Sec3™

Security Assessment Report

Multi-Gov Solana Program

February 24, 2025

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the Multi-Gov Solana Program smart contracts.

The artifact of the audit was the source code of the following programs, excluding tests, in a private repository.

The initial audit focused on the following versions and revealed 10 issues or questions.

program	type	commit
wormhole-staking-program	Solana	`41e152d787b227e4326e99c98df84dbd14dfeaec`

This report provides a detailed description of the findings and their respective resolutions.

Table of Contents

Result Overview

Issue	Impact	Status
WORMHOLE-STAKING-PROGRAM		
[H-01] Unchecked "delegate_duplication" enables unlimited voting power	High	Resolved
[H-02] Duplicated "StakeAccountMetadata" allows unlimited voting power	High	Resolved
[L-01] "previous_balance" is mistakenly set to "new_balance"	Low	Resolved
[L-02] Inaccurate account size	Low	Resolved
[L-03] Add "close_vesting_balance" instruction	Low	Resolved
[L-04] "window_length" should not exceed "max_checkpoints_account_limit"	Low	Resolved
[L-05] Missing "mut" attribute on "refund_recipient"	Low	Resolved
[I-01] Optimization opportunities in "parse_abi_encoded_message"	Info	Acknowledged
[I-02] Use syscall to get sysvars instead of passing in accounts	Info	Resolved
[Q-01] Max size of "vote_weight_window_length_account"	Question	Resolved

Findings in Detail

WORMHOLE-STAKING-PROGRAM

[H-01] Unchecked "delegate_duplication" enables unlimited voting power

The "`transfer_vesting`" function permits new stake accounts even when "`self.new_vesting_balance.stake_account_metadata`" is uninitialized (equal to "`Pubkey::default`").

If the same account is provided as both the "`delegate_stake_account_checkpoints`" and the "`new_delegate_stake_account_checkpoints`", the "`delegate_duplication`" flag is set to "`true`".

This bypass occurs because no validation is performed on "`new_delegate_stake_account_checkpoints`" when the "`self.new_vesting_balance.stake_account_metadata`" is empty.

```
/* solana/programs/staking/src/contexts/transfer_vesting.rs */
if let (
    Some(_stake_account_metadata),
    Some(_delegate_stake_account_metadata),
    Some(delegate_stake_account_checkpoints),
    Some(_new_stake_account_metadata),
    Some(_new_delegate_stake_account_metadata),
    Some(new_delegate_stake_account_checkpoints),
) = (
    &mut self.stake_account_metadata,
    &mut self.delegate_stake_account_metadata,
    &mut self.delegate_stake_account_checkpoints,
    &mut self.new_stake_account_metadata,
    &mut self.new_delegate_stake_account_metadata,
    &mut self.new_delegate_stake_account_checkpoints,
) {
    delegate_duplication = delegate_stake_account_checkpoints.key()
                           == new_delegate_stake_account_checkpoints.key();
}
```

This would bypass the following reduction of voting power operations:

```
/* solana/programs/staking/src/contexts/transfer_vesting.rs */
if !delegate_duplication {
    delegate_votes_changed = Some(push_checkpoint(
        delegate_stake_account_checkpoints,
        &delegate_checkpoints_account_info,
        self.vest.amount,
        Operation::Subtract,
        current_timestamp,
            &self.vester.to_account_info(),
            &self.system_program.to_account_info(),
        )?);
        let loaded_checkpoints = delegate_stake_account_checkpoints.load()?;
        if loaded_checkpoints.next_index
            >= self.global_config.max_checkpoints_account_limit.into()
        {
            if delegate_stake_account_metadata.key() == stake_account_metadata.key() {
                stake_account_metadata.stake_account_checkpoints_last_index += 1;
            } else {
                delegate_stake_account_metadata.stake_account_checkpoints_last_index += 1;
            }
        }
    }
```

Consequently, the user receiving vesting could call the "`delegate`" instruction to add "`new_vest.amount`" to the voting power, resulting in duplicated voting power.

PoC

- **Prerequisites:** User Y has no "`vest`" account of vesting config C
- **Step 1:** "`vest`" account V of vesting config C is created by the "`vesting_admin`" for user X
- **Step 2:** X transfers his "`vest`" V to Y
  - The six accounts mentioned above are all filled with his delegate's
  - "`delegate_stake_account_checkpoints`" won't be updated as "`delegate_duplication`" will be "`true`"
  - Y's "`new_vesting_balance.stake_account_metadata`" remains empty
- **Step 3:** Y delegates to himself with his "`new_vesting_balance`"

Now X (or his delegate) and Y have a duplicate voting power of "`vest.amount`". More importantly, Y can repeat the above steps to accumulate almost unlimited voting power.

To mitigate this issue, when"`self.new_vesting_balance.stake_account_metadata`" is empty, the program should enforce that all three new stake accounts are explicitly set to "`None`".

Resolution

This issue was fixed by commit “`130d8c8`”.

WORMHOLE-STAKING-PROGRAM

[H-02] Duplicated "StakeAccountMetadata" allows unlimited voting power

In the “`transfer_vesting`” instruction, there are four accounts of the same type.

```
/* solana/programs/staking/src/contexts/transfer_vesting.rs */
#[derive(Accounts)]
pub struct TransferVesting<'info> {
    #[account(mut)]
    pub delegate_stake_account_metadata: Option<Box<Account<'info, StakeAccountMetadata>>>,
    #[account(mut)]
    pub stake_account_metadata: Option<Box<Account<'info, StakeAccountMetadata>>>,
    #[account(mut)]
    pub new_delegate_stake_account_metadata: Option<Box<Account<'info, StakeAccountMetadata>>>,
    #[account(mut)]
    pub new_stake_account_metadata: Option<Box<Account<'info, StakeAccountMetadata>>>,
}
```

All four accounts are declared using the Anchor “`Account`” type. This can result in a situation where, if any two of these accounts refer to the same underlying account, writes to the earlier account are invalidated and overwritten by the latter.

Scenarios of “`delegate_stake_account_metadata == new_delegate_stake_account_metadata`” and “`stake_account_metadata == new_stake_account_metadata`” are handled.

However, the program does not handle cases when “`stake_account_metadata`” is the same as “`new_delegate_stake_account_metadata`”. Specifically, this occurs when User A wishes to transfer vesting to User B, and User B has already delegated their voting rights to User A.

In this case, because the modifications to the “`stake_account_metadata`” are overwritten when “`new_delegate_stake_account_metadata`” is written back, the logic of reducing User A’s “`stake_account_metadata.recorded_vesting_balance`” has no impact. However, the “`new_vest`” account will still be created, and the “`new_stake_account_metadata.recorded_vesting_balance`” of User B will increase as expected.

Since the “`recorded_vesting_balance`” is used to calculate voting weight, an attacker who controls at least two accounts with vesting balances (A and B) could exploit this by constructing the aforementioned delegation relationship to artificially inflate their voting weight.

After completing the attack, B would still receive the new vesting balance, allowing the attacker to reverse the transfer using B and repeat the process, effectively gaining unlimited voting power.

```
/* solana/programs/staking/src/contexts/transfer_vesting.rs */
pub fn transfer_vesting(
) -> Result<TransferVestingEvents> {
    if self.vesting_balance.stake_account_metadata != Pubkey::default() {
        if let (
            Some(stake_account_metadata),
            Some(delegate_stake_account_metadata),
            Some(delegate_stake_account_checkpoints),
        ) = (
            &mut self.stake_account_metadata,
            &mut self.delegate_stake_account_metadata,
            &mut self.delegate_stake_account_checkpoints,
        ) {
            let new_recorded_vesting_balance = stake_account_metadata
                .recorded_vesting_balance
                .checked_sub(self.vest.amount)
                .ok_or(VestingError::Underflow)?;

            let recorded_vesting_balance_changed = stake_account_metadata
                .update_recorded_vesting_balance(new_recorded_vesting_balance); // @audit: if `stake_account_metadata` == `new_delegate_stake_account_metadata`, this write will be invalid.

    if self.new_vesting_balance.stake_account_metadata != Pubkey::default() {
        if let (
            Some(new_stake_account_metadata),
            Some(new_delegate_stake_account_metadata),
            Some(new_delegate_stake_account_checkpoints),
        ) = (
            &mut self.new_stake_account_metadata,
            &mut self.new_delegate_stake_account_metadata,
            &mut self.new_delegate_stake_account_checkpoints,
        ) {
            let new_recorded_vesting_balance = new_stake_account_metadata
                .recorded_vesting_balance
                .checked_add(self.vest.amount)
                .ok_or(VestingError::Overflow)?;

            let recorded_vesting_balance_changed = new_stake_account_metadata
                .update_recorded_vesting_balance(new_recorded_vesting_balance); // @audit: `stake_account_metadata` == `new_delegate_stake_account_metadata` does not affect the write operation to `new_stake_account_metadata`, `new_stake_account_metadata.recorded_vesting_balance` will increase as expected.

/* solana/programs/staking/src/state/stake_account.rs */
pub fn update_recorded_vesting_balance(
) -> RecordedVestingBalanceChanged {
    self.recorded_vesting_balance = new_recorded_vesting_balance;
```

Anchor's Behavior

In Anchor, accounts of the "`Account`" type are deserialized into heap objects before the instruction handler is executed.

If accounts A and B refer to the same underlying account, Anchor deserializes them into two separate heap objects, and modifications are applied independently.

After the handler completes, these heap objects are serialized and written back to the account storage, with the earlier accounts in the account struct being written first. As a result, the modifications to the first account can be overwritten by the second.

Tests

The following test code demonstrates the scenario where two identical accounts are passed to the instruction. It shows that only the writes to "`account2`" take effect:

```rust
/* lib.rs */
#[program]
pub mod anchor_test {
    use super::*;

    pub fn create_dup_account(ctx: Context<CreateDupAccount>) -> Result<()> {
        ctx.accounts.dup_account.value = 100;
        Ok(())
    }

    pub fn edit_dup_account(ctx: Context<EditDupAccount>) -> Result<()> {
        if let (
            Some(account1),
            Some(account2),
        ) = (
            &mut ctx.accounts.dup_account1,
            &mut ctx.accounts.dup_account2,
        ) {
            // @audit: we can see these two accounts have the same key, but are different objects on
            ↪  heap.
            msg!("account1: {:?}, {:p}", account1.key(), *account1);
            msg!("account2: {:?}, {:p}", account2.key(), *account2);
            account1.value = account1.value.checked_sub(20).unwrap();
            account2.value = account2.value.checked_add(20).unwrap();
        }
        Ok(())
    }
}
```

```
#[derive(Accounts)]
pub struct CreateDupAccount<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,

    #[account(
        init,
        payer = payer,
        space = 0x10,
    )]
    pub dup_account: Account<'info, DupAccount>,
    pub system_program: Program<'info, System>,
}

#[derive(Accounts)]
pub struct EditDupAccount<'info> {
    #[account(mut)]
    pub payer: Signer<'info>,

    #[account(mut)]
    pub dup_account1: Option<Box<Account<'info, DupAccount>>>,

    #[account(mut)]
    pub dup_account2: Option<Box<Account<'info, DupAccount>>>,

    pub system_program: Program<'info, System>,
}

#[account]
#[derive(Default, Debug, BorshSchema)]
pub struct DupAccount {
    pub value: u64,
}
```

```
/* test.ts */
it("dup test", async () => {
  await provider.connection.confirmTransaction(
    await provider.connection.requestAirdrop(
      admin.publicKey,
      10000000000
    ),
    "confirmed"
  );

  const dupAccount = anchor.web3.Keypair.generate();

  let tx = await program.methods.createDupAccount().accounts({
    payer: admin.publicKey,
    dupAccount: dupAccount.publicKey
  }).signers([admin, dupAccount]).rpc();
  console.log("Your transaction signature", tx);

  await program.provider.connection.confirmTransaction(tx, "confirmed");
  let txDetails = await program.provider.connection.getTransaction(tx, {
    maxSupportedTransactionVersion: 0,
    commitment: "confirmed",
  });
  let logs = txDetails?.meta?.logMessages || null;
  console.log(logs);

  tx = await program.methods.editDupAccount().accounts({
    payer: admin.publicKey,
    dupAccount1: dupAccount.publicKey,
    dupAccount2: dupAccount.publicKey
  }).signers([admin]).rpc();
  console.log("Your transaction signature", tx);

  await program.provider.connection.confirmTransaction(tx, "confirmed");
  txDetails = await program.provider.connection.getTransaction(tx, {
    maxSupportedTransactionVersion: 0,
    commitment: "confirmed",
  });
  logs = txDetails?.meta?.logMessages || null;
  console.log(logs);

  let dupAccountData = await program.account.dupAccount.fetch(dupAccount.publicKey);
  console.log("dupAccount.value:", dupAccountData.value.toNumber());
});
```

```
[
  'Program EkbZxX3AyVsP958cGcRnRSpy5nuxcPCMRsoJBJmEcrPD invoke [1]',
  'Program log: Instruction: EditDupAccount',
  'Program log: account1: 8VXEiFheEXzQhDgh2Q5EM6Zf4whUDv5trQkSLLjtgh2x, 0x300007e58',
  'Program log: account2: 8VXEiFheEXzQhDgh2Q5EM6Zf4whUDv5trQkSLLjtgh2x, 0x300007e48',
  'Program EkbZxX3AyVsP958cGcRnRSpy5nuxcPCMRsoJBJmEcrPD consumed 26583 of 200000 compute units',
  'Program EkbZxX3AyVsP958cGcRnRSpy5nuxcPCMRsoJBJmEcrPD success'
]
dupAccount.value: 120 // @audit: not the expected 100
```

Resolution

This issue was fixed by commit "`130d8c8`".

Three allowed scenarios are whitelisted for these three potentially equal accounts, where the values may only be "`Some`" when their corresponding account is actively in use; in all other cases, the values must be "`None`".

WORMHOLE-STAKING-PROGRAM

[L-01] "previous_balance" is mistakenly set to "new_balance"

Functions "`update_recorded_balance()`" and "`update_recorded_vesting_balance()`" return "`RecordedBalanceChanged`" and "`RecordedVestingBalanceChanged`", whose "`previous_balance`" is set to "`self.recorded_balance`".

However, the "`self.recorded_balance`" has been updated to "`new_recorded_balance`" just before the assignments, which is not the previous balance.

```
/* solana/programs/staking/src/state/stake_account.rs */
058 | pub fn update_recorded_balance(&mut self, new_recorded_balance: u64) -> RecordedBalanceChanged {
059 |     emit!(RecordedBalanceChanged {
061 |         previous_balance: self.recorded_balance,
062 |         new_balance: new_recorded_balance,
063 |     });
065 |     self.recorded_balance = new_recorded_balance;
067 |     RecordedBalanceChanged {
069 |         previous_balance: self.recorded_balance,
070 |         new_balance: new_recorded_balance,
071 |     }
072 | }
073 |
074 | pub fn update_recorded_vesting_balance(
075 |     &mut self,
076 |     new_recorded_vesting_balance: u64,
077 | ) -> RecordedVestingBalanceChanged {
078 |     emit!(RecordedVestingBalanceChanged {
080 |         previous_balance: self.recorded_vesting_balance,
081 |         new_balance: new_recorded_vesting_balance,
082 |     });
084 |     self.recorded_vesting_balance = new_recorded_vesting_balance;
086 |     RecordedVestingBalanceChanged {
088 |         previous_balance: self.recorded_vesting_balance,
089 |         new_balance: new_recorded_vesting_balance,
090 |     }
```

However, the impact is limited as the return value is only used by "`emit_cpi!`" for event logging.

Resolution

This issue was fixed by commit "`b939c45`".

WORMHOLE-STAKING-PROGRAM
[L-02] Inaccurate account size

Some structs in the program use "`std::mem::size_of`" to calculate account space. However, Anchor uses Borsh serialization, which calculates sizes differently from "`std::mem::size_of`".

Notable differences include:

1. "`Vec`" and "`String`" are variable-length when serialized, while "`std::mem::size_of`" only calculates the fixed length of their metadata.
2. "`std::mem::size_of`" may include padding introduced by the Rustc, while Borsh never includes padding.

Fortunately, none of the current structs using "`std::mem::size_of`" contain "`Vec`" or "`String`" types so the allocated space is sufficient.

However, extra space is allocated because of the paddings, such as the space for the struct "`SpokeMessageExecutor`".

Without considering the discriminator, the "`std::mem::size_of`" returns a length of 70, while the actual Borsh-serialized length is 69.

```
/* solana/programs/staking/src/context.rs */
#[derive(Accounts)]
pub struct InitializeSpokeMessageExecutor<'info> {
    #[account(
        init,
        payer = governance_authority,
        space = SpokeMessageExecutor::LEN,
        seeds = [SPOKE_MESSAGE_EXECUTOR_SEED.as_bytes()],
        bump
    )]
    pub executor: Account<'info, SpokeMessageExecutor>,
}

/* solana/programs/staking/src/state/spoke_message_executor.rs */
#[account]
#[derive(Default, Debug, BorshSchema)]
pub struct SpokeMessageExecutor {
    pub bump: u8,
    // The hub dispatcher address
    pub hub_dispatcher: Pubkey,
    // The hub chain id
```

```rust
    pub hub_chain_id: u16,
    // The spoke chain id
    pub spoke_chain_id: u16,
    // Wormhole contract handling messages
    pub wormhole_core: Pubkey,
}

impl SpokeMessageExecutor {
    pub const LEN: usize =
        SpokeMessageExecutor::DISCRIMINATOR.len() + std::mem::size_of::<SpokeMessageExecutor>();
}

#[cfg(test)]
pub mod tests {
    use super::{MessageReceived, SpokeMessageExecutor};
    use super::*;
    #[test]
    fn check_spoke_message_executor_size() {
        assert!(SpokeMessageExecutor::LEN == 8 + 2 + 32 + 2 + 2 + 32); // 78
    }
    #[test]
    fn check_anchor_serialize() {
        // serialize a spoke message executor and print length
        let spoke_message_executor = SpokeMessageExecutor {
            bump: 0,
            hub_dispatcher: Pubkey::new_unique(),
            hub_chain_id: 1,
            spoke_chain_id: 2,
            wormhole_core: Pubkey::new_unique(),
        };
        let serialized = spoke_message_executor.try_to_vec().unwrap();
        assert_eq!(serialized.len(), 69); // without discriminator
    }
}
```

Consider replacing “`std::mem::size_of`” with Anchor’s “`INIT_SPACE`”.

Resolution

This issue was fixed by commit “`5398758`”.

WORMHOLE-STAKING-PROGRAM

[L-03] Add "close_vesting_balance" instruction

A “`vesting_balance`” account will be created for each “`vester`” under every “`vesting_config`”, whose rents are paid by the “`vesting_admin`”.

```
/* solana/programs/staking/src/contexts/create_vesting_balance.rs */
#[account(
    init,
    payer = admin,
    space = VestingBalance::INIT_SPACE,
    seeds = [VESTING_BALANCE_SEED.as_bytes(), config.key().as_ref(),
             vester_ta.owner.key().as_ref()],
    bump
)]
vesting_balance: Account<'info, VestingBalance>,
```

Consider implementing an instruction to close these unnecessary “`vesting_balance`” accounts. In this way, the “`vesting_admin`” can reclaim the rents and avoid wasting the funds used to create these accounts.

Resolution

This issue was fixed by commit “`ff2c875`”.

WORMHOLE-STAKING-PROGRAM

[L-04] "window_length" should not exceed "max_checkpoints_account_limit"

The "`cast_vote`" instruction scans all checkpoint data within the time window "`(vote_start - window_length, vote_start]`" and takes the minimum value as the "`total_weight`".

This instruction requires passing in a maximum of two checkpoint accounts, with each containing up to "`config.max_checkpoints_account_limit`" entries.

```
/* solana/programs/staking/src/context.rs */
pub struct CastVote<'info> {
    /// CheckpointData account that contains the checkpoint for the timestamp
    /// vote_start - vote_weight_window_length
    #[account(
        mut,
        has_one = owner,
        seeds = [CHECKPOINT_DATA_SEED.as_bytes(), owner.key().as_ref(),
                 stake_account_checkpoints_index.to_le_bytes().as_ref()],
        bump
    )]
    pub voter_checkpoints: AccountLoader<'info, checkpoints::CheckpointData>,
    /// Next CheckpointData account if it exists Necessary for handle the
    /// case when the vote window contains checkpoints stored on two accounts
    pub voter_checkpoints_next: Option<AccountLoader<'info, checkpoints::CheckpointData>>,
```

However, if the "`window_length`" is greater than "`max_checkpoints_account_limit`", the time window to be scanned may span across three or more checkpoint accounts.

In such cases, the program only processes the checkpoint data from the first two accounts, ignoring the data in the subsequent window. This can result in a calculated "`total_weight`" that is higher than the actual value, potentially inflating the user's voting power.

```
/* solana/programs/staking/src/lib.rs */
pub fn cast_vote(
) -> Result<()> {
    let window_start = proposal.vote_start - window_length.value;
    if let Some((window_start_checkpoint_index, window_start_checkpoint)) = find_checkpoint_le(
        &ctx.accounts.voter_checkpoints.to_account_info(),
        window_start,
    )? {
        let mut total_weight = window_start_checkpoint.value;
        let mut checkpoint_index = window_start_checkpoint_index + 1;
        let mut reading_from_next_account = false;
        loop {
            if !reading_from_next_account
```

```
                && (checkpoint_index as u32) == config.max_checkpoints_account_limit
            {
                // Reset checkpoint_index for the next account
                checkpoint_index = 0;
                // Now reading from the next account
                reading_from_next_account = true;
                // Continue to the next iteration to read further checkpoints from the next account
                continue;
            } else {
                // Read from the current or next account based on reading_from_next_account
                let (voter_checkpoints_loader, voter_checkpoints_data) =
                    if reading_from_next_account {
                        let voter_checkpoints_next_loader =
                            ctx.accounts.voter_checkpoints_next.as_ref().unwrap();
                        let voter_checkpoints_next_data =
                            voter_checkpoints_next_loader.load()?;
                        (voter_checkpoints_next_loader, voter_checkpoints_next_data)
                    } else {
                        let voter_checkpoints_loader = &ctx.accounts.voter_checkpoints;
                        let voter_checkpoints_data = voter_checkpoints_loader.load()?;
                        (voter_checkpoints_loader, voter_checkpoints_data)
                    };
                let next_index = voter_checkpoints_data.next_index;
                if checkpoint_index >= next_index as usize {
                    // No more checkpoints in account
                    break;
                } // @audit: if window is across more than 2 accounts,
                  //          loop will end here after processing the second account
                let checkpoint = read_checkpoint_at_index(
                    &voter_checkpoints_loader.to_account_info(),
                    checkpoint_index,
                )?;
                if checkpoint.timestamp > vote_start {
                    // Checkpoint is beyond the vote start time
                    break;
                }
                if checkpoint.value < total_weight {
                    total_weight = checkpoint.value;
                } // @audit: find the minimum value within the window
                checkpoint_index += 1;
            }
        }
```

Since these configurations ("`window_length`" and "`max_checkpoints_account_limit`") are controlled by the "`HubSolanaMessageDispatcher`" contract on EVM and the administrator, and cannot be modified by regular users, the likelihood of this issue occurring is low.

At the same time, since "`window_length`" is restricted to a value less than 850, while "`max_checkpoints_account_limit`" is expected to be a large value, albeit less than $655,000$, the probability of an administrator configuring a "`window_length`" greater than or equal to "`max_checkpoints_account_limit`" is also low under normal circumstances.

Nevertheless, it is recommended to address this issue to prevent potential problems arising from incorrect configurations.

```
/* solana/programs/staking/src/state/vote_weight_window_lengths.rs */
pub fn write_window_length_at_index(
) -> Result<()> {
    require!(
        window_length.value <= VoteWeightWindowLengths::MAX_VOTE_WEIGHT_WINDOW_LENGTH,
                                   // @audit: <= 850
        ErrorCode::ExceedsMaxAllowableVoteWeightWindowLength
    );

/* solana/programs/staking/src/lib.rs */
pub fn init_config(ctx: Context<InitConfig>, global_config: GlobalConfig) -> Result<()> {
    // Make sure the caller can't set the checkpoint account limit too high
    // We don't want to be able to fill up a checkpoint account and cause a DoS
    // Solana accounts are 10MB maximum = 10485760 bytes
    // The checkpoint account contains 8 + 32 + 8 = 48 bytes of fixed data
    // Every checkpoint is 8 + 8 = 16 bytes, so we can fit in (10485760 - 48) / 16 = 655,357
    // checkpoints
    require!(global_config.max_checkpoints_account_limit <= 655_000,
             ErrorCode::InvalidCheckpointAccountLimit);
```

Resolution

This issue was fixed by commit "`0e9cfeb`".

WORMHOLE-STAKING-PROGRAM

[L-05] Missing "mut" attribute on "refund_recipient"

In the “`AddProposal`” and “`CloseSignatures`” instructions, the lamports returned from closing the “`guardian_signatures`” account are refunded to “`refund_recipient`”.

However, this account is not marked as “`mut`”, which causes it to be marked as read-only in the generated IDL. As a result, invoking these instructions using the IDL will fail.

```
/* solana/programs/staking/src/context.rs */
pub struct AddProposal<'info> {
    #[account(mut, has_one = refund_recipient, close = refund_recipient)]
    pub guardian_signatures: Account<'info, GuardianSignatures>,

    /// CHECK: This account is the refund recipient for the above signature_set
    #[account(address = guardian_signatures.refund_recipient)]
    pub refund_recipient: AccountInfo<'info>, // @audit: This account is used to receive the refunded lamports after `guardian_signatures` is closed, should have the `mut` attribute.

/* solana/programs/staking/src/context.rs */
pub struct CloseSignatures<'info> {
    #[account(mut, has_one = refund_recipient, close = refund_recipient)]
    pub guardian_signatures: Account<'info, GuardianSignatures>,

    #[account(address = guardian_signatures.refund_recipient)]
    pub refund_recipient: Signer<'info>, // @audit: same issue
}
```

Resolution

This issue was fixed by commit “`907fbd4`”.

WORMHOLE-STAKING-PROGRAM

[I-01] Optimization opportunities in "parse_abi_encoded_message"

1. Trailing data in parse_abi_encoded_message

The "`parse_abi_encoded_message`" function uses the "`decode`" function from the "`ethabi`" crate, which allows trailing data after decoding:

```
/* solana/programs/staking/src/utils/execute_message.rs */
let tokens = decode(&params, data).map_err(|e| {
    IoError::new(
        ErrorKind::InvalidData,
        format!("Failed to decode ABI data: {}", e),
    )
})?;

/* ~/.cargo/registry/src/index.crates.io-6f17d22bba15001f/ethabi-18.0.0/src/decoder.rs */
/// Decodes ABI compliant vector of bytes into vector of tokens described by types param.
/// Returns ok, even if some data left to decode
pub fn decode(types: &[ParamType], data: &[u8]) -> Result<Vec<Token>, Error> {
    decode_offset(types, data).map(|(tokens, _)| tokens)
}
```

It is recommended to use the "`decode_offset`" function and check whether the returned offset is consistent with the data length.

2. Use try_from instead of as

```
/* solana/programs/staking/src/utils/execute_message.rs */
// Extract wormhole_chain_id
let wormhole_chain_id = tokens
    .get(1)
    .ok_or_else(|| IoError::new(ErrorKind::InvalidData, "Missing wormhole_chain_id"))?
    .clone()
    .into_uint()
    .ok_or_else(|| IoError::new(ErrorKind::InvalidData, "Failed to parse wormhole_chain_id"))?
    .as_u64() as u16;
```

The "`as_u64`" method from the "`primitive-types`" crate has a fitting check, but "`as u16`" does not. To avoid overflow, it is recommended to use "`try_from`" instead of "`as`".

Resolution

The team acknowledged this issue.

WORMHOLE-STAKING-PROGRAM

[I-02] Use syscall to get sysvars instead of passing in accounts

Anchor recommends using syscalls to get the "`clock`" or "`rent`" instead of using user-provided accounts.

```
/* solana/programs/staking/src/context.rs */
039 | pub struct InitConfig<'info> {
054 |     pub rent: Sysvar<'info, Rent>,
056 | }

/* solana/programs/staking/src/context.rs */
484 | pub struct CreateStakeAccount<'info> {
527 |     pub rent: Sysvar<'info, Rent>,
530 | }
```

The two occurrences of the "`rent`" account are in fact not used. Instead, the implementation uses syscalls directly.

Therefore, the account can be removed safely.

Resolution

This issue was fixed by commit "`7af0666`".

WORMHOLE-STAKING-PROGRAM

[Q-01] Max size of "vote_weight_window_length_account"

The program uses a single "`vote_weight_window_length_account`", whose size grows continuously as new "`window_length`" entries are added.

The "`vote_weight_window_length`" account contains a 16-byte fixed data. Each "`window_length`" is 16 bytes. Solana accounts have a maximum size limit of 10 MB $(10,485,760$ bytes), which allows a maximum capacity of $(10,485,760 - 16)/16 = 674,109$ entries.

As a new "`window_length`" can only be pushed by the owner of the "`HubSolanaMessageDispatcher`" contract, is the max account size limit still a concern?

Resolution

The team clarified this is not a concern. The vote weight window will only be changed a handful of times, which will be done through governance proposals and voted on.

Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and Wormhole Foundation (the ''Client'') (the "Client''). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification. We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.

Sec3™