VRust

Security Assessment

O2Lab VRust Team

10/29/2022 21:52:23

Contents

Summary

This report has been prepared for O2Lab VRust Team to discover issues and vulnerabilities in the source code of the O2Lab VRust Team project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Static Analysis and Manual Review techniques. The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

Overview

Project Summary

Project Name	O2Lab VRust Team
Platform	Ethereum
Language	Solana
Crate	sol_payment_processor
GitHub Location	https://github.com/parasol-aser/vrust
sha256	Unknown

Audit Summary

Delivery Date	10/29/2022
Audit Methodology	Static Analysis
Key Components	

Vulnerability Summary

Vulnerability Level	Total
Critical	12
Major	0
Medium	0
Minor	0
Informational	0
Discussion	0

Findings

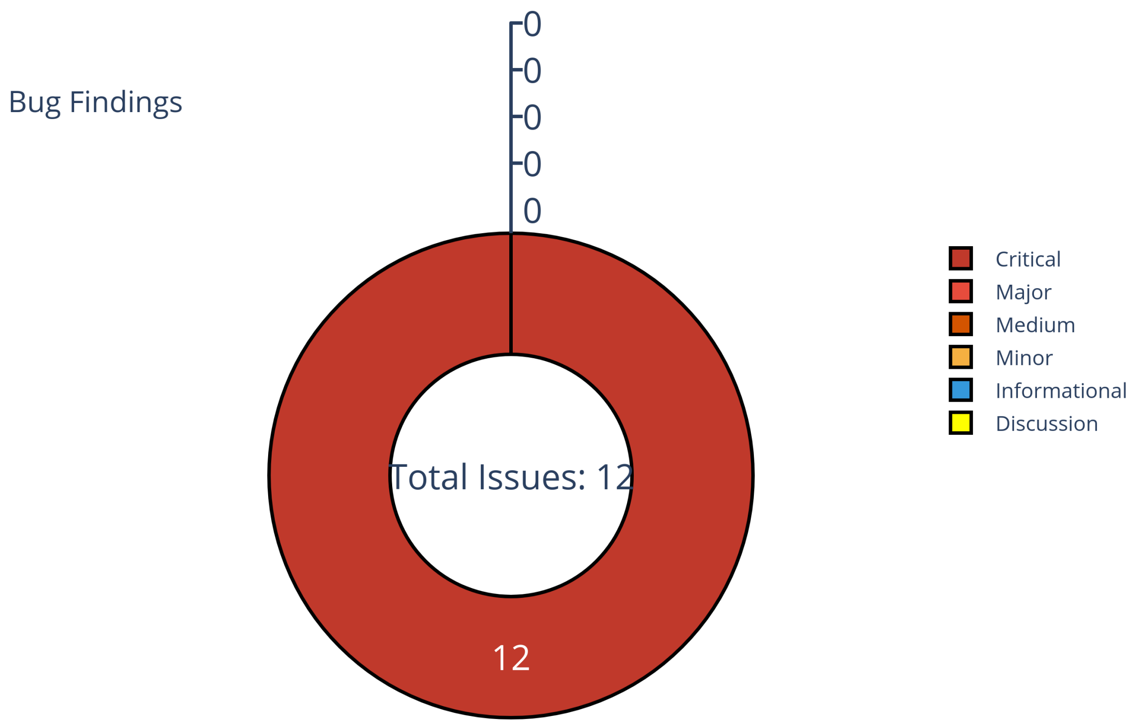

Figure 1: Findings

Finding Statistic

Category	Count
IntegerFlow	5
MissingKeyCheck	2
CrossProgramInvocation	5

ID	Category	Severity	Status
0	IntegerFlow	Critical	UnResolved
1	IntegerFlow	Critical	UnResolved
2	IntegerFlow	Critical	UnResolved
3	IntegerFlow	Critical	UnResolved
4	IntegerFlow	Critical	UnResolved
5	MissingKeyCheck	Critical	UnResolved
6	MissingKeyCheck	Critical	UnResolved
7	CrossProgramInvocation	Critical	UnResolved
8	CrossProgramInvocation	Critical	UnResolved
9	CrossProgramInvocation	Critical	UnResolved
10	CrossProgramInvocation	Critical	UnResolved
11	CrossProgramInvocation	Critical	UnResolved

Issue: 0: IntegerFlow

Category	Severity	Status
IntegerFlow	Critical	UnResolved

- Location

src/engine/cancel_subscription.rs:101:21: 101:67

```
(subscription_account.joined + trial_duration)

```

- Code Context

Vulnerability at Line: 101

```
    let trial_duration: i64 = match package.trial {
        None => 0,
        Some(value) => value,
    };
    // don't allow cancellation if trial period ended
    if timestamp >= (subscription_account.joined + trial_duration) {
        msg!("Info: Subscription amount not refunded because trial period
         ↪  has ended.");
    } else {
        // Transferring payment back to the payer...
        invoke_signed(

```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪  instruction::PaymentProcessorInstruction>::process(){//
    ↪  src/processor.rs:15:5: 61:6 }
        fn engine::cancel_subscription::process_cancel_subscription(){//
         ↪  src/engine/cancel_subscription.rs:24:1: 165:2 }

```

- description:

- link:
- alleviation:

Issue: 1: IntegerFlow

Category	Severity	Status
IntegerFlow	Critical	UnResolved

- Location

src/engine/renew.rs:52:27: 52:60

```
(quantity as u64) * package.price

```

- Code Context

Vulnerability at Line: 52

```
        order_info,
        subscription_info,
        &subscription_account.name,
    )?;
    // ensure the amount paid is as expected
    let expected_amount = (quantity as u64) * package.price;
    if expected_amount > order_account.paid_amount {
        return Err(PaymentProcessorError::NotFullyPaid.into());
    }
    // update subscription account

```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪   instruction::PaymentProcessorInstruction>::process(){//
    ↪   src/processor.rs:15:5: 61:6 }
        fn engine::renew::process_renew_subscription(){//
        ↪   src/engine/renew.rs:14:1: 74:2 }

```

- description:

- link:
- alleviation:

Issue: 2: IntegerFlow

Category	Severity	Status
IntegerFlow	Critical	UnResolved

- Location

src/engine/subscribe.rs:115:21: 115:47

```
timestamp + trial_duration

```

- Code Context

Vulnerability at Line: 115

```
        owner: signer_info.key.to_bytes(),
        merchant: merchant_info.key.to_bytes(),
        name,
        joined: timestamp,
        period_start: timestamp,
        period_end: timestamp + trial_duration + package.duration,
        data,
    };
    subscription.pack(&mut subscription_data);


```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪  instruction::PaymentProcessorInstruction>::process(){//
    ↪  src/processor.rs:15:5: 61:6 }
        fn engine::subscribe::process_subscribe(){//
        ↪  src/engine/subscribe.rs:16:1: 126:2 }

```

- description:

- link:
- alleviation:

Issue: 3: IntegerFlow

Category	Severity	Status
IntegerFlow	Critical	UnResolved

- Location

src/engine/withdraw.rs:123:24: 123:70

```
(subscription_account.joined + trial_duration)

```

- Code Context

Vulnerability at Line: 123

```
        let trial_duration: i64 = match package.trial {
            None => 0,
            Some(value) => value,
        };
        // don't allow withdrawal if still within trial period
        if timestamp < (subscription_account.joined + trial_duration) {
            return
            ↪  Err(PaymentProcessorError::CantWithdrawDuringTrial.into());
        }
    }
    // Transferring payment to the merchant...

```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪  instruction::PaymentProcessorInstruction>::process(){//
    ↪  src/processor.rs:15:5: 61:6 }
        fn engine::withdraw::process_withdraw_payment(){//
        ↪  src/engine/withdraw.rs:23:1: 186:2 }

```

- description:

- link:
- alleviation:

Issue: 4: IntegerFlow

Category	Severity	Status
IntegerFlow	Critical	UnResolved

- Location

src/utils.rs:12:41: 12:74

```
(amount as u128 * fee_percentage)

```

- Code Context

Vulnerability at Line: 12

```
pub fn get_amounts(amount: u64, fee_percentage: u128) -> (u64, u64) {
    let mut fee_amount: u64 = 0;
    let mut take_home_amount: u64 = amount;

    if amount >= 100 {
        let possible_fee_amount: u128 = (amount as u128 * fee_percentage) /
        ↪  1000;
        fee_amount = 1;
        if possible_fee_amount > 0 {
            fee_amount = possible_fee_amount as u64;
        }

```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪  instruction::PaymentProcessorInstruction>::process(){//
    ↪  src/processor.rs:15:5: 61:6 }
        fn engine::pay::process_chain_checkout(){//
        ↪  src/engine/pay.rs:351:1: 368:2 }
            fn engine::pay::process_order(){// src/engine/pay.rs:137:1:
            ↪  329:2 }
```

```
                    fn utils::get_amounts(){// src/utils.rs:7:1: 21:2 }

```

- description:
- link:
- alleviation:

Issue: 5: MissingKeyCheck

Category	Severity	Status
MissingKeyCheck	Critical	UnResolved

- Location

/home/yifei/.cargo/registry/src/github.com-1ecc6299db9ec823/solana-program-1.7.1/src/account_info.rs:70:11: 70:33

```
self.lamports.borrow()

```

- Code Context

Vulnerability at Line: 70

```
pub fn lamports(&self) -> u64 {
        **self.lamports.borrow()
    }

```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪   instruction::PaymentProcessorInstruction>::process(){//
    ↪   src/processor.rs:15:5: 61:6 }
        fn engine::cancel_subscription::process_cancel_subscription(){//
        ↪   src/engine/cancel_subscription.rs:24:1: 165:2 }
            fn
            ↪   solana_program::account_info::AccountInfo::<'a>::lamports(){//
            ↪   /home/yifei/.cargo/registry/src/github.com-
            ↪   1ecc6299db9ec823/solana-program-
            ↪   1.7.1/src/account_info.rs:69:5: 71:6
            ↪   }

```

- description:

- link:
- alliviation:

Issue: 6: MissingKeyCheck

Category	Severity	Status
MissingKeyCheck	Critical	UnResolved

- Location

src/engine/cancel_subscription.rs:152:49: 152:77

```
order_info.data.borrow_mut()

```

- Code Context

Vulnerability at Line: 152

```
147              order_info.lamports(),
148          )?;
149          // Updating order account information...
150          order_account.status = OrderStatus::Cancelled as u8;
151          order_account.modified = timestamp;
152          OrderAccount::pack(&order_account, &mut
           ↪   order_info.data.borrow_mut());
153          // set period end to right now
154          subscription_account.period_end = timestamp;
155      }
156
157
```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪   instruction::PaymentProcessorInstruction>::process(){//
    ↪   src/processor.rs:15:5: 61:6 }
        fn engine::cancel_subscription::process_cancel_subscription(){//
        ↪   src/engine/cancel_subscription.rs:24:1: 165:2 }

```

- description:

- link:
- alleviation:

Issue: 7: CrossProgramInvocation

Category	Severity	Status
CrossProgramInvocation	Critical	UnResolved

- Location

```
src/engine/cancel_subscription.rs
```

- Code Context

```rust
pub fn process_cancel_subscription(program_id: &Pubkey, accounts:
    &[AccountInfo]) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();

    let signer_info = next_account_info(account_info_iter)?;
    let subscription_info = next_account_info(account_info_iter)?;
    let merchant_info = next_account_info(account_info_iter)?;
    let order_info = next_account_info(account_info_iter)?;
    let order_token_info = next_account_info(account_info_iter)?;
    let refund_token_info = next_account_info(account_info_iter)?;
    let account_to_receive_sol_refund_info =
        next_account_info(account_info_iter)?;
    let pda_info = next_account_info(account_info_iter)?;
    let token_program_info = next_account_info(account_info_iter)?;

    let timestamp = Clock::get()?.unix_timestamp;

    // ensure signer can sign
    if !signer_info.is_signer {
        return Err(ProgramError::MissingRequiredSignature);
    }
    // ensure subscription account is owned by this program
    if *subscription_info.owner != *program_id {
        msg!("Error: Wrong owner for subscription account");
        return Err(ProgramError::IncorrectProgramId);
    }
    // ensure token accounts are owned by token program
    if *order_token_info.owner != spl_token::id() {
        msg!("Error: Order token account must be owned by token program");
        return Err(ProgramError::IncorrectProgramId);
    }
    if *refund_token_info.owner != spl_token::id() {
        msg!("Error: Refund token account must be owned by token program");
        return Err(ProgramError::IncorrectProgramId);
    }
    // check that provided pda is correct
    let (pda, pda_nonce) = Pubkey::find_program_address(&[PDA_SEED],
    ↪   &program_id);
    if pda_info.key != &pda {
        return Err(ProgramError::InvalidSeeds);
    }

    // get the subscription account
    let mut subscription_account =
    ↪   SubscriptionAccount::unpack(&subscription_info.data.borrow())?;
    if !subscription_account.is_initialized() {
        return Err(ProgramError::UninitializedAccount);
    }
    if subscription_account.is_closed() {
        return Err(PaymentProcessorError::ClosedAccount.into());
    }
    if subscription_account.discriminator != Discriminator::Subscription as
    ↪   u8 {
        msg!("Error: Invalid subscription account");
        return Err(ProgramError::InvalidAccountData);
    }
    let (mut order_account, package) = subscribe_checks(
        program_id,
        signer_info,
        merchant_info,
        order_info,
        subscription_info,
        &subscription_account.name,
    )?;

    // ensure the order payment token account is the right one
    if order_token_info.key.to_bytes() != order_account.token {
        msg!("Error: Incorrect order token account");
        return Err(ProgramError::InvalidAccountData);
    }
    // ensure the signer is the order payer
    if signer_info.key.to_bytes() != order_account.payer {
        msg!("Error: One can only cancel their own subscription payment");
        return Err(ProgramError::InvalidAccountData);
    }

    // get the trial period duration
    let trial_duration: i64 = match package.trial {
        None => 0,
        Some(value) => value,
    };
    // don't allow cancellation if trial period ended
    if timestamp >= (subscription_account.joined + trial_duration) {
        msg!("Info: Subscription amount not refunded because trial period
         ↪  has ended.");
    } else {
        // Transferring payment back to the payer...
        invoke_signed(
            &spl_token::instruction::transfer(
                token_program_info.key,
                order_token_info.key,
                refund_token_info.key,
                &pda,
                &[&pda],
                order_account.paid_amount,
            )
            .unwrap(),
            &[
                token_program_info.clone(),
                pda_info.clone(),
                order_token_info.clone(),
                refund_token_info.clone(),
            ],
            &[&[&PDA_SEED, &[pda_nonce]]],
        )?;
        // Close the order token account since it will never be needed
         ↪  again
        invoke_signed(
            &spl_token::instruction::close_account(
                token_program_info.key,
                order_token_info.key,
                account_to_receive_sol_refund_info.key,
                &pda,
                &[&pda],
            )
            .unwrap(),
            &[
                token_program_info.clone(),
                order_token_info.clone(),
                account_to_receive_sol_refund_info.clone(),
                pda_info.clone(),
            ],
            &[&[&PDA_SEED, &[pda_nonce]]],
        )?;
        // mark order account as closed
        order_account.discriminator = Discriminator::Closed as u8;
        // Transfer all the sol from the order account to the
         ↪  sol_destination.
        transfer_sol(
            order_info.clone(),
            account_to_receive_sol_refund_info.clone(),
            order_info.lamports(),
        )?;
        // Updating order account information...
        order_account.status = OrderStatus::Cancelled as u8;
        order_account.modified = timestamp;
        OrderAccount::pack(&order_account, &mut
         ↪  order_info.data.borrow_mut());
        // set period end to right now
        subscription_account.period_end = timestamp;
    }

    // Updating subscription account information...
    subscription_account.status = SubscriptionStatus::Cancelled as u8;
    SubscriptionAccount::pack(
        &subscription_account,
        &mut subscription_info.data.borrow_mut(),
    );

    Ok(())
}

```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪   instruction::PaymentProcessorInstruction>::process(){//
    ↪   src/processor.rs:15:5: 61:6 }
        fn engine::cancel_subscription::process_cancel_subscription(){//
        ↪   src/engine/cancel_subscription.rs:24:1: 165:2 }

```

- description:
- link:
- alleviation:

Issue: 8: CrossProgramInvocation

Category	Severity	Status
CrossProgramInvocation	Critical	UnResolved

- Location

```
src/engine/withdraw.rs
```

- Code Context

```rust
pub fn process_withdraw_payment(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    close_order_account: bool,
) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    let signer_info = next_account_info(account_info_iter)?;
    let order_info = next_account_info(account_info_iter)?;
    let merchant_info = next_account_info(account_info_iter)?;
    let order_payment_token_info = next_account_info(account_info_iter)?;
    let merchant_token_info = next_account_info(account_info_iter)?;
    let account_to_receive_sol_refund_info =
    ↪   next_account_info(account_info_iter)?;
    let pda_info = next_account_info(account_info_iter)?;
    let token_program_info = next_account_info(account_info_iter)?;

    let timestamp = Clock::get()?.unix_timestamp;

    // ensure signer can sign
    if !signer_info.is_signer {
        return Err(ProgramError::MissingRequiredSignature);
    }
    // ensure merchant and order accounts are owned by this program
    if *merchant_info.owner != *program_id {
        msg!("Error: Wrong owner for merchant account");
        return Err(ProgramError::IncorrectProgramId);
    }
    if *order_info.owner != *program_id {
        msg!("Error: Wrong owner for order account");
        return Err(ProgramError::IncorrectProgramId);
    }
    // ensure buyer token account is owned by token program
    if *merchant_token_info.owner != spl_token::id() {
        msg!("Error: Token account must be owned by token program");
        return Err(ProgramError::IncorrectProgramId);
    }
    // check that provided pda is correct
    let (pda, pda_nonce) = Pubkey::find_program_address(&[PDA_SEED],
    ↪  &program_id);
    if pda_info.key != &pda {
        return Err(ProgramError::InvalidSeeds);
    }
    // get the merchant account
    let merchant_account =
    ↪  MerchantAccount::unpack(&merchant_info.data.borrow())?;
    if merchant_account.is_closed() {
        return Err(PaymentProcessorError::ClosedAccount.into());
    }
    if !merchant_account.is_initialized() {
        return Err(ProgramError::UninitializedAccount);
    }
    // ensure that the token account that we will withdraw to is owned by
    ↪  this
    // merchant.  This ensures that anyone can call the withdraw
    ↪  instruction
    // and the money will still go to the right place
    let merchant_token_data =
    ↪  TokenAccount::unpack(&merchant_token_info.data.borrow())?;
    if merchant_token_data.owner !=
    ↪  Pubkey::new_from_array(merchant_account.owner) {
        return Err(PaymentProcessorError::WrongMerchant.into());
    }
    // get the order account
    let mut order_account =
    ↪  OrderAccount::unpack(&order_info.data.borrow())?;
    if order_account.is_closed() {
        return Err(PaymentProcessorError::ClosedAccount.into());
    }
    if !order_account.is_initialized() {
```

```rust
        return Err(ProgramError::UninitializedAccount);
    }
    // ensure order belongs to this merchant
    if merchant_info.key.to_bytes() != order_account.merchant {
        return Err(ProgramError::InvalidAccountData);
    }
    // ensure the order payment token account is the right one
    if order_payment_token_info.key.to_bytes() != order_account.token {
        return Err(ProgramError::InvalidAccountData);
    }
    // ensure order is not already paid out
    if order_account.status != OrderStatus::Paid as u8 {
        return Err(PaymentProcessorError::AlreadyWithdrawn.into());
    }
    // check if this is for a subscription payment that has a trial period
    if merchant_account.discriminator == Discriminator::MerchantSubscriptionWithTrial as u8 {
        let subscription_info = next_account_info(account_info_iter)?;
        // ensure subscription account is owned by this program
        if *subscription_info.owner != *program_id {
            msg!("Error: Wrong owner for subscription account");
            return Err(ProgramError::IncorrectProgramId);
        }
        // ensure this order is for this subscription
        verify_subscription_order(subscription_info, &order_account)?;
        // get the subscription account
        let subscription_account = SubscriptionAccount::unpack(&subscription_info.data.borrow())?;
        if subscription_account.is_closed() {
            return Err(PaymentProcessorError::ClosedAccount.into());
        }
        if !subscription_account.is_initialized() {
            return Err(ProgramError::UninitializedAccount);
        }
        let package = get_subscription_package(&subscription_account.name, &merchant_account)?;
        // get the trial period duration
        let trial_duration: i64 = match package.trial {
            None => 0,
            Some(value) => value,
        };
        // don't allow withdrawal if still within trial period
        if timestamp < (subscription_account.joined + trial_duration) {
            return
            ↪  Err(PaymentProcessorError::CantWithdrawDuringTrial.into());
        }
    }
    // Transferring payment to the merchant...
    invoke_signed(
        &spl_token::instruction::transfer(
            token_program_info.key,
            order_payment_token_info.key,
            merchant_token_info.key,
            &pda,
            &[&pda],
            order_account.paid_amount,
        )
        .unwrap(),
        &[
            token_program_info.clone(),
            order_payment_token_info.clone(),
            merchant_token_info.clone(),
            pda_info.clone(),
        ],
        &[&[&PDA_SEED, &[pda_nonce]]],
    )?;
    // Close the order token account since it will never be needed again
    invoke_signed(
        &spl_token::instruction::close_account(
            token_program_info.key,
            order_payment_token_info.key,
            account_to_receive_sol_refund_info.key,
            &pda,
            &[&pda],
        )
        .unwrap(),
        &[
            token_program_info.clone(),
            order_payment_token_info.clone(),
            account_to_receive_sol_refund_info.clone(),
            pda_info.clone(),
        ],
        &[&[&PDA_SEED, &[pda_nonce]]],
    )?;

    if close_order_account {
        if merchant_account.owner != signer_info.key.to_bytes() {
            msg!("Error: Only merchant account owner can close order
             ↪  account");
            return Err(ProgramError::MissingRequiredSignature);
        }
        // mark account as closed
        order_account.discriminator = Discriminator::Closed as u8;
        // Transfer all the sol from the order account to the
         ↪  sol_destination.
        transfer_sol(
            order_info.clone(),
            account_to_receive_sol_refund_info.clone(),
            order_info.lamports(),
        )?;
    }

    // Updating order account information...
    order_account.status = OrderStatus::Withdrawn as u8;
    order_account.modified = timestamp;
    OrderAccount::pack(&order_account, &mut order_info.data.borrow_mut());

    Ok(())
}

```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪  instruction::PaymentProcessorInstruction>::process(){//
    ↪  src/processor.rs:15:5: 61:6 }
        fn engine::withdraw::process_withdraw_payment(){//
         ↪  src/engine/withdraw.rs:23:1: 186:2 }

```

- description:
- link:
- alleviation:

Issue: 9: CrossProgramInvocation

Category	Severity	Status
CrossProgramInvocation	Critical	UnResolved

- Location

```
src/engine/common.rs
```

- Code Context

```rust
pub fn create_program_owned_associated_token_account(
    program_id: &Pubkey,
    accounts: &[AccountInfo; 8],
    rent: &Rent,
) -> ProgramResult {
    let signer_info = &accounts[0];
    let base_account_info = &accounts[1];
    let new_account_info = &accounts[2];
    let mint_info = &accounts[3];
    let pda_info = &accounts[4];
    let token_program_info = &accounts[5];
    let system_program_info = &accounts[6];
    let rent_sysvar_info = &accounts[7];

    let (associated_token_address, bump_seed) =
     ↪  Pubkey::find_program_address(
        &[
            &base_account_info.key.to_bytes(),
            &spl_token::id().to_bytes(),
            &mint_info.key.to_bytes(),
        ],
        program_id,
    );
    // assert that the derived address matches the one supplied
    if associated_token_address != *new_account_info.key {
        msg!("Error: Associated address does not match seed derivation");
        return Err(ProgramError::InvalidSeeds);
    }
    // get signer seeds
    let associated_token_account_signer_seeds: &[&[_]] = &[
        &base_account_info.key.to_bytes(),
        &spl_token::id().to_bytes(),
        &mint_info.key.to_bytes(),
        &[bump_seed],
    ];
    // Fund the associated seller token account with the minimum balance to
     ↪  be rent exempt
    let required_lamports = rent
        .minimum_balance(spl_token::state::Account::LEN)
        .max(1)
        .saturating_sub(new_account_info.lamports());
    if required_lamports > 0 {
        // Transfer lamports to the associated seller token account
        invoke(
            &system_instruction::transfer(
                &signer_info.key,
                new_account_info.key,
                required_lamports,
            ),
            &[
                signer_info.clone(),
                new_account_info.clone(),
                system_program_info.clone(),
            ],
        )?;
    }
    // Allocate space for the associated seller token account
    invoke_signed(
        &system_instruction::allocate(new_account_info.key,
↪   spl_token::state::Account::LEN as u64),
        &[new_account_info.clone(), system_program_info.clone()],
        &[&associated_token_account_signer_seeds],
    )?;
    // Assign the associated seller token account to the SPL Token program
    invoke_signed(
        &system_instruction::assign(new_account_info.key,
↪   &spl_token::id()),
        &[new_account_info.clone(), system_program_info.clone()],
        &[&associated_token_account_signer_seeds],
    )?;
    // Initialize the associated seller token account
    invoke(
        &spl_token::instruction::initialize_account(
            &spl_token::id(),
            new_account_info.key,
            mint_info.key,
            pda_info.key,
        )?,
        &[
            new_account_info.clone(),
            mint_info.clone(),
            pda_info.clone(),
            rent_sysvar_info.clone(),
            token_program_info.clone(),
        ],
    )?;

    Ok(())
}

```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪   instruction::PaymentProcessorInstruction>::process(){//
    ↪   src/processor.rs:15:5: 61:6 }
        fn engine::pay::process_chain_checkout(){//
        ↪   src/engine/pay.rs:351:1: 368:2 }
            fn engine::pay::process_order(){// src/engine/pay.rs:137:1:
            ↪   329:2 }
                    fn en-
                    ↪   gine::common::create_program_owned_associated_token_account(){//
                    ↪   src/engine/common.rs:136:1: 220:2 }

```

- description:
- link:
- alleviation:

Issue: 10: CrossProgramInvocation

Category	Severity	Status
CrossProgramInvocation	Critical	UnResolved

- Location

```
src/engine/pay.rs
```

- Code Context

```
pub fn process_order(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    amount: u64,
    order_id: String,
    secret: String,
    maybe_data: Option<String>,
    checkout_items: Option<OrderItems>,
) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();

    let signer_info = next_account_info(account_info_iter)?;
    let order_info = next_account_info(account_info_iter)?;
    let merchant_info = next_account_info(account_info_iter)?;
    let seller_token_info = next_account_info(account_info_iter)?;
    let buyer_token_info = next_account_info(account_info_iter)?;
    let program_owner_info = next_account_info(account_info_iter)?;
    let sponsor_info = next_account_info(account_info_iter)?;
    let mint_info = next_account_info(account_info_iter)?;
    let pda_info = next_account_info(account_info_iter)?;
    let token_program_info = next_account_info(account_info_iter)?;
    let system_program_info = next_account_info(account_info_iter)?;
    let rent_sysvar_info = next_account_info(account_info_iter)?;

    let rent = &Rent::from_account_info(rent_sysvar_info)?;
    let timestamp = Clock::get()?.unix_timestamp;

```

```rust
    let merchant_account = order_checks(
        program_id,
        signer_info,
        merchant_info,
        buyer_token_info,
        mint_info,
        program_owner_info,
        sponsor_info,
    )?;

    // get data
    let mut data = match maybe_data {
        None => String::from(DEFAULT_DATA),
        Some(value) => value,
    };

    let mut order_account_type = Discriminator::OrderExpressCheckout as u8;

    // process chain checkout
    if checkout_items.is_some() {
        order_account_type = Discriminator::OrderChainCheckout as u8;
        let order_items = checkout_items.unwrap();
        chain_checkout_checks(&merchant_account, &mint_info.clone(),
↪   &order_items, amount)?;
        if data == String::from(DEFAULT_DATA) {
            data = json!({ PAID: order_items }).to_string();
        } else {
            // let possible_json_data: Result<BTreeMap<&str, Value>,
             ↪  JSONError> = serde_json::from_str(&data);
            // let json_data = match possible_json_data {
            let json_data: Value = match serde_json::from_str(&data) {
                Err(_error) => return
                 ↪  Err(PaymentProcessorError::InvalidOrderData.into()),
                Ok(data) => data,
            };
            data = json!({
                INITIAL: json_data,
                PAID: order_items
            })
            .to_string();
        }
    }

    // create order account
    let order_account_size = get_order_account_size(&order_id, &secret,
    ↪  &data);
    // the order account amount includes the fee in SOL
    let order_account_amount =
    ↪  Rent::default().minimum_balance(order_account_size);
    invoke(
        &system_instruction::create_account(
            signer_info.key,
            order_info.key,
            order_account_amount,
            order_account_size as u64,
            program_id,
        ),
        &[
            signer_info.clone(),
            order_info.clone(),
            system_program_info.clone(),
        ],
    )?;

    // next we are going to try and create a token account owned by the
    ↪  program
    // but whose address is derived from the order account
    // TODO: for subscriptions, should this use the subscription account as
    ↪  the base?
    create_program_owned_associated_token_account(
        program_id,
        &[
            signer_info.clone(),
            order_info.clone(),
            seller_token_info.clone(),
            mint_info.clone(),
            pda_info.clone(),
            token_program_info.clone(),
            system_program_info.clone(),
            rent_sysvar_info.clone(),
        ],
        rent,
    )?;

    // Transfer payment amount to associated seller token account...
    invoke(
        &spl_token::instruction::transfer(
            token_program_info.key,
            buyer_token_info.key,
            seller_token_info.key,
            signer_info.key,
            &[&signer_info.key],
            amount,
        )
        .unwrap(),
        &[
            buyer_token_info.clone(),
            seller_token_info.clone(),
            signer_info.clone(),
            token_program_info.clone(),
        ],
    )?;

    if Pubkey::new_from_array(merchant_account.sponsor) ==
    ↪  Pubkey::from_str(PROGRAM_OWNER).unwrap()
    {
        // Transferring processing fee to the program owner...
        invoke(
            &system_instruction::transfer(
                &signer_info.key,
                program_owner_info.key,
                merchant_account.fee,
            ),
            &[
                signer_info.clone(),
                program_owner_info.clone(),
                system_program_info.clone(),
            ],
        )?;
    } else {
        // we need to pay both the program owner and the sponsor
        let (program_owner_fee, sponsor_fee) =
        ↪  get_amounts(merchant_account.fee, SPONSOR_FEE);
        // Transferring processing fee to the program owner and sponsor...
        invoke(
            &system_instruction::transfer(
                &signer_info.key,
                program_owner_info.key,
                program_owner_fee,
            ),
            &[
                signer_info.clone(),
                program_owner_info.clone(),
                system_program_info.clone(),
            ],
        )?;
        invoke(
            &system_instruction::transfer(&signer_info.key,
↪   sponsor_info.key, sponsor_fee),
            &[
                signer_info.clone(),
                sponsor_info.clone(),
                system_program_info.clone(),
            ],
        )?;
    }

    // get the order account
    // TODO: ensure this account is not already initialized
    let mut order_account_data = order_info.try_borrow_mut_data()?;
    // Saving order information...
    let order = OrderAccount {
        discriminator: order_account_type,
        status: OrderStatus::Paid as u8,
        created: timestamp,
        modified: timestamp,
        merchant: merchant_info.key.to_bytes(),
        mint: mint_info.key.to_bytes(),
        token: seller_token_info.key.to_bytes(),
        payer: signer_info.key.to_bytes(),
        expected_amount: amount,
        paid_amount: amount,
        order_id,
        secret,
        data,
    };

    order.pack(&mut order_account_data);
```

```rust

    // ensure order account is rent exempt
    if !rent.is_exempt(order_info.lamports(), order_account_size) {
        return Err(ProgramError::AccountNotRentExempt);
    }

    Ok(())
}

```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪  instruction::PaymentProcessorInstruction>::process(){//
    ↪  src/processor.rs:15:5: 61:6 }
        fn engine::pay::process_chain_checkout(){//
        ↪  src/engine/pay.rs:351:1: 368:2 }
            fn engine::pay::process_order(){// src/engine/pay.rs:137:1:
            ↪  329:2 }

```

- description:
- link:
- alleviation:

Issue: 11: CrossProgramInvocation

Category	Severity	Status
CrossProgramInvocation	Critical	UnResolved

- Location

```
src/engine/pay.rs
```

- Code Context

```rust
pub fn process_order(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    amount: u64,
    order_id: String,
    secret: String,
    maybe_data: Option<String>,
    checkout_items: Option<OrderItems>,
) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();

    let signer_info = next_account_info(account_info_iter)?;
    let order_info = next_account_info(account_info_iter)?;
    let merchant_info = next_account_info(account_info_iter)?;
    let seller_token_info = next_account_info(account_info_iter)?;
    let buyer_token_info = next_account_info(account_info_iter)?;
    let program_owner_info = next_account_info(account_info_iter)?;
    let sponsor_info = next_account_info(account_info_iter)?;
    let mint_info = next_account_info(account_info_iter)?;
    let pda_info = next_account_info(account_info_iter)?;
    let token_program_info = next_account_info(account_info_iter)?;
    let system_program_info = next_account_info(account_info_iter)?;
    let rent_sysvar_info = next_account_info(account_info_iter)?;

    let rent = &Rent::from_account_info(rent_sysvar_info)?;
    let timestamp = Clock::get()?.unix_timestamp;

    let merchant_account = order_checks(
        program_id,
        signer_info,
        merchant_info,
        buyer_token_info,
        mint_info,
        program_owner_info,
        sponsor_info,
    )?;

    // get data
    let mut data = match maybe_data {
        None => String::from(DEFAULT_DATA),
        Some(value) => value,
    };

    let mut order_account_type = Discriminator::OrderExpressCheckout as u8;

    // process chain checkout
    if checkout_items.is_some() {
        order_account_type = Discriminator::OrderChainCheckout as u8;
        let order_items = checkout_items.unwrap();
        chain_checkout_checks(&merchant_account, &mint_info.clone(),
↪   &order_items, amount)?;
        if data == String::from(DEFAULT_DATA) {
            data = json!({ PAID: order_items }).to_string();
        } else {
            // let possible_json_data: Result<BTreeMap<&str, Value>,
            ↪   JSONError> = serde_json::from_str(&data);
            // let json_data = match possible_json_data {
            let json_data: Value = match serde_json::from_str(&data) {
                Err(_error) => return
                ↪   Err(PaymentProcessorError::InvalidOrderData.into()),
                Ok(data) => data,
            };
            data = json!({
                INITIAL: json_data,
                PAID: order_items
            })
            .to_string();
        }
    }
```

```rust

    // create order account
    let order_account_size = get_order_account_size(&order_id, &secret,
    ↪   &data);
    // the order account amount includes the fee in SOL
    let order_account_amount =
    ↪   Rent::default().minimum_balance(order_account_size);
    invoke(
        &system_instruction::create_account(
            signer_info.key,
            order_info.key,
            order_account_amount,
            order_account_size as u64,
            program_id,
        ),
        &[
            signer_info.clone(),
            order_info.clone(),
            system_program_info.clone(),
        ],
    )?;

    // next we are going to try and create a token account owned by the
    ↪   program
    // but whose address is derived from the order account
    // TODO: for subscriptions, should this use the subscription account as
    ↪   the base?
    create_program_owned_associated_token_account(
        program_id,
        &[
            signer_info.clone(),
            order_info.clone(),
            seller_token_info.clone(),
            mint_info.clone(),
            pda_info.clone(),
            token_program_info.clone(),
            system_program_info.clone(),
            rent_sysvar_info.clone(),
        ],
        rent,
    )?;

    // Transfer payment amount to associated seller token account...
    invoke(
        &spl_token::instruction::transfer(
            token_program_info.key,
            buyer_token_info.key,
            seller_token_info.key,
            signer_info.key,
            &[&signer_info.key],
            amount,
        )
        .unwrap(),
        &[
            buyer_token_info.clone(),
            seller_token_info.clone(),
            signer_info.clone(),
            token_program_info.clone(),
        ],
    )?;

    if Pubkey::new_from_array(merchant_account.sponsor) ==
    ↪  Pubkey::from_str(PROGRAM_OWNER).unwrap()
    {
        // Transferring processing fee to the program owner...
        invoke(
            &system_instruction::transfer(
                &signer_info.key,
                program_owner_info.key,
                merchant_account.fee,
            ),
            &[
                signer_info.clone(),
                program_owner_info.clone(),
                system_program_info.clone(),
            ],
        )?;
    } else {
        // we need to pay both the program owner and the sponsor
        let (program_owner_fee, sponsor_fee) =
        ↪  get_amounts(merchant_account.fee, SPONSOR_FEE);
        // Transferring processing fee to the program owner and sponsor...
        invoke(
            &system_instruction::transfer(
```

```rust
                &signer_info.key,
                program_owner_info.key,
                program_owner_fee,
            ),
            &[
                signer_info.clone(),
                program_owner_info.clone(),
                system_program_info.clone(),
            ],
        )?;
        invoke(
            &system_instruction::transfer(&signer_info.key, sponsor_info.key, sponsor_fee),
            &[
                signer_info.clone(),
                sponsor_info.clone(),
                system_program_info.clone(),
            ],
        )?;
    }

    // get the order account
    // TODO: ensure this account is not already initialized
    let mut order_account_data = order_info.try_borrow_mut_data()?;
    // Saving order information...
    let order = OrderAccount {
        discriminator: order_account_type,
        status: OrderStatus::Paid as u8,
        created: timestamp,
        modified: timestamp,
        merchant: merchant_info.key.to_bytes(),
        mint: mint_info.key.to_bytes(),
        token: seller_token_info.key.to_bytes(),
        payer: signer_info.key.to_bytes(),
        expected_amount: amount,
        paid_amount: amount,
        order_id,
        secret,
        data,
    };

    order.pack(&mut order_account_data);

    // ensure order account is rent exempt
    if !rent.is_exempt(order_info.lamports(), order_account_size) {
        return Err(ProgramError::AccountNotRentExempt);
    }

    Ok(())
}

```

- Call Stack

```
fn entrypoint::process_instruction(){// src/entrypoint.rs:11:1: 23:2 }
    fn processor::<impl
    ↪  instruction::PaymentProcessorInstruction>::process(){//
    ↪  src/processor.rs:15:5: 61:6 }
        fn engine::pay::process_express_checkout(){//
        ↪  src/engine/pay.rs:331:1: 349:2 }
            fn engine::pay::process_order(){// src/engine/pay.rs:137:1:
            ↪  329:2 }

```

- description:
- link:
- alleviation:

Appendix

Copied from https://leaderboard.certik.io/projects/aave

Finding Categories

Gas Optimization

Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.

Mathematical Operations

Mathematical Operation findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.

Logical Issue

Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.

Language Specific

Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.

Coding Style

Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.

Checksum Calculation Method

The “Checksum” field in the “Audit Scope” section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux “sha256sum” command against the target file.

Disclaimer