KUDELSKI SECURITY

Security Assessment

Findings and Recommendations Report Presented to:

Port.Finance

November 01, 2021
Version: 2.1

Presented by:

Kudelski Security, Inc.
5090 North 40th Street, Suite 450
Phoenix, Arizona 85018

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

Overview

Port.Finance engaged Kudelski Security to perform a Security Assessment.

The assessment was conducted remotely by the Kudelski Security Team. Testing took place on August 2 - August 25, 2021, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks that were discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the result of our tests.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security Teams took to identify and validate each issue, as well as any applicable recommendations for remediation.

Additional reviews where conducted during September 13 – September 30, 2021.

Key Findings

The following are the major themes and issues identified during the testing period. These, along with other items, within the findings section, should be prioritized for remediation to reduce to the risk they pose.

- Precision loss during SLOTS_PER_YEAR calculation
- Reserve config can be modified by anyone.
- Flash loans without fee possible

During the test, the following positive observations were noted regarding the scope of the engagement:

- The team was very supportive and open to discuss the design choices made

Based on the account relationship graphs or reference graphs and the formal verification we can conclude that the reviewed code implements the documented functionality.

As of the issuance of this report, all findings have been resolved or risk accepted.

Scope and Rules Of Engagement

Kudelski performed an Security Assessment for Port.Finance. The following table documents the targets in scope for the engagement. No additional systems or resources were in scope for this assessment.

The source code was supplied through private repositories at https://github.com/port-finance/lending/tree/fixed_price with the commit hash 78855c4980bd5ed673846010c988ba65047bcdd4

Files included in the code review	
lending/token-lending ├── flash_loan_design.md ├── program │ ├── Cargo.toml │ └── src │ ├── entrypoint.rs │ ├── error.rs │ ├── instruction.rs │ ├── lib.rs │ ├── math │ │ ├── common.rs │ │ ├── decimal.rs │ │ ├── mod.rs │ │ └── rate.rs │ ├── processor.rs │ ├── pyth.rs │ └── state │ ├── last_update.rs │ ├── lending_market.rs │ ├── mod.rs │ ├── obligation.rs │ └── reserve.rs └── README.md	Rust implementation of the token-lending program
staking/ ├── program │ ├── Cargo.toml │ ├── src │ │ ├── entrypoint.rs │ │ ├── error.rs │ │ ├── instruction.rs │ │ ├── lib.rs │ │ ├── math │ │ │ ├── common.rs │ │ │ ├── decimal.rs │ │ │ ├── mod.rs │ │ │ └── rate.rs │ │ ├── processor.rs │ │ └── state │ │ ├── mod.rs │ │ ├── stake_account.rs │ │ └── staking_pool.rs	Rust implementation of the staking program

Table 1: Scope

TECHNICAL ANALYSIS & FINDINGS

During the Security Assessment, we discovered 3 findings that had a HIGH severity rating, as well as 1 INFORMATIONAL

The following chart displays the findings by severity.

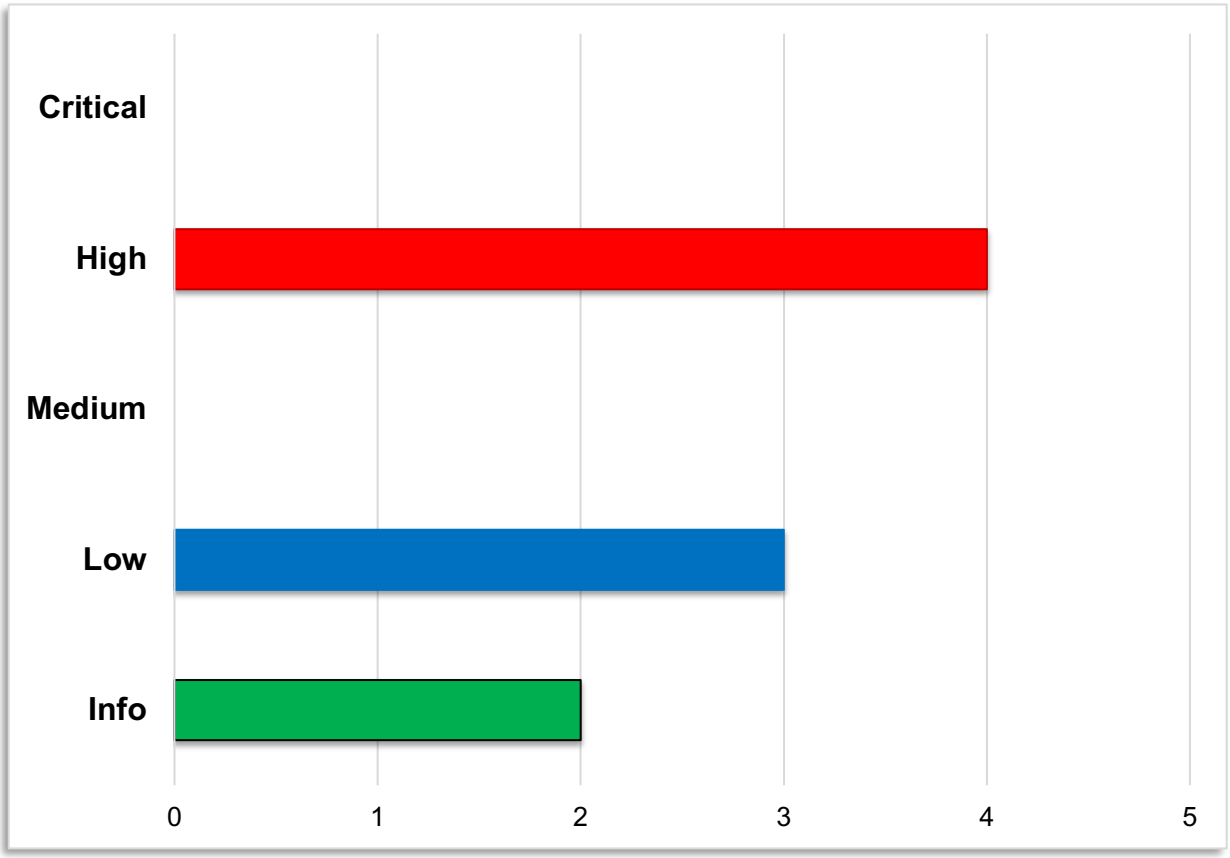

Figure 1: Findings by Severity

Findings

The *Findings* section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

#	Severity	Description
KS-PRTFNC1-F-01	High	Precision loss during SLOTS_PER_YEAR calculation
KS-PRTFNC1-F-02	High	Reserve config can be modified by anyone.
KS-PRTFNC1-F-03	High	Flash loans without fee possible
KS-PRTFNC1-F-05	High	Update Unlimited staking pool deposit possible
KS-PRTFNC1-F-06	Low	Misleading log message if it is too early to claim reward
KS-PRTFNC1-F-07	Low	Create stake account instruction allows passing staking pool not owned by program
KS-PRTFNC1-F-08	Low	Claim reward instruction allows passing stake account not owned by program and transfer tokens without error
KS-PRTFNC1-F-04	Informational	Update reserve config is dangerous from user point of view
KS-PRTFNC1-F-09	Informational	Logging public keys is increasing binary size significantly

Table 2: Findings Overview

Technical analysis

Based on the source code the following account relationship graphs or reference graphs was made to verify the validity of the code as well as comfirmating that the intended functionality was implemented correctly and to the extent that the state of the repository allowed.

A number of further investigations were made which conluded that they did not pose a risk to the application. They were

- No potential panics were detected
- No potential errors regarding wraps/unwraps, expect and wildcards
- No internal unintentional unsafe references

Authorization

The review used relationship graphs to show the relations between account input passed to the instructions of the program. The relations are used to verify if the authorization is sufficient for invoking each instruction. The graphs show if any unreferenced accounts exist. Accounts that are not referred to by trusted accounts can be replaced by any account of an attacker's choosing and thus pose a security risk.

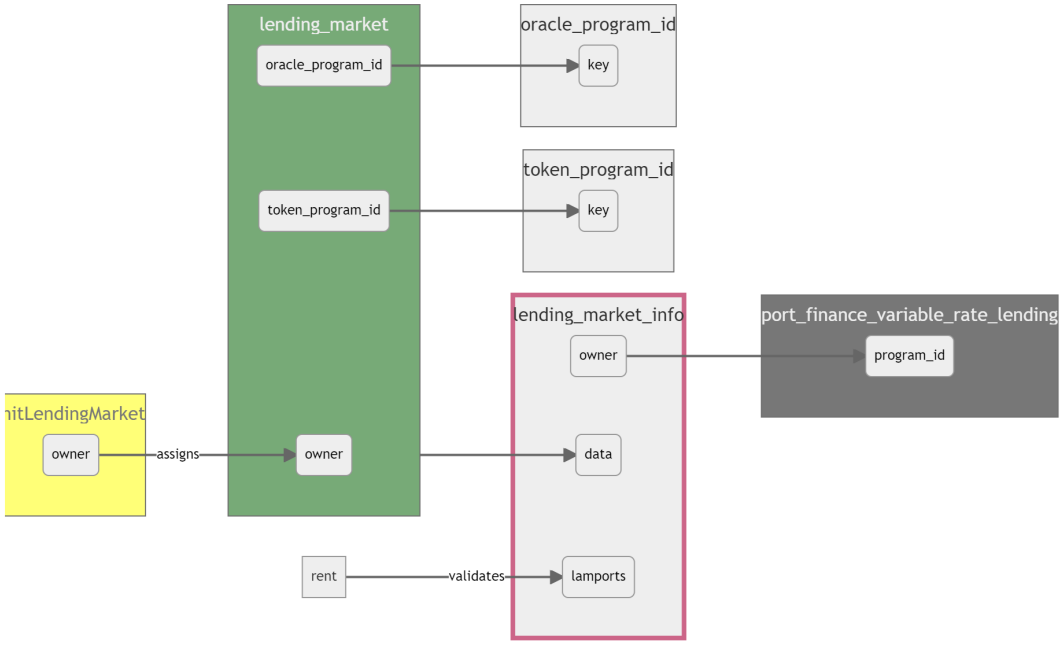

Figure 2: process_init_lending_market

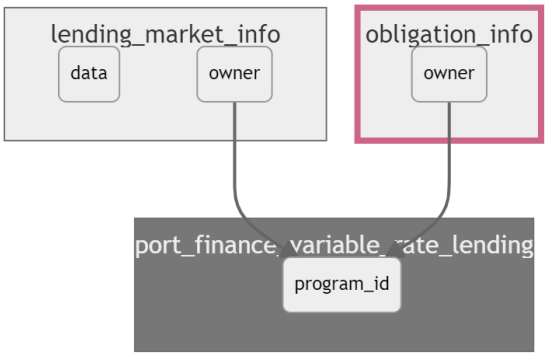

Figure 3: process_init_obligation

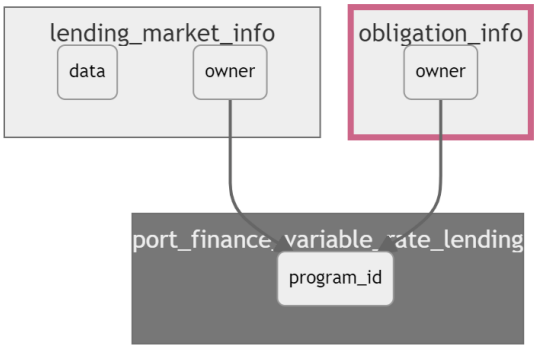

Figure 4: process_init_reserve

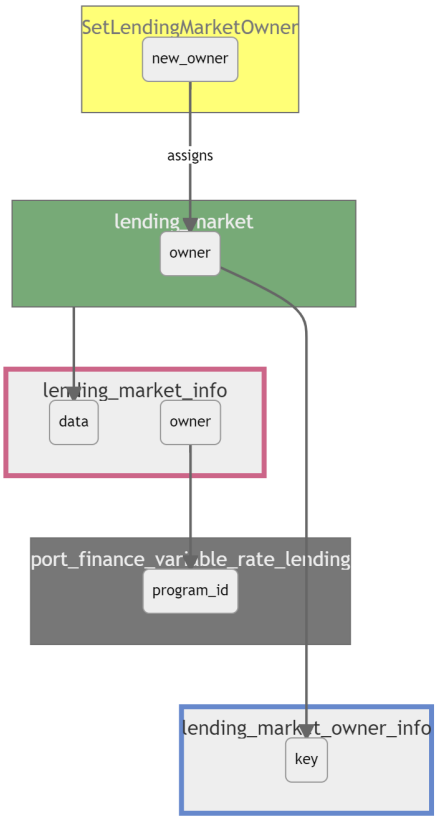

Figure 5: process_set_lending_market_owner

In particular, the graphs will show if signing accounts are referred to. If a signing account is not referred to then any account can be used to sign the transaction causing insufficient authorization.

Conclusion

Based on the account relationship graphs or reference graphs and the formal verification we can conclude that the code implements the documented functionality to the extent of the code reviewed.

Technical Findings

General Observations

Port.finance token lending program is mostly based on Solana program library token lending program. This makes the port.finance code as good/bad as the original one.

It is unrealistic to require better documentation, smaller modules, bigger code coverage and cleaner authorisation checks, because the most important priority is to keep the code as close as possible to the Solana official repo. Especially essential it is to merge all the latest commits, even if they seem to be optional (like "lending: add extra owner check", which solves the found high severity issue).

The added code, which could be reviewed is recently added. By adding `DepositReserveLiquidityAndObligationCollateral` and `WithdrawObligationCollateralAndRedeemReserveCollateral` it simplifies some common steps within the program.

The `UpdateReserve` functionality is something that we deem very problematic. It allows the lending market owner all rights which could be used for malicious purposes. It would at least need to be improved to solve the description in the recommendations in the technical findings. Without this change, it is going to be very hard for users to trust such application as the lending market owner may change the rules at any given point in time.

There are references to insecure crates in the project

```
Crate:         failure
Version:       0.1.8
Warning:       unmaintained
Title:         failure is officially deprecated/unmaintained
Date:          2020-05-02
ID:            RUSTSEC-2020-0036
URL:           https://rustsec.org/advisories/RUSTSEC-2020-0036
Dependency tree:
failure 0.1.8

Crate:         net2
Version:       0.2.37
Warning:       unmaintained
Title:         `net2` crate has been deprecated; use `socket2` instead
Date:          2020-05-01
ID:            RUSTSEC-2020-0016
URL:           https://rustsec.org/advisories/RUSTSEC-2020-0016
Dependency tree:
net2 0.2.37

Crate:         port-finance-variable-rate-lending
Version:       0.1.0
Warning:       yanked
Dependency tree:
port-finance-variable-rate-lending 0.1.0
```

Staking program is a well written smart contract. All the keys, signatures and relations are checked properly as suggested by the documentation. The math logic easily pass all the corner cases tests without adding extra complexity.

The Staking program is designed to be run by the other program, as it gives too much power for staking pool owner.

The only places, where something could go wrong is caused by lack of an ownership check, when the account is written. It is very important to remember, that ownership should always be checked, because if it happens to keep the account unchanged, Solana runtime will not fail, even if the account is owned by a malicious program.

The Staking program analysis shows that developer must be aware that unfunded account ownership can be changed after transaction. To prevent such an attack, a check to assure that data is not empty should be present.

Precision loss during SLOTS_PER_YEAR calculation

Finding ID: KS-PRTFNC1-F-01

Severity: [**High**]

Status: [**Risk Accepted – Determined that plausible impact low in current configuration**]

Description

The interest rate depends on the borrow rate and the calculated number of slots per year.

Proof of Issue

Filename: state/mod.rs

Beginning Line Number: 35

```
/// Number of slots per year
pub const SLOTS_PER_YEAR: u64 =
    DEFAULT_TICKS_PER_SECOND / DEFAULT_TICKS_PER_SLOT * SECONDS_PER_DAY * 365;
```

We have precision loss here: DEFAULT_TICKS_PER_SECOND / DEFAULT_TICKS_PER_SLOT.

```
pub const DEFAULT_TICKS_PER_SECOND: u64 = 160
pub const DEFAULT_TICKS_PER_SLOT: u64 = 64
pub const SECONDS_PER_DAY: u64 = 24 * 60 * 60
pub const SLOTS_PER_YEAR: u64 =
    DEFAULT_TICKS_PER_SECOND / DEFAULT_TICKS_PER_SLOT * SECONDS_PER_DAY * 365;
```

The division should be the last operation to maximize the precision. In this case, we have less slots per year than we expect. Affected variables are

```
let slot_interest_rate = current_borrow_rate.try_div(SLOTS_PER_YEAR)?;
let compounded_interest_rate = Rate::one()
    .try_add(slot_interest_rate)?
    .try_pow(slots_elapsed)?;
self.cumulative_borrow_rate_wads = self
    .cumulative_borrow_rate_wads
    .try_mul(compounded_interest_rate)?;
self.borrowed_amount_wads = self
    .borrowed_amount_wads
    .try_mul(compounded_interest_rate)?;
```

As can be seen, an erroneous number of slots could cause inaccurate `slot_interest_rate` calculation and it will be lower than expected making borrows cheaper, and loans less profitable.

Severity and Impact Summary

Loss of funds will be the result as the interest rate will be calculated inaccurately. In the Solana program module slot time is defined as 400ms, but is 500ms as reflected in the actual code, this is because of the precision loss during the integer division.

Recommendation

A fix seems to be easy to apply, but the following change

```
pub const SLOTS_PER_YEAR: u64 =
    DEFAULT_TICKS_PER_SECOND * SECONDS_PER_DAY * 365 / DEFAULT_TICKS_PER_SECOND
```

has unexpected side effects. As the actual author of that line of code mentioned, the constant values of `DEFAULT_TICKS_PER_SECOND` and `DEFAULT_TICKS_PER_SLOT` specifies the desired "time per slot" for the cluster which is `400 ms` per slot.

However, in reality the `mainnet-beta` cluster is closer to `550 ms` per slot. So by pure luck the `20%` loss actually causes the result to be a better estimate. He agreed that this is a serious issue as it is very unclear why this works for now. It will be a critical issue when the mainnet-beta reaches its desired slot rate.

References

- Discussion about potential fix is ongoing: https://github.com/solana-labs/solana-program-library/pull/2210

Reserve config can be modified by anyone

Finding ID: KS-PRTFNC1-F-02

Severity: [**High**]

Status: [**Resolved**]

Description

Update reserve config instruction is designed to set new parameters for reserve.

Proof of Issue

Filename: processor.rs

Beginning Line Number: 1953

```
fn process_update_reserve(
    program_id: &Pubkey,
    config: ReserveConfig,
    accounts: &[AccountInfo],
) -> ProgramResult {
    validate_reserve_config(config)?;

    let account_info_iter = &mut accounts.iter();
    let reserve_info = next_account_info(account_info_iter)?;
    let lending_market_info = next_account_info(account_info_iter)?;
    let lending_market_authority_info = next_account_info(account_info_iter)?;
    let lending_market_owner_info = next_account_info(account_info_iter)?;
    let rent_info = next_account_info(account_info_iter)?;
    let rent = &Rent::from_account_info(rent_info)?;
    let token_program_id = next_account_info(account_info_iter)?;

    assert_rent_exempt(rent, reserve_info)?;
    let mut reserve = Reserve::unpack(&reserve_info.data.borrow())?;

    if reserve_info.owner != program_id {
        msg!("Reserve provided is not owned by the lending program");
        return Err(LendingError::InvalidAccountOwner.into());
    }

    let lending_market = LendingMarket::unpack(&lending_market_info.data.borrow())?;
    if lending_market_info.owner != program_id {
        msg!("Lending market provided is not owned by the lending program");
        return Err(LendingError::InvalidAccountOwner.into());
    }
    if &lending_market.token_program_id != token_program_id.key {
        msg!("Lending market token program does not match the token program provided");
        return Err(LendingError::InvalidTokenProgram.into());
    }
    if &lending_market.owner != lending_market_owner_info.key {
        msg!("Lending market owner does not match the lending market owner provided");
        return Err(LendingError::InvalidMarketOwner.into());
    }
    if !lending_market_owner_info.is_signer {
        msg!("Lending market owner provided must be a signer");
        return Err(LendingError::InvalidSigner.into());
    }

    let authority_signer_seeds = &[
        lending_market_info.key.as_ref(),
        &[lending_market.bump_seed],
    ];
    let lending_market_authority_pubkey =
        Pubkey::create_program_address(authority_signer_seeds, program_id)?;
    if &lending_market_authority_pubkey != lending_market_authority_info.key {
        msg!(
            "Derived lending market authority does not match the lending market authority provided"
        );
        return Err(LendingError::InvalidMarketAuthority.into());
    }

    reserve.config = config;

    Reserve::pack(reserve, &mut reserve_info.data.borrow_mut())?;

    Ok(())
}
```

The following check is missing:

```
if &reserve.lending_market != lending_market_info.key {
    msg!("Reserve lending market does not match the lending market provided");
    return Err(LendingError::InvalidAccountInput.into());
}
```

Severity and Impact Summary

As this affects the lending market at a high degree, only the lending market owner should be allowed to invoke such an instruction. In the actual code it is checked if the lending market owner has signed the transaction, but any check as if the reserve lending market is the same as the lending market passed to the instruction is missing. This lets an attacker to pass his own lending market, sign the transaction and update any reserve config.

An attacker can therefore steal tokens manipulating parameters like:

- optimal utilisation rate
- loan to value rate
- liquidation bonus rate
- liquidation threshold
- min, optimal and max borrow rate
- flash loan, borrow and host fees

Recommendation

The following check should be applied as soon as possible.

```
if &reserve.lending_market != lending_market_info.key {
    msg!("Reserve lending market does not match the lending market provided");
    return Err(LendingError::InvalidAccountInput.into());
}
```

References

- This fix has already been applied in similar project https://github.com/solendprotocol/solana-program-library/commit/132d74cf171dac66f896c4009ad6836f8c0b3799#diff-0777d207db84ba35f5fdecd53e234db22dfd582fa632bd5628cf2eb045c443f8

Flash loans without fee possible

Finding ID: KS-PRTFNC1-F-03

Severity: [High]

Status: [Resolved]

Description

Flash loan instruction takes fee rates, fee receiver and liquidity supply from reserve account.

Proof of Issue

Filename: processor.rs

Beginning Line Number: 1715

There are only three checks for reserve_account_info

```
let mut reserve = Reserve::unpack(&reserve_account_info.data.borrow())?;
if &reserve.lending_market != lending_market_info.key {
    msg!("Invalid reserve lending market account");
    return Err(LendingError::InvalidAccountInput.into());
}
if &reserve.liquidity.supply_pubkey != source_liquidity_info.key {
    msg!("Reserve liquidity supply must be used as the source liquidity provided");
    return Err(LendingError::InvalidAccountInput.into());
}

if &reserve.liquidity.fee_receiver != flash_loan_fee_receiver_account_info.key {
    msg!("Reserve liquidity fee receiver does not match the flash loan fee receiver provided");
    return Err(LendingError::InvalidAccountInput.into());
}
```

For accounts which does not have to be writable, the most important check is

```
if &reserve_account_info.owner != program_id {
    msg!("Reserve is not owned by the lending program");
    return Err(LendingError::InvalidAccountOwner.into());
}
```

and that check is not present in the code.

A reserve account can actually be modified by a call as decscribed below, which finally won't cause any data to be changed and runtime validator won't fail and let the call pass.

```
...
reserve.liquidity.borrow(flash_loan_amount_decimal)?;
...
reserve.liquidity.repay(flash_loan_amount, flash_loan_amount_decimal)?;
```

It is possible then, to prepare `mutating_program` which manipulates the `reserve` configs and keys

```
let init_my_reserve = {
    let accounts = vec![
        AccountMeta::new(my_reserve, false)
    ];
    let mut data = client.get_account_data(&real_reserve)?;
    let mut r = Reserve::unpack_from_slice(data.as_slice())?;
    r.liquidity.fee_receiver = my_spl_account;  //we can
    r.config.fees.flash_loan_fee_wad = WAD / 2; //set whatever
    r.config.fees.host_fee_percentage = 99;     //we want
    r.pack_into_slice(data.as_mut_slice());
    Instruction::new_with_bytes(mutating_program_id, data.as_slice(), accounts)
};
```

Afterwards, a well prepared `my_reserve` allows to take flash loan without fee

```
let flash_loan = {
    let accounts = &[
        AccountMeta::new(reserve_liquidity_supply, false),
        AccountMeta::new(my_spl_account, false),
        AccountMeta::new_readonly(my_reserve, false),
        AccountMeta::new_readonly(lending_market, false),
        AccountMeta::new_readonly(lending_market_authority, false),
        AccountMeta::new_readonly(flash_callback_id, false),
        AccountMeta::new(my_spl_account, false),
        AccountMeta::new(my_spl_account, false),
        AccountMeta::new_readonly(token_program_id, false),
        AccountMeta::new_readonly(my_spl_account_authority, true),
    ];
    let bytes = &LendingInstruction::FlashLoan {
        amount: u64::MAX
    }.pack();
    Instruction::new_with_bytes(lending_program_id, bytes, accounts)
};
```

Severity and Impact Summary

To be sure, that all reserve data are consistent, it should be checked if the reserve account owner is the token lending program. Such a check is missing and the runtime will not fail the call. This because the reserve account is actually not modified (the same amount is borrowed and the repaid). This lets an attacker to pass his own reserve as parameter.

Recommendation

The following fix should be applied as soon as possible

https://github.com/solana-labs/solana-program-library/commit/23c487dd9c08803cfbb7039564d55795478b3b61

References

- https://github.com/solana-labs/solana-program-library/commit/23c487dd9c08803cfbb7039564d55795478b3b61

Update Unlimited staking pool deposit possible

Finding ID: KS-PRTFNC1-F-05

Severity: [**High**]

Status: [**Resolved**]

Description

Only staking program is allowed to modify stake account during deposit and withdraw. Staking program id is not verified, so it is possible to pass empty program as staking program for withdraw and achieve unlimited deposited amount for stake account without storing any collateral.

Proof of Issue

Filename: processor.rs

Beginning Line Number: 1154

```
if account_info_iter.peek().is_some() {
    let stake_account_info = next_account_info(account_info_iter)?;
    let staking_pool_info = next_account_info(account_info_iter)?;
    let staking_program_id = next_account_info(account_info_iter)?;
    let withdraw_reserve_info = Reserve::unpack(&withdraw_reserve_info.data.borrow())?;
    if withdraw_reserve_info
        .config
        .deposit_staking_pool
        .map_or(true, |k| k != *staking_pool_info.key)
    {
        msg!("Invalid staking pool, not the one corresponded to the reserve");
        return Err(LendingError::InvalidStakingPool.into());
    }
    withdraw_from_staking_program(
        program_id,
        withdraw_amount,
        lending_market_info,
        lending_market_authority_info,
        clock_info,
        stake_account_info,
        staking_pool_info,
        staking_program_id,
        *obligation_owner_info.key,
    )
}
```

```rust
fn withdraw_from_staking_program<'a>(
    program_id: &Pubkey,
    collateral_amount: u64,
    lending_market_info: &AccountInfo<'a>,
    lending_market_authority_info: &AccountInfo<'a>,
    clock_info: &AccountInfo<'a>,
    stake_account_info: &AccountInfo<'a>,
    staking_pool_info: &AccountInfo<'a>,
    staking_program_id: &AccountInfo<'a>,
    owner: Pubkey,
) -> ProgramResult {
    let lending_market = LendingMarket::unpack(&lending_market_info.data.borrow())?;

    let authority_signer_seeds = &[
        lending_market_info.key.as_ref(),
        &[lending_market.bump_seed],
    ];
    let lending_market_authority_pubkey =
        Pubkey::create_program_address(authority_signer_seeds, program_id)?;

    if &lending_market_authority_pubkey != lending_market_authority_info.key {
        msg!(
            "Derived lending market authority does not match the lending market authority provided"
        );
        return Err(LendingError::InvalidMarketAuthority.into());
    }
    let stake_account = StakeAccount::unpack(&stake_account_info.data.borrow())?;

    if stake_account.owner != owner {
        msg!("You are not withdrawing from your stake account");
        return Err(LendingError::InvalidStakeAccount.into());
    }

    invoke_signed(
        &withdraw(
            *staking_program_id.key,
            collateral_amount,
            *lending_market_authority_info.key,
            *stake_account_info.key,
            *staking_pool_info.key,
        ),
        &(vec![
            lending_market_authority_info.clone(),
            stake_account_info.clone(),
            staking_pool_info.clone(),
            staking_program_id.clone(),
            clock_info.clone(),
        ])[..],
        &[authority_signer_seeds],
    )?;

    Ok(())
}
```

```
pub fn withdraw(
    program_id: Pubkey,
    amount: u64,
    authority: Pubkey,
    stake_account: Pubkey,
    staking_pool: Pubkey,
) -> Instruction {
    let write_accounts = create_write_accounts(vec![stake_account, staking_pool]);

    let accounts = vec![AccountMeta::new_readonly(authority, true)]
        .into_iter()
        .chain(write_accounts)
        .chain(vec![AccountMeta::new_readonly(sysvar::clock::id(), false)])
        .collect();

    Instruction {
        program_id,
        accounts,
        data: Withdraw(amount).pack(),
    }
}
```

Severity and Impact Summary

In the code above, staking program id is not verified anywhere, and it allows to pass any program. Of course it is impossible to transfer tokens directly from staking pool, because spl account is not passed to the program. Direct modification of staking pool and stake account is also infeasible, because those accounts are owned by staking program. However, lack of modification which could be achieved by passing empty program will allow making deposited amount bigger and bigger by invoking many deposit-withdraw instruction pairs

Recommendation

The safest method for fixing this issue is to compare staking program id to hardcoded value. Other method is to check if staking program id is the same as staking pool owner, but it allows passing staking pool owned by malicious program during reserve update and run such program during deposit/withdraw instruction.

References

- https://docs.solana.com/developing/programming-model/runtime

Misleading log message if it is too early to claim reward

Finding ID: KS-PRTFNC1-F-06

Severity: [**Low**]

Status: [**Open**]

Description

If it's too early to claim reward, user get information saying: "It is the time to claim reward yet".

Proof of Issue

Filename: processor.rs

Beginning Line Number: 310

```
if clock.slot < staking_pool.earliest_reward_claim_time {
    msg!("It is the time to claim reward yet");
    return Ok(());
}
```

Severity and Impact Summary

It seems that "not" word is missing.

Recommendation

Problem can be fixed by replacing "is" with "is not".

References

- N/A

Create stake account instruction allows passing staking pool not owned by program.

Finding ID: KS-PRTFNC1-F-07

Severity: [Low]

Status: [Resolved]

Description

Checking if program owns staking pool in `create_stake_account` instruction is not enough to prove that staking pool is actually owned by the program.

Proof of Issue

Filename: processor.rs

Beginning Line Number: 181

```
fn process_create_stake_account(program_id: &Pubkey, accounts: &[AccountInfo]) -> ProgramResult {
    let account_info_iter = &mut accounts.iter();
    if let [stake_account_info, staking_pool_info, stake_account_owner_info, rent_info] =
    next_account_infos(account_info_iter, 4)?
    {
        let rent = &Rent::from_account_info(rent_info)?;
        assert_rent_exempt(rent, stake_account_info)?;

        if stake_account_info.owner != program_id {
            msg!("Stake account is not owned by the staking program");
            return Err(StakingError::InvalidAccountOwner.into());
        }

        if staking_pool_info.owner != program_id {
            msg!("Staking pool is not owned by the staking program");
            return Err(StakingError::InvalidAccountOwner.into());
        }

        let mut stake_account = assert_uninitialized::<StakeAccount>(stake_account_info)?;

        stake_account.init(*stake_account_owner_info.key, *staking_pool_info.key)?;

        StakeAccount::pack(stake_account, &mut stake_account_info.data.borrow_mut())?;
        Ok(())
    } else {
        msg!("Wrong number of accounts");
        Err(StakingError::InvalidArgumentError.into())
    }
}
```

Severity and Impact Summary

Even if instruction seems to be well protected, one check is missed. If staking pool is unfunded, it is possible to set owner in the same transaction to staking program, so all the checks will pass. It turned out that for unfunded account ownership change is not stored anywhere and after transaction account can be assigned to malicious program.

In staking program, it is not possible to do anything more with stake account having invalid staking pool.

Recommendation

To protect instruction from that kind of attack, staking pool account should be checked if contains any data (unfunded accounts have no data).

References

- https://docs.solana.com/developing/runtime-facilities/programs

Claim reward instruction allows passing stake account not owned by program and transfer tokens without error.

Finding ID: KS-PRTFNC1-F-08

Severity: [**Low**]

Status: [**Resolved**]

Description

In claim reward instruction it is not checked if account is owned by program, when account will be written, as it is checked by solana runtime. However as seen in flash loan bug, if account remain unchanged, no error will be raised!

Proof of Issue

Filename: processor.rs

Beginning Line Number: 294

```
pub fn claim_reward(&mut self, current_rate: Decimal) -> Result<u64, ProgramError> {
    let reward = self.calculate_reward(current_rate)?;
    self.unclaimed_reward_wads = self.unclaimed_reward_wads.try_add(reward)?;
    let reward_lamports = self.unclaimed_reward_wads.try_floor_u64()?;
    self.unclaimed_reward_wads = self.unclaimed_reward_wads.try_sub(reward_lamports.into())?;
    self.start_rate = current_rate;
    Ok(reward_lamports)
}
```

Severity and Impact Summary

As official documentation suggests, if account is writing during instruction, it must be owned by program, otherwise transaction will fail. There is one case, when transaction will not fail: if data remains unchanged. In claim reward instruction, it is possible to pass any stake account, because stake account owner is not checked anywhere. It makes whole security relay on the above referenced piece of code.

If it was possible to generate non zero `reward_lamports` without changing `self`, it would be possible to transfer tokens from any staking pool.

However calculation part for staking pool and stake account is well written and transferring more than zero tokens is not possible. It is still unexpected, that instruction may pass, when invalid stake account is used.

Recommendation

Program should always verify the owner of an account.

References

- https://docs.solana.com/developing/programming-model/runtime

Update reserve config is dangerous from user point of view

Finding ID: KS-PRTFNC1-F-04

Severity: [Informational]

Status: [Resolved]

Description

From the user point of view, it is quite unexpected, that the service conditions of use can change at any time, to potentially an unacceptable level.

Proof of Issue

Filename: processor.rs

Beginning Line Number: 1953

```
fn process_update_reserve(
```

Severity and Impact Summary

There are a few bad things the lending market owner can do by changing reserve config

- Set extremely high borrow rates if lending market owner is lending.
- Set borrow rates to 0 if lending market owner is borrowing.
- Set liquidation threshold to 0 and make all obligation unhealthy.
- Set LTV to 0 and make impossible to withdraw any obligation.
- Set flash fees to 0 and take free flash loans.

Recommendation

Reserve config should not be mutated at any time (the same should be true for money lendings). The proper solution would be to create a new reserve, with a new config and mark the old one as obsolete. An obsolete reserve should not be allowed to borrow nor to take flash loans.

References

- N/A

Logging public keys is increasing binary size significantly

Finding ID: KS-PRTFNC1-F-09

Severity: [Informational]

Status: [Resolved]

Description

Formatting public keys into a message requires about 20kB.

Proof of Issue

Filename: processor.rs

Beginning Line Number: 52

```
fn assert_rent_exempt(rent: &Rent, account_info: &AccountInfo) -> ProgramResult {
    if !rent.is_exempt(account_info.lamports(), account_info.data_len()) {
        msg!("lamports {} pubkey {}", account_info.lamports(), account_info.key);
        msg!("required minimum lamports {}", &rent.minimum_balance(account_info.data_len()).to_string());
        Err(StakingError::NotRentExempt.into())
    } else {
        Ok(())
    }
}

fn assert_uninitialized<T: Pack + IsInitialized>(
    account_info: &AccountInfo,
) -> Result<T, ProgramError> {
    let account: T = T::unpack_unchecked(&account_info.data.borrow())?;
    if account.is_initialized() {
        msg!("The account {} is already init", account_info.key);
        Err(StakingError::AlreadyInitialized.into())
    } else {
        Ok(account)
    }
}
```

Severity and Impact Summary

Actually binary size: 223 024 B

Binary size after removing public key logs: 206 264 B

Recommendation

Logs above aren't giving too much informations and they can be safely removed to save almost 10% of binary size.

References

- N/A

METHODOLOGY

Kudelski Security uses the following high-level methodology when approaching engagements. They are broken up into the following phases.

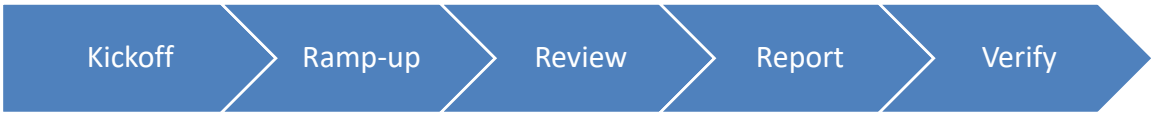

Figure 6: Methodology Flow

Kickoff

The project is kicked all of the sales process has concluded. We typically set up a kickoff meeting where project stakeholders are gathered to discuss the project as well as the responsibilities of participants. During this meeting we verify the scope of the engagement and discuss the project activities. It's an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff there is an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

Ramp-up

Ramp-up consists of the activities necessary to gain proficiency on the particular project. This can include the steps needed for familiarity with the codebase or technological innovation utilized. This may include, but is not limited to:

- Reviewing previous work in the area including academic papers
- Reviewing programming language constructs for specific languages
- Researching common flaws and recent technological advancements

Review

The review phase is where a majority of the work on the engagement is completed. This is the phase where we analyze the project for flaws and issues that impact the security posture. Depending on the project this may include an analysis of the architecture, a review of the code, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol
2. Review of the code written for the project
3. Compliance of the code with the provided technical documentation

The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom built scripts and tools were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following sections.

Code Safety

We analyzed the provided code, checking for issues related to the following categories:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This list is general list and not comprehensive, meant only to give an understanding of the issues we are looking for.

Technical Specification Matching

We analyzed the provided documentation and checked that the code matches the specification. We checked for things such as:

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

Reporting

Kudelski Security delivers a preliminary report in PDF format that contains an executive summary, technical details, and observations about the project.

The executive summary contains an overview of the engagement including the number of findings as well as a statement about our general risk assessment of the project as a whole. We may conclude that the overall risk is low, but depending on what was assessed we may conclude that more scrutiny of the project is needed.

We not only report security issues identified but also informational findings for improvement categorized into several buckets:

- Critical

- High
- Medium
- Low
- Informational

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

As we perform the audit, we may identify issues that aren't security related, but are general best practices and steps, that can be taken to lower the attack surface of the project. We will call those out as we encounter them and as time permits.

As an optional step, we can agree on the creation of a public report that can be shared and distributed with a larger audience.

Verify

After the preliminary findings have been delivered, this could be in the form of the approved communication channel or delivery of the draft report, we will verify any fixes withing a window of time specified in the project. After the fixes have been verified, we will change the status of the finding in the report from open to remediated.

The output of this phase will be a final report with any mitigated findings noted.

Additional Note

It is important to note that, although we did our best in our analysis, no code audit or assessment is a guarantee of the absence of flaws. Our effort was constrained by resource and time limits along with the scope of the agreement.

While assessment the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. These is a solid baseline for severity determination.

The Classification of identified problems and vulnerabilities

There are four severity levels of an identified security vulnerability.

Critical – vulnerability that will lead to loss of protected assets

- This is a vulnerability that would lead to immediate loss of protected assets
- The complexity to exploit is low
- The probablillty of exploit is high

High - A vulnerability that can lead to loss of protected assets

- All discrepancies found where there is a security claim made in the documentation that can not be found in the code
- All mismatches from the stated and actual functionality
- Unprotected key material

- Weak encryption of keys
- Badly generated key materials
- Tx signatures not verified
- Spending of funds through logic errors
- Calculation errors overflows and underflows

Medium - a vulnerability that hampers the uptime of the system or can lead to other problems

- Insecure calls to third party libraries
- Use of untested or nonstandard or non-peer-revied crypto functions
- Program crashes leaves core dumps or write sensitive data to log files

Low - Problems that have a security impact but does not directly impact the protected assets

- Overly complex functions
- Unchecked return values from 3rd party libraries that could alter the execution flow

Informational

- General recommendations

Tools

The following tools were used during this portion of the test. A link for more information about the tool is provided as well.

Tools used during the code review and assessment

- Rust – cargo tools
- IDE modules for Rust and analysis of source code
- Cargo audit which uses https://rustsec.org/advisories/ to find vulnerabilities cargo.

RustSec.org

About RustSec
The RustSec Advisory Database is a repository of security advisories filed against Rust crates published and maintained by the Rust Secure Code Working Group.

The RustSec Tool-set used in projects and CI/CD pipelines

‘cargo-audit’ - audit Cargo.lock files for crates with security vulnerabilities.
‘cargo-deny’ - audit `Cargo.lock` files for crates with security vulnerabilities, limit the usage of particular dependencies, their licenses, sources to download from, detect multiple versions of same packages in the dependency tree and more.

KUDELSKI SECURITY CONTACTS

NAME	POSITION	CONTACT INFORMATION