SECURITY AUDIT REPORT

for

Abel Finance

Prepared By: Xiaomi Huang

PeckShield

February 9, 2023

Document Properties

Client	Abel
Title	Security Audit Report
Target	Abel
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	February 9, 2023	Daisy Cao	Final Release
1.0-rc	February 8, 2023	Daisy Cao	Release Candidate #1

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

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Abel` 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 Abel

`Abel` aims to be the first cross-chain lending platform on `Aptos` and `Sui`. Through efficient cross-chain lending, `Abel` allows for convenient and reliable transfer of multi-chain assets. Moreover, liquidity providers can deposit tokens as liquidity into the market and then earn interest income from borrowing as well as additional protocol incentives. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Abel

Item	Description
Name	Abel
Website	https://abelfinance.xyz
Type	Aptos
Language	Move
Audit Method	Whitebox
Latest Audit Report	February 9, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that `Abel` assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/abelfinance/contracts (c80c45e3)

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

- https://github.com/abelfinance/contracts (eb2bba56)

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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

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

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.

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 `Abel` implementation. 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	1	■
Undetermined	1	■
Total	5	

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, 1 low-severity vulnerability, and 1 undetermined issue.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Timely Interest Accrual in ACoinLend::redeem_underlying()	Business Logic	Resolved
PVE-002	Medium	Possibly Stale Rewards Without Refresh Upon Setting Changes	Business Logic	Confirmed
PVE-003	Low	Possible Redeem Failure Avoidance	Business Logic	Resolved
PVE-004	Medium	Trust on Admin Keys	Security Features	Mitigated
PVE-005	Undetermined	Potential Protocol Risk from Low-Liquidity Assets	Business Logic	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.

3 | Detailed Results

3.1 Timely Interest Accrual in ACoinLend::redeem_underlying()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `ACoinLend.move`
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

In the `Abel` protocol, the assets supplied to a market are represented by `ACoin`, which entitles the owner to an increasing quantity of the underlying asset. Users can exchange their `ACoin` for the underlying assets using the two functions (`redeem_entry()`, `redeem_underlying()`) via the `ACoinLend` contract. While reviewing their logic, we notice the current implementation needs to be improved.

In the following, we show the related `redeem_underlying()` function that is used to redeem `ACoin` for the underlying assets. The routine makes use of three variables – `total_cash`, `total_borrows`, `total_reserves` – to calculate `exchange_rate_mantissa`. It comes to our attention the used variables may not be up-to-date (line 19). In fact, there is a need to invoke `accrue_interest()` right before calling the `acoin::exchange_rate_mantissa<CoinType>()` (line 85) to ensure their refreshness.

```
10      public fun exchange_rate_mantissa<CoinType>(): u128 acquires ACoinInfo {
11          let supply = total_supply<CoinType>();
12          if (supply == 0) {
13              initial_exchange_rate_mantissa<CoinType>()
14          } else {
15              let supply = total_supply<CoinType>();
16              let total_cash = (get_cash<CoinType>() as u128);
17              let total_borrows = total_borrows<CoinType>();
18              let total_reserves = total_reserves<CoinType>();
19              let cash_plus_borrows_minus_reserves = total_cash + total_borrows -
                    total_reserves;
20              cash_plus_borrows_minus_reserves * Exp_Scale() / supply
21          }
```

```
    }
```

Listing 3.1: `ACoin::exchange_rate_mantissa()`

```
    public entry fun redeem_underlying<CoinType>(
        redeemer: &signer,
        redeem_amount: u64,
    ) {
        let redeemer_addr = signer::address_of(redeemer);
        let exchange_rate_mantissa = acoin::exchange_rate_mantissa<CoinType>();
        let redeem_tokens = ((redeem_amount as u128) * Exp_Scale() /
            exchange_rate_mantissa as u64);
        let acoin = withdraw<CoinType>(redeemer, redeem_tokens);
        let coin = redeem<CoinType>(redeemer, acoin);
        coin::deposit<CoinType>(redeemer_addr, coin);
    }
```

Listing 3.2: `ACoinLend::redeem_underlying()`

Recommendation Add `accrue_interest()` before calling `exchange_rate_mantissa()` in the above function `redeem_underlying()`.

Status This issue has been addressed by the following commit: `61a1766f`.

3.2 Possibly Stale Rewards Without Refresh Upon Setting Changes

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `stake.move`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The the `Abel` protocol provides a number of routines to allow for dynamic configuration of protocol rewards. For example, the `stake` contract provides staking pools for users to stake tokens for rewards. The reward calculation depends on a number of key parameters, e.g., `abel_per_block` and `alloc_point`. If there is a desire to update them, there is always a need to compute the latest reward before they can be changed.

For example, the function `set_abel_per_block()` updates the `abel_per_block` key parameter (line 405), which configures the emission of `Abel` rewards to active stakers. However, it comes to our attention the current implementation does not timely update `acc_abel_per_share`, which may result in an inaccurate reward calculation for current protocol users.

```
    public entry fun set_abel_per_block(admin: &signer, abel_per_block: u64) acquires StakeStore {
        only_admin(admin);

        try_init_global(admin);
        let global_store = borrow_global_mut<StakeStore>(admin());
        global_store.abel_per_block = abel_per_block;

        event::emit_event<SetAbelPerBlockEvent>(
            &mut global_store.set_abel_per_block_events,
            SetAbelPerBlockEvent {
                abel_per_block,
            },
        );
    }
```

Listing 3.3: `stake::set_abel_per_block()`

Note that other routines share the same issue, including `set_abel_per_block`, `set_reserve_factor_mantissa`, and `set_abel_rate`.

Recommendation Refresh the reward calculation right before applying changes in the above-mentioned functions.

Status This issue has been confirmed by the team.

3.3 Possible Redeem Failure Avoidance

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `ACoinLend.move`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

In the `Aptos` chain, users must actively and explicitly register to receive a token other than the default `APT` coin. The design principle here is to ensure that no tokens will be randomly airdropped. While reviewing the `Abel` protocol, we notice the `redeem_entry()` implementation can be improved.

To elaborate, we show below the related code snippet of the `redeem_entry()` routine. This routine is used to redeem the underlying assets with `ACoin`. However, there is no validation on whether the redeemer's account has been registered for the underlying assets.

```
    public entry fun redeem_entry<CoinType>(
        redeemer: &signer,
        redeem_tokens: u64,
```

```
    ) {
        let redeemer_addr = signer::address_of(redeemer);
        let acoin = withdraw<CoinType>(redeemer, redeem_tokens);
        let coin = redeem<CoinType>(redeemer, acoin);
        coin::deposit<CoinType>(redeemer_addr, coin);
    }
```

Listing 3.4: `ACoinLend::redeem_entry()`

Note that the same issue is also applicable to the `redeem_underlying()` routine.

Recommendation Add necessary validation to ensure the account is registered for `CoinType` in the above-mentioned functions.

Status This issue has been addressed by the following commit: `78f16b92`.

3.4 Trust on Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `Abel` protocol, there is a privileged account, i.e., `@abel`. This account plays a critical role in regulating the protocol-wide operations (e.g., approve/drop a certain market). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `market_storage` contract as an example and show the representative functions potentially affected by the privileges of the `@abel` account.

```
    public(friend) fun approve_market<CoinType>(admin: &signer) acquires MarketConfig {
        assert!(signer::address_of(admin) == admin(), ENOT_ADMIN);
        assert!(!is_approved<CoinType>(), EALREADY_APPROVED);
        if (!exists<GlobalConfig>(admin())) {
            init(admin);
        };
        move_to(admin, MarketConfig<CoinType> {
            is_approved: true,
            is_listed: false,
            mint_guardian_paused: false,
            borrow_guardian_paused: false,
            market_action_paused_events: account::new_event_handle<
                MarketActionPausedEvent>(admin),
            new_collateral_factor_events: account::new_event_handle<
                NewCollateralFactorEvent>(admin),
```

```
        });
    }
```

Listing 3.5: market_storage.move

We understand the need of the privileged functions for proper operations, but at the same time the extra power to the @abel may also be a counter-party risk to the Abel 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 @abel explicit to Abel users.

Status This issue has been confirmed.

3.5 Potential Protocol Risk from Low-Liquidity Assets

- ID: PVE-005
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: Abel Protocol
- Category: Business Logic [4]
- CWE subcategory: CWE-837 [2]

Description

With the occurrence of the Mango Market Incident on Solana, the risk of liquidity attacks on lending platforms attracts much attention from the entire DeFi community. In the following, we give one possible vector to illustrate the risk. A malicious actor may borrow a large amount of the available supply of a token (such as ZRX) from the lending market and sell it across multiple centralized and decentralized exchanges to depress the open market price. Once the price oracle of the lending market is updated with a lower price, the malicious actor may then withdraw most of the original collateral.

As an example, the malicious actor supplies $30M stablecoins as collateral firstly (Step I), secondly borrows $20M illiquid token (Step II), next sells it to depress the token's market price by 95% and realizes $7.5M (Step III), and finally withdraws $28M collateral with the user's debt going down to $1M (Step IV). Overall, the malicious actor profits $5.5M leaving lending market with bad debt.

Our suggest that the Abel protocol pay attention to this kind of low liquidity assets, which expose the similar market manipulation risk.

Recommendation Evaluate the current set of assets supported in Abel and revisit possible risks from low-liquidity ones to avoid the above market manipulation risk.

Status This issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the `Abel` design and implementation. 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.

References

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

[2] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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.