Security Assessment Report

Lavarage

July 28, 2024

Summary

The Sec3 team (formerly Soteria) was engaged to conduct a thorough security analysis of the Lavarage smart contracts.

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

The initial audit focused on the following versions and revealed 8 issues or questions.

#	program	type	commit
P1	Lavarage	Solana	`70946c2cc76e11fa2a562488621ce58791fa53de`
P2	Staking	Solana	`70946c2cc76e11fa2a562488621ce58791fa53de`

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

Table of Contents

Result Overview

Issue	Impact	Status
LAVARAGE		
[P1-H-01] Misplaced exposure update	High	Resolved
[P1-M-01] Minimum loan amount limit can be bypassed	Medium	Resolved
[P1-I-01] Inaccurate account sizes	Info	Resolved
[P1-I-02] Defined but not used config in NodeWallet	Info	Acknowledged
[P1-Q-01] Unchecked fee_receipient	Question	Resolved
STAKING		
[P2-I-01] Better realloc implementation	Info	Resolved
[P2-I-02] Use conditional compilation for test-only insturction	Info	Acknowledged
[P2-I-03] Inaccurate account sizes	Info	Acknowledged

Findings in Detail

LAVARAGE

[P1-H-01] Misplaced exposure update

The "add_collateral" instruction is intended to verify the collateral amount after the user has borrowed and provided collateral through preceding instructions. However, the current implementation updates and checks the exposure for the entire pool within this instruction.

```
/* programs/lavarage/src/processor/swap.rs */
108 | pub fn add_collateral(ctx: Context<AddCollateral>, max_interest_rate: u8) -> Result<()> {
109 |     if ctx.accounts.trading_pool.max_exposure != 0 {
110 |       require_gte!(ctx.accounts.trading_pool.max_exposure,
                ctx.accounts.trading_pool.current_exposure + ctx.accounts.position_account.amount)
111 |     }
112 |     require_gte!(max_interest_rate, ctx.accounts.trading_pool.interest_rate);
113 |     let amt = ctx.accounts.to_token_account.amount;
114 |     require_keys_eq!(ctx.accounts.mint.key(), ctx.accounts.trading_pool.collateral_type.key(),
                         FlashFillError::AddressMismatch);
116 |     require!(
117 |       ((amt as u128)
118 |       .checked_mul(ctx.accounts.trading_pool.max_borrow as u128).expect("overflow")
119 |       .checked_div(10_u128.pow(ctx.accounts.mint.decimals as u32))).expect("overflow") as u64
120 |        >= ctx.accounts.position_account.amount,
121 |       FlashFillError::ExpectedCollateralNotEnough);
122 |     ctx.accounts.position_account.collateral_amount = amt;
123 |     ctx.accounts.trading_pool.current_exposure += ctx.accounts.position_account.amount;
133 | }
```

Since there are no limits on the number of times the "add_collateral" instruction can be called, malicious users can repeatedly invoke this instruction, manipulating the "current_exposure" until the limit is reached, effectively executing a denial-of-service attack.

It is recommended to move the exposure-related checks and updates to the "borrow" instruction.

Resolution

This issue has been resolved by commit 6731eb572012b36344da4cb7830fad2495510469.

LAVARAGE

[P1-M-01] Minimum loan amount limit can be bypassed

To facilitate the cleanup of bad debt, a threshold is set in the "borrow" instruction to restrict users from making very small loans. However, since "position_size" and "user_pays" are both user-controlled parameters, it is the difference between these two values that constitutes the actual loan amount. Users can bypass this check by setting the "position_size" above the threshold and providing a "user_pays" amount that is very close to the "position_size".

```
/* programs/lavarage/src/processor/swap.rs */
013 | pub fn borrow(ctx: Context<Borrow1>, position_size: u64, user_pays: u64) -> Result<()> {
014 |     require_gt!(position_size, 50000000);
015 |     //initialize
016 |     ctx.accounts.position_account.pool = ctx.accounts.trading_pool.key();
017 |     ctx.accounts.position_account.user_paid = user_pays;
018 |     ctx.accounts.position_account.amount = position_size - user_pays;
019 |     require!(position_size / user_pays < 8, FlashFillError::ExpectedCollateralNotEnough);
```

Resolution

This issue has been resolved by commit b8008a8a3e7d249961c505bc6691395b43e1e615.

LAVARAGE

[P1-I-01] Inaccurate account sizes

During the initialization of "`Pool`" and "`Position`" account types, the account sizes used are larger than necessary, resulting in some rent wastage.

The correct sizes, including the 8-byte discriminator, are as follows:

- NodeWallet: 8 + 8 + 8 + 1 + 1 + 32 = 8 + 50 = 58
- Pool: 8 + 1 + 32 + 8 + 32 + 8 + 8 = 8 + 89 = 97
- Position: 8 + 32 + 8 + 8 + 8 + 8 + 8 + 32 + 32 + 8 + 8 + 1 + 8 = 8 + 161 = 169

Resolution

This issue has been resolved by commit `1139601eefe5f2967d326598e584ce6e17b90a74`.

LAVARAGE

[P1-I-02] Defined but not used config in NodeWallet

In the definition of "NodeWallet", there are two LTV-related thresholds.

```
/* programs/lavarage/src/state/node_wallet.rs */
011 | // The maintenance LTV, at which borrowers need to add more collateral or pay back part of the
↪  loan.
012 | pub maintenance_ltv: u8,
013 |
014 | // The liquidation LTV, at which the collateral can be liquidated to cover the loan.
015 | pub liquidation_ltv: u8,
```

However, they are not properly initialized and are not utilized. Instead, the threshold used for liquidation is hard-coded to 90%.

Resolution

The team acknowledged this finding.

LAVARAGE

[P1-Q-01] Unchecked fee_receipient

In several places where fee calculations are performed, the "fee_recipient" account is used without any verification. Users can even set themselves as the "fee_recipient".

```
/* programs/lavarage/src/context/borrow.rs */
024 | /// CHECK: We just want the value of this account
025 | #[account(mut)]
026 | pub fee_receipient: UncheckedAccount<'info>,
/* programs/lavarage/src/context/repay_sol.rs */
019 | /// CHECK: We just want the value of this account
020 | #[account(mut)]
021 | pub fee_receipient: UncheckedAccount<'info>,
```

We would like to understand if the "fee_recipient" is intended to act as a referral-like role that does not require any checks by design.

Resolution

The team clarified that this is an intentional design.

STAKING

[P2-I-01] Better realloc implementation

In the data account, there exists a whitelist of type "Vec<Pubkey>", allowing the administrator to adjust its length and content subsequently. However, when adjusting its length, the "realloc" method was used directly instead of utilizing Anchor's realloc account constraint. This method does not guarantee that the account balance is sufficient to cover rent-exempt threshold.

```
/* programs/staking/src/lib.rs */
075 | pub fn update_whitelisted_stakes_size(ctx: Context<UpdateResizeDataAccount>) -> Result<()> {
076 |     let vec_size = ctx.remaining_accounts.len() * 32;
077 |     let _ = ctx.accounts.data.realloc(100 + vec_size, false);
078 |     Ok(())
079 | }
080 |
081 | pub fn update_whitelisted_stakes(ctx: Context<UpdateResizeDataAccount2>) -> Result<()> {
082 |     ctx.accounts.data.whitelisted_stake_accounts = ctx
083 |         .remaining_accounts
084 |         .into_iter()
085 |         .map(|ac| ac.key())
086 |         .collect();
087 |     Ok(())
088 | }
```

It is recommended to consolidate "update_whitelisted_stakes_size" and "update_whitelisted_stakes" into a single instruction and employ Anchor's realloc account constraint. Example code is provided below:

```
pub fn update_whitelisted_stakes(ctx: Context<UpdateResizeDataAccount>, new_whitelist: Vec<Pubkey>) ->
↪  Result<()> {
    ctx.accounts.data.whitelisted_stake_accounts = new_whitelist
        .into_iter()
        .map(|ac| ac.key())
        .collect();
    Ok(())
}

#[derive(Accounts)]
#[instruction(new_whitelist: Vec<Pubkey>)]
pub struct UpdateResizeDataAccount<'info> {
    #[account(mut,
        seeds = [b"data"],
        bump,
        realloc = 84 + new_whitelist.len() * 32,
        realloc::zero = false,
        realloc::payer = updater,
```

```
    )]
    pub data: Account<'info, DataAccount>,
    #[account(mut, address = Pubkey::from_str(env!("UPDATER_NAV")).expect(""))]
    pub updater: Signer<'info>,
    pub system_program: Program<'info, System>,
}
```

Resolution

This issue has been resolved by commits `12fae26e0c161afff7e9c41e5c8ca76bb36e7bd7` and `fb7e5df653f6b45a04b51dc41b1b5b4a88fb2088`.

STAKING

[P2-I-02] Use conditional compilation for test-only insturction

Consider using conditional compilation to avoid compiling instructions like "`mock_node_wallet`", which are intended solely for testing purposes, into the production version.

```
/* programs/staking/src/lib.rs */
021 | pub fn mock_node_wallet(ctx: Context<InitializeNodeWallet>) -> Result<()> {
022 |     ctx.accounts.new_account.node_operator = ctx.accounts.bank.key();
023 |     ctx.accounts.new_account.total_borrowed = 2300000000;
024 |     Ok(())
025 | }
```

Resolution

The team acknowledged this finding.

STAKING

[P2-I-03] Inaccurate account sizes

During the initialization of "`DataAccount`" and "`UnstakeAccount`" account types, the account sizes used are larger than necessary, resulting in some rent wastage. The correct sizes, including the 8-byte discriminator, are as follows:

- NodeWallet: 8 + 8 + 8 + 1 + 1 + 32 = 8 + 50 = 58
- DataAccount (empty vector): 8 + 8 + 8 + 32 + 8 + 8 + 8 + 4 = 8 + 76 = 84
- UnstakeAccount: 8 + 8 + 8 + 32 = 8 + 48 = 56

Resolution

The team acknowledged this finding.

Appendix: Methodology and Scope of Work

The Sec3 (formerly Soteria) audit team, which consists of Computer Science professors and industrial researchers with extensive experience in smart contract security, program analysis, testing and formal verification, performed a comprehensive manual code review, software static analysis and penetration testing.

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

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

DISCLAIMER

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

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

ABOUT

Founded by leading academics in the field of software security and senior industrial veterans, Sec3 (formerly Soteria) is a leading blockchain security company. We are also building sophisticated security tools that incorporate static analysis, penetration testing, and formal verification.

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

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

Sec3™