SMART CONTRACT AUDIT REPORT

for

Formless

Prepared By: Patrick Lou

PeckShield

March 10, 2022

Document Properties

Client	Formless
Title	Smart Contract Audit Report
Target	Formless
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 10, 2022	Jing Wang	Final Release
1.0-rc1	March 10, 2022	Jing Wang	Release Candidate #1

Contact

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

Name	Patrick Lou
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 Formless 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 Formless

The Formless protocol is an on-chain platform for conducting IDO that allows users to raise desired tokens. Investors could swap the accepted tokens to the IDO tokens and depending on the level of the investors, the user could invest different amount of tokens. This project also has an IDOLaunchpad contract that allows new instances of the Pool contract to be deployed. The basic information of the Formless protocol is as follows:

Table 1.1: Basic Information of The Formless Protocol

Item	Description
Name	Formless
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 10, 2022

In the following, we show the MD5 hash value of the compressed file used in this audit:

- MD5 (formless_launchpad_contracts.zip) = 327ec996dbbc4c7e91226cd13c5950df

And here is the final MD5 hash value of the compressed file after all fixes for the issues found in the audit have been checked in:

- MD5 (formless_launchpad_contracts.zip) = ff5345c5f3b2ee5fc5182f93ec5f59fc

1.2 About PeckShield

PeckShield Inc. [9] 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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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 classified into four categories accordingly, 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) [7], 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.

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 `Formless` 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	0	
Low	3	■ ■ ■
Informational	1	■
Total	4	

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 that 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 low-severity vulnerabilities and 1 informational recommendation.

Table 2.1: Key Formless Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic of Amount Calculation In withdraw()	Business Logic	Confirmed
PVE-002	Informational	Generation of Meaningful Events For Important State Changes	Coding Practices	Fixed
PVE-003	Low	Incompatibility With Deflationary Tokens	Business Logic	Confirmed
PVE-004	Low	Reentrancy Risk in Pool::invest()	Time and State	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Logic of Amount Calculation In withdraw()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Pool
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned before, the Formless protocol allows users to invest ERC20 tokens to swap tokens provided by the IDO creator. The creator can withdraw the raised amount any time, and the left amounts remain on the Pool contract. While examining the related logic, we notice an issue in current implementation. To elaborate, we show below the handling withdraw() routine.

```
    function withdraw(address payable toAddr, uint256 exact)
    external
    onlyOwner
    nonZeroAddress(toAddr)
    returns (bool)
    {
        uint256 amount;
        if (exact == 0) {
            amount = _raisedBalance;
            _raisedBalance = 0;
        } else {
            amount = exact;
            if (_raisedBalance > exact) {
                _raisedBalance = _raisedBalance.sub(exact);
            } else {
                _raisedBalance = 0;
            }
        }

        if (acceptingTokenContractIsSet()) {
            // withdraw ERC20 token
```

```
            _acceptingTokenContract.safeTransfer(toAddr, amount);
        } else {
            // withdraw native token
            toAddr.transfer(amount);
        }

        emit Withdraw(block.timestamp, _poolId, toAddr, amount, true);
        return true;
    }
```

Listing 3.1: `Pool::withdraw()`

The `withdraw()` routine implements a rather straightforward logic in allowing the creator to withdraw `ERC20` tokens into this contract and deduct the value of `amount` from `_raisedBalance` each time. However, we notice the `amount` is not properly adjusted in the case of the investor giving a larger input value of `exact` than `_raisedBalance`. In this case, the investor is trying to withdraw a larger amount of tokens than the `IDO` has raised which will cause the transaction revert.

Recommendation Improve the calculation logic for the `amount` variable inside the `withdraw()` routine.

Status The issue has been confirmed by the team. And the team clarifies the current implementation is complied with design and allowing specifying `exact` is a failsafe for worst situation when real balance is greater than `_rasiedBalance`.

3.2 Generation of Meaningful Events For Important State Changes

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Pool`
- Category: Coding Practices [4]
- CWE subcategory: CWE-1126 [1]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `Pool` contract as an example. This contract has a privileged function `initialize()` that is used to configure several important parameters. While examining the events that reflect these parameters changes, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the `_supplyOnInvestorLevels`, `_whitelistOnInvestorLevels`, `_tokenDecimal`, `_acceptingTokenContract` and `_acceptingTokenDecimal` are being updated in `Pool::initialize()`, there are no respective events being emitted to reflect their updates (lines 180-184).

```
162     function initialize(
163         uint256[] memory supplyOnInvestorLevels_,
164         bytes32[] memory whitelistOnInvestorLevels_,
165         uint256 tokenDecimal_,
166         address acceptingTokenContract_,
167         uint256 acceptingTokenDecimal_
168     )
169     external
170     onlyOwnerOrOperator
171     poolIsUpcoming
172     returns (bool)
173     {
174         require(supplyOnInvestorLevels_.length > 0, "Supply list is empty");
175         require(whitelistOnInvestorLevels_.length > 0, "Merkle root hash list is empty")
            ;
176         require(tokenDecimal_ > 0, "Token decimal must be greater than 0");
177         require(10 ** tokenDecimal_ >= TOKEN_MIN_INVESTING, "Token decimal is less than
            TOKEN_MIN_INVESTING");
178         require(acceptingTokenDecimal_ > 0, "Accepting Token decimal must be greater
            than 0");

180         _supplyOnInvestorLevels = supplyOnInvestorLevels_;
181         _whitelistOnInvestorLevels = whitelistOnInvestorLevels_;
182         _tokenDecimal = tokenDecimal_;
183         _acceptingTokenContract = IERC20(acceptingTokenContract_);
184         _acceptingTokenDecimal = acceptingTokenDecimal_;

186         initialized = true;
187         return true;
188     }
```

Listing 3.2: `Pool::initialize()`

Recommendation Properly emit respective events when important states are updated.

Status The issue has been fixed.

3.3 Incompatibility With Deflationary Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Pool`
- Category: Business Logic [5]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.1, in the `Formless` protocol, the `Pool` contract allows users to invest `ERC20` tokens to get another kind of tokens. The investor can withdraw amounts that have been invested at any time.

In particular, one interface, i.e., `invest()`, accepts asset transfer-in and records the raised balance. Another interface, i.e, `withdraw()`, allows the user to withdraw the asset. For the above two operations, i.e., `invest()` and `withdraw()`, the contract makes the use of `safeTransferFrom()` or `safeTransfer()` routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
function invest(bytes32[] memory proof, uint256 value)
external
payable
poolIsInitialized
poolIsOnGoing
poolIsNotPaused
returns (bool)
{
    ...

    if (acceptingTokenContractIsSet()) {
        // accepting ERC20 token, approve needed
        _acceptingTokenContract.safeTransferFrom(msg.sender, address(this), spending);
    }

    _raised = _raised.add(spending);
    _raisedBalance = _raisedBalance.add(spending);
    _balanceOf[msg.sender] = _balanceOf[msg.sender].add(spending);

    _tokenContract.safeTransferFrom(_tokenWallet, msg.sender, tokenValue);

    emit Invest(block.timestamp, _poolId, msg.sender, spending, tokenValue, true);
    return true;
}
```

Listing 3.3: `Pool::invest()`

```
function withdraw(address payable toAddr, uint256 exact)
external
onlyOwner
nonZeroAddress(toAddr)
returns (bool)
{
    ...

    if (acceptingTokenContractIsSet()) {
        // withdraw ERC20 token
        _acceptingTokenContract.safeTransfer(toAddr, amount);
    } else {
        // withdraw native token
        toAddr.transfer(amount);
    }

    emit Withdraw(block.timestamp, _poolId, toAddr, amount, true);
    return true;
}
```

Listing 3.4: `Pool::withdraw()`

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer. As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `invest()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the contract and affects protocol-wide operation and maintenance.

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary. Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Pool` for support.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate.

Status This issue has been confirmed. And the team clarifies that the protocol will not support deflationary tokens.

3.4 Reentrancy Risk in Pool::invest()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Pool
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [11] exploit, and the recent Uniswap/Lendf.Me hack [10].

We notice there is an occasion where the checks-effects-interactions principle is violated. Using the Pool as an example, the invest() function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy.

Apparently, the interaction with the external contract (line 57) starts before effecting the update on the internal state (lines 60-62), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```
45      function invest(bytes32[] memory proof, uint256 value)
46      external
47      payable
48      poolIsInitialized
49      poolIsOnGoing
50      poolIsNotPaused
51      returns (bool)
52      {
53          ...
54
55          if (acceptingTokenContractIsSet()) {
56              // accepting ERC20 token, approve needed
57              _acceptingTokenContract.safeTransferFrom(msg.sender, address(this), spending
                    );
58          }
59
60          _raised = _raised.add(spending);
61          _raisedBalance = _raisedBalance.add(spending);
```

```
        _balanceOf[msg.sender] = _balanceOf[msg.sender].add(spending);

        _tokenContract.safeTransferFrom(_tokenWallet, msg.sender, tokenValue);

        emit Invest(block.timestamp, _poolId, msg.sender, spending, tokenValue, true);
        return true;
    }
```

Listing 3.5: `Pool::invest()`

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been confirmed. The team clarifies reentrancy attack in this case would make no harm to contract and no benefits to the attacker.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Formless` protocol, which implements an on-chain `IDO` project that allows investors to swap tokens with the `IDO` creator. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that `Solidity`-based 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

[4] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

[6] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[11] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.