SECURITY AUDIT REPORT

for

GoatSwap

PeckShield

Prepared By: Xiaomi Huang

PeckShield

June 5, 2024

Document Properties

Client	GoatSwap
Title	Security Audit Report
Target	GoatSwap
Version	1.0
Author	Daisy Cao
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 5, 2024	Daisy Cao	Final Release
1.0-rc	June 3, 2024	Daisy Cao	Release Candidate #1

PeckShield

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

PeckShield

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the GoatSwap protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About GoatSwap

GoatSwap is a decentralized exchange built on Solana. It allows users to trade and swap thousands of Solana tokens in realtime with the most liquidity and the best rate. In the meantime, it also allows liquidity providers to create trading pairs and add liquidity in a trustless manner. The basic information of audited contracts is as follows:

PeckShield

Table 1.1: Basic Information of GoatSwap

Item	Description
Name	GoatSwap
Website	https://www.goatswap.com/
Type	Solana
Language	Rust
Audit Method	Whitebox
Latest Audit Report	June 5, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/goatswap-amm/goatswap-clmm (ae415ff)

And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/goatswap-amm/goatswap-clmm (e702514)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

PeckShield

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

PeckShield

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

PeckShield

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, **Return Values,** **Status Codes**	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the `GoatSwap` implementations. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■ ■ ■
Low	2	■ ■
Total	5	

PeckShield

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Swap Logic in swap_base_input.rs	Business Logic	Resolved
PVE-002	Low	Lack of Token Account Validation in swap_base_input.rs	Business Logic	Resolved
PVE-003	Low	Possible Overflow Risk in lp_tokens_to_trading_tokens()	Business Logic	Resolved
PVE-004	Medium	Possible Fake Account in Pool Initialization	Business Logic	Resolved
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

PeckShield

3 Detailed Results

3.1 Improved Swap Logic in swap_base_input.rs

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: swap_base_input.rs
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

In the GoatSwap protocol, the swap_base_input() function makes use of the constant product formula to calculate the amount of output tokens can be obtained for a given amount of input tokens. While examining the related logic, we notice that current implementation can be improved.

In the following, we show the related code snippet from the related routine, i.e., CurveCalculator::swap_base_input(). In this routine, we notice the return amount of source_amount_less_fees is directly from the input amount source_amount_swapped (line 416). However, when there is trade_fee, it is not equal to source_amount_swapped. Instead, it should be equal to source_amount, which is the actual_amount_in (line 75 and line 401).

```
400     pub fn swap_base_input(
401         source_amount: u128,
402         swap_source_amount: u128,
403         swap_destination_amount: u128,
404         trade_fee_rate: u64,
405         protocol_fee_rate: u64,
406         fund_fee_rate: u64,
407     ) -> Option<SwapResult> {
408         // debit the fee to calculate the amount swapped
409         let trade_fee = Fees::trading_fee(source_amount, trade_fee_rate)?;
410         let protocol_fee = Fees::protocol_fee(trade_fee, protocol_fee_rate)?;
411         let fund_fee = Fees::fund_fee(trade_fee, fund_fee_rate)?;
412 
413         let source_amount_less_fees = source_amount.checked_sub(trade_fee)?;
414 
```

```
        let SwapWithoutFeesResult {
            source_amount_swapped,
            destination_amount_swapped,
        } = ConstantProductCurve::swap_base_input_without_fees(
            source_amount_less_fees,
            swap_source_amount,
            swap_destination_amount,
        )?;

        let source_amount_swapped = source_amount_swapped.checked_add(trade_fee)?;
        Some(SwapResult {
            new_swap_source_amount: swap_source_amount.checked_add(source_amount_swapped
                )?,
            new_swap_destination_amount: swap_destination_amount
                .checked_sub(destination_amount_swapped)?,
            source_amount_swapped,
            destination_amount_swapped,
            trade_fee,
            protocol_fee,
            fund_fee,
        })
    }
```

Listing 3.1: `CurveCalculator::swap_base_input()`

Note the same issue is also applicable to the `swap_base_output()` routine where current `actual_amount_out` is always equal to the `result.destination_amount_swapped`, which should not be the case.

Recommendation Improve the above-mentioned routines to ensure correct amount is returned.

Status The issue has been fixed by this commit: e702514.

PeckShield

3.2 Lack of Token Account Validation in swap_base_input.rs

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `swap_base_input.rs`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

In `GoatSwap`, users can swap tokens via two following functions, i.e., `swap_base_input()` and `swap_base_output()`. In these two functions, the swapping user needs to provide `input_token_account` and `output_token_account` accounts, which are used to deduct the input tokens and receive the output tokens, respectively. While examining the related swap logic, we notice current implementation can be improved.

In the following, we show the code snippet of the related `Swap` data structure. The `input_token_account` account is provided by the user (line 523), and its mint needs to match the input token, i.e., `token::mint = input_vault.mint`. Moreover, the input token account's authority should be the current user, who actually signs the transaction, i.e., `token::authority = payer`. These two validations are currently missing and need to be in place for proper functionality and security.

```
500     pub struct Swap<'info> {
501     /// The user performing the swap
502     pub payer: Signer<'info>,
503 
504     /// CHECK: pool vault and lp mint authority
505     #[account(
506         seeds = [
507             crate::AUTH_SEED.as_bytes(),
508         ],
509         bump,
510     )]
511     pub authority: UncheckedAccount<'info>,
512 
513     /// The factory state to read protocol fees
514     #[account(address = pool_state.load()?.amm_config)]
515     pub amm_config: Box<Account<'info, AmmConfig>>,
516 
517     /// The program account of the pool in which the swap will be performed
518     #[account(mut)]
519     pub pool_state: AccountLoader<'info, PoolState>,
520 
521     /// The user token account for input token
522     #[account(mut)]
523     pub input_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
524 
525     /// The user token account for output token
526     #[account(mut)]
527     pub output_token_account: Box<InterfaceAccount<'info, TokenAccount>>,
528 
529     /// The vault token account for input token
530     #[account(
531         mut,
532         constraint = input_vault.key() == pool_state.load()?.token_0_vault ||
            input_vault.key() == pool_state.load()?.token_1_vault
533     )]
534     pub input_vault: Box<InterfaceAccount<'info, TokenAccount>>,
535 
536     /// The vault token account for output token
537     #[account(
538         mut,
539         constraint = output_vault.key() == pool_state.load()?.token_0_vault ||
            output_vault.key() == pool_state.load()?.token_1_vault
540     )]
541     pub output_vault: Box<InterfaceAccount<'info, TokenAccount>>,
542 
543     /// SPL program for input token transfers
544     pub input_token_program: Interface<'info, TokenInterface>,
545 
546     /// SPL program for output token transfers
547     pub output_token_program: Interface<'info, TokenInterface>,
548 
549     /// The mint of input token
550     #[account(
551         address = input_vault.mint
552     )]
553     pub input_token_mint: Box<InterfaceAccount<'info, Mint>>,
554 
555     /// The mint of output token
556     #[account(
557         address = output_vault.mint
558     )]
559     pub output_token_mint: Box<InterfaceAccount<'info, Mint>>,
560 }
```

Listing 3.2: The `swap_base_input::Swap` Structure

Note the `mint` validation issue is applicable to the `output_token_account` account (line 527). In addition, the `deposit.rs` instruction has the same issue with the defined `owner_lp_token` account.

Recommendation Improve the above-mentioned routines with additional validations for involved accounts.

Status The issue has been fixed by this commit: e702514.

3.3 Possible Overflow Risk in lp_tokens_to_trading_tokens()

- ID: PVE-003
- Severity: Low
- Likelihood: low
- Impact: Medium

- Target: `constant_product.rs`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

In `GoatSwap`, when a user wants to provide or withdraw liquidity, the handling logic is implemented in the `lp_tokens_to_trading_tokens()` function. In the process of examining its logic, we notice it makes use of the constant product formula to calculate the amount of underlying tokens for a certain number of pool tokens. And our analysis shows the current implementation has a possible integer overflow that needs to be resolved.

In particular, the arithmetic operation to calculate pool tokens, i.e., `token_0_amount = token_0_amount +1` (line 625), does not make use of the `checked_add` validation. As a result, it may unfortunately lead to an integer overflow if the given `token_0_amount` is extremely large.

```
600     pub fn lp_tokens_to_trading_tokens(
601         lp_token_amount: u128,
602         lp_token_supply: u128,
603         swap_token_0_amount: u128,
604         swap_token_1_amount: u128,
605         round_direction: RoundDirection,
606     ) -> Option<TradingTokenResult> {
607         let mut token_0_amount = lp_token_amount
608             .checked_mul(swap_token_0_amount)?
609             .checked_div(lp_token_supply)?;
610         let mut token_1_amount = lp_token_amount
611             .checked_mul(swap_token_1_amount)?
612             .checked_div(lp_token_supply)?;
613         let (token_0_amount, token_1_amount) = match round_direction {
614             RoundDirection::Floor => (token_0_amount, token_1_amount),
615             RoundDirection::Ceiling => {
616                 let token_0_remainder = lp_token_amount
617                     .checked_mul(swap_token_0_amount)?
618                     .checked_rem(lp_token_supply)?;
619                 // Also check for 0 token A and B amount to avoid taking too much
620                 // for tiny amounts of pool tokens.  For example, if someone asks
621                 // for 1 pool token, which is worth 0.01 token A, we avoid the
622                 // ceiling of taking 1 token A and instead return 0, for it to be
623                 // rejected later in processing.
624                 if token_0_remainder > 0 && token_0_amount > 0 {
625                     token_0_amount += 1;
626                 }
627                 let token_1_remainder = lp_token_amount
```

```
                    .checked_mul(swap_token_1_amount)?
                    .checked_rem(lp_token_supply)?;
                if token_1_remainder > 0 && token_1_amount > 0 {
                    token_1_amount += 1;
                }
                (token_0_amount, token_1_amount)
            }
        };
        Some(TradingTokenResult {
            token_0_amount,
            token_1_amount,
        })
    }
```

Listing 3.3: `constant_product::lp_tokens_to_trading_tokens()`

Recommendation Replacec + with `checked_add` in the above function.

Status The issue has been fixed by this commit: e702514.

PeckShield

3.4 Possible Fake Account in Pool Initialization

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `swap_base_output.rs`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

In `GoatSwap`, the `initialize` instruction can be invoked by anyone to create a new pool. The pool creator may provide the `amm_config` input to define fee-related fields for the new pool. While examining the logic to initialize a new pool, we notice current implementation can be improved.

To elaborate, we show below the definition of the related `Initialize` data structure. The `amm_config` input (lines 706) needs to be properly verified. For example, a trusted entity (e.g., admin) should be involved to ensure the given pool configuration is sound and appropriate.

```
700    pub struct Initialize<’info> {
701    /// Address paying to create the pool. Can be anyone
702    #[account(mut)]
703    pub creator: Signer<’info>,
704
705    /// Which config the pool belongs to.
706    pub amm_config: Box<Account<’info, AmmConfig>>,
707
708    /// CHECK: pool vault and lp mint authority
709    #[account(
710        seeds = [
711            crate::AUTH_SEED.as_bytes(),
712        ],
713        bump,
714    )]
715    pub authority: UncheckedAccount<’info>,
716
717    /// Initialize an account to store the pool state
718    #[account(
719        init,
720        seeds = [
721            POOL_SEED.as_bytes(),
722            amm_config.key().as_ref(),
723            token_0_mint.key().as_ref(),
724            token_1_mint.key().as_ref(),
725        ],
726        bump,
727        payer = creator,
728        space = PoolState::LEN
729    )]
730    pub pool_state: AccountLoader<’info, PoolState>,

    /// Token_0 mint, the key must smaller then token_1 mint.
    #[account(
        constraint = token_0_mint.key() < token_1_mint.key(),
        mint::token_program = token_0_program,
    )]
    pub token_0_mint: Box<InterfaceAccount<’info, Mint>>,

    /// Token_1 mint, the key must grater then token_0 mint.
    #[account(
        mint::token_program = token_1_program,
    )]
    pub token_1_mint: Box<InterfaceAccount<’info, Mint>>,

    /// pool lp mint
    #[account(
        init,
        seeds = [
            POOL_LP_MINT_SEED.as_bytes(),
            pool_state.key().as_ref(),
        ],
        bump,
        mint::decimals = if token_0_mint.decimals >= token_1_mint.decimals{
            token_0_mint.decimals
        }else{
            token_1_mint.decimals
        },
        mint::authority = authority,
        payer = creator,
        mint::token_program = token_program,
    )]
    pub lp_mint: Box<InterfaceAccount<’info, Mint>>,

    /// payer token0 account
    #[account(
        mut,
        token::mint = token_0_mint,
        token::authority = creator,
    )]
    pub creator_token_0: Box<InterfaceAccount<’info, TokenAccount>>,

    /// creator token1 account
    #[account(
        mut,
        token::mint = token_1_mint,
        token::authority = creator,
    )]
    pub creator_token_1: Box<InterfaceAccount<’info, TokenAccount>>,
    ...
    }
```

Listing 3.4: The `initialize::Initialize` Structure

Recommendation Improve the above routine with necessary validation for the given pool configuration.

Status This issue has been confirmed, and the team clarifies that only the admin wallet can update new configuration.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple contracts
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In GoatSwap, there is a privileged account, i.e., admin. This account plays a critical role in governing and regulating the system-wide operations (e.g., create AMM configuration, collect fee etc.). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the create_amm_config instruction as an example and show the representative functions potentially affected by the privileges of the admin account.

```
    pub struct CreateAmmConfig<'info> {
    /// Address to be set as protocol owner.
    #[account(
        mut,
        address = crate::admin::id() @ ErrorCode::InvalidOwner
    )]
    pub owner: Signer<'info>,

    /// Initialize config state account to store protocol owner address and fee rates.
    #[account(
        init,
        seeds = [
            AMM_CONFIG_SEED.as_bytes(),
            &index.to_be_bytes()
        ],
        bump,
        payer = owner,
        space = AmmConfig::LEN
    )]
    pub amm_config: Account<'info, AmmConfig>,

    pub system_program: Program<'info, System>,
}

```

```rust
pub fn create_amm_config(
    ctx: Context<CreateAmmConfig>,
    index: u16,
    trade_fee_rate: u64,
    protocol_fee_rate: u64,
    fund_fee_rate: u64,
    create_pool_fee: u64,
) -> Result<()> {
    let amm_config = ctx.accounts.amm_config.deref_mut();
    amm_config.protocol_owner = ctx.accounts.owner.key();
    amm_config.bump = ctx.bumps.amm_config;
    amm_config.disable_create_pool = false;
    amm_config.index = index;
    amm_config.trade_fee_rate = trade_fee_rate;
    amm_config.protocol_fee_rate = protocol_fee_rate;
    amm_config.fund_fee_rate = fund_fee_rate;
    amm_config.create_pool_fee = create_pool_fee;
    amm_config.fund_owner = ctx.accounts.owner.key();
    Ok(())
}
```

Listing 3.5: `create_config::create_amm_config()`

We understand the need of the privileged functions for proper `GoatSwap` operations, but at the same time the extra power to the `admin` may also be a counter-party risk to the `GoatSwap` contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to `GoatSwap` explicit to `GoatSwap` contract users.

Status This issue has been confirmed.

PeckShield

4 Conclusion

In this audit, we have analyzed the design and implementation of the `GoatSwap` protocol, is a decentralized exchange built on `Solana`. It allows users to trade and swap thousands of `Solana` tokens in realtime with the most liquidity and the best rate. In the meantime, it also allows liquidity providers to create trading pairs and add liquidity in a trustless manner. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

PeckShield

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[3] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[4] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[5] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[6] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[7] PeckShield. PeckShield Inc. https://www.peckshield.com.

PeckShield