts, signer)?;
    ix_at_index.did_execute = true;
    ctx.accounts.proposal.num_executed_ixs += 1;
    Ok(())
```

Patch

Resolved in #278. Removed error matching during CPI.

OS-CSH-SUG-07 [resolved] | Space Calculation Code Redundancy

Description

It was found that the program uses redundant code for calculating space for the `Multisig` structure when updating the owner set. This not only makes the code error prone during future updates, but could also increase the size of the executable.

multisig/src/lib.rs RUST

```
    let new_max_owners = owners.len() + 4;
    let new_size =  4 // Anchor discriminator
    + std::mem::size_of::<Multisig>()
    + 4 // 4 = the Vec discriminator
    + std::mem::size_of::<Pubkey>() * (new_max_owners as usize)
    + 64;  // fudge - extra space for fun
```

multisig/src/state.rs RUST

```
impl Multisig {
    // computes the amount of space to allocate
    pub fn space(max_owners: u8) -> usize {
       4 // Anchor discriminator [0]
           + std::mem::size_of::<Multisig>()
           + 4 // 4 = the Vec discriminator
           + std::mem::size_of::<Pubkey>() * (max_owners as usize)
           + 64 // fudge - extra space for fun [1]
```

Remediation

The program should use `Multisig::space` function when re-calculating the size for the `MultiSig` structure. For more information, please refer to the code snippet below.

multisig/src/lib.rs RUST

```
    let new_max_owners = owners.len() + 4;
    let new_size =  Multisig::space(new_max_owners as u8);
```

Patch

Resolved in #278 by using `Multisig::space`.

A | Program Files

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

```
Cargo.toml                          336ab16e2e2eed67f759db1b4b803b29
Xargo.toml                          815f2dfb6197712a703a8e1f75b03c69
src
  lib.rs                            ea4d12e8ce50d9a2339a51ee67c0fa19
  state.rs                          3096e9137a0788db8c57f4d85daa9510
```

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)