SMART CONTRACT AUDIT REPORT

for

RALLY NETWORK

Prepared By: Shuxiao Wang

Hangzhou, China
October 30, 2020

Document Properties

Client	Rally Network
Title	Smart Contract Audit Report
Target	Yield Delegating Vaults
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi, Jeff Liu
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 30, 2020	Xuxian Jiang	Final Release
1.0-rc	October 22, 2020	Xuxian Jiang	Release Candidate
0.2	October 18, 2020	Xuxian Jiang	Additional Findings
0.1	October 15, 2020	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction **5**

1.1 About Yield Delegating Vaults 5

1.2 About PeckShield 6

1.3 Methodology 6

1.4 Disclaimer 8

2 Findings **10**

2.1 Summary 10

2.2 Key Findings 11

3 Detailed Results **12**

3.1 Potential Reentrancy Risk in YieldDelegatingVault 12

3.2 The Mismatched Addition of Amounts Denominated in Different Tokens 14

3.3 Inconsistent Enforcement of Individual/Global DepositCap 15

3.4 Inaccurate yToken Conversion in depositYToken() 16

3.5 Improved Precision in YieldDelegatingVault::harvest() 18

3.6 Incompatibility with Deflationary/Rebasing Tokens 19

3.7 Duplicate Pool Detection and Prevention 21

3.8 Recommended Explicit Pool Validity Checks 22

3.9 Suggested Adherence of Checks-Effects-Interactions 25

3.10 Timely massUpdatePools During Pool Weight Changes 27

4 Conclusion **29**

5 Appendix **30**

5.1 Basic Coding Bugs 30

5.1.1 Constructor Mismatch 30

5.1.2 Ownership Takeover 30

5.1.3 Redundant Fallback Function 30

5.1.4 Overflows & Underflows 30

5.1.5 Reentrancy . . . 31
5.1.6 Money-Giving Bug . . . 31
5.1.7 Blackhole . . . 31
5.1.8 Unauthorized Self-Destruct . . . 31
5.1.9 Revert DoS . . . 31
5.1.10 Unchecked External Call . . . 32
5.1.11 Gasless Send . . . 32
5.1.12 Send Instead Of Transfer . . . 32
5.1.13 Costly Loop . . . 32
5.1.14 (Unsafe) Use Of Untrusted Libraries . . . 32
5.1.15 (Unsafe) Use Of Predictable Variables . . . 33
5.1.16 Transaction Ordering Dependence . . . 33
5.1.17 Deprecated Uses . . . 33
5.2 Semantic Consistency Checks . . . 33
5.3 Additional Recommendations . . . 33
5.3.1 Avoid Use of Variadic Byte Array . . . 33
5.3.2 Make Visibility Level Explicit . . . 34
5.3.3 Make Type Inference Explicit . . . 34
5.3.4 Adhere To Function Declaration Strictly . . . 34

References **35**

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of Yield Delegating Vaults, we in the report outline 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 Yield Delegating Vaults

Yield Delegating Vaults is a modular protocol that is heavily influenced by `YFI`. In essence, the protocol is a wrapper around these `YFI` vaults and allows users to delegate the earnings generated on their behalf to a beneficiary. In short, contributors to these yield delegating vaults would expect their principal to be just as secure as if they deposited into a YFI vault directly but they would allow their profit to flow to a designated treasury supporting various projects instead of accruing to themselves. A reward token related to the project associated with the designated treasury would be emitted in lieu of the delegated yield.

The basic information of the Yield Delegating Vaults protocol is as follows:

Table 1.1: Basic Information of Yield Delegating Vaults

Item	Description
Issuer	Rally Network
Website	http://www.rally.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 30, 2020

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

- https://github.com/starcard-org/yield-delegation/tree/peckshield-audit (e2b9200)

1.2 About PeckShield

PeckShield Inc. [18] 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 [13]:

- 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.

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

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 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) [12], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 design and implementation of the Yield Delegating Vaults protocol. 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	1	■
Medium	2	■ ■
Low	4	■ ■ ■ ■
Informational	3	■ ■ ■
Total	10	

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 1 high-severity vulnerability, 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key Audit Findings of Yield Delegating Vaults

ID	Severity	Title	Category	Status
PVE-001	Low	Potential Reentrancy Risk in YieldDelegatingVault	Time and State	Fixed
PVE-002	Medium	The Mismatched Addition of Amounts Denominated in Different Tokens	Business Logics	Fixed
PVE-003	Medium	Inconsistent Enforcement of Individual/Global DepositCap	Security Features	Fixed
PVE-004	High	Inaccurate yToken Conversion in depositYToken()	Business Logics	Fixed
PVE-005	Informational	Improved Precision in YieldDelegatingVault::harvest()	Numeric Errors	Confirmed
PVE-006	Informational	Incompatibility with Deflationary/Rebasing Tokens	Business Logics	Confirmed
PVE-007	Low	Duplicate Pool Detection and Prevention	Business Logics	Confirmed
PVE-008	Informational	Recommended Explicit Pool Validity Checks	Security Features	Confirmed
PVE-009	Low	Suggested Adherence of Checks-Effects-Interactions	Time and State	Confirmed
PVE-010	Low	Timely massUpdatePools During Pool Weight Changes	Business Logics	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 Potential Reentrancy Risk in YieldDelegatingVault

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `YieldDelegatingVault`
- Category: Time and State [10]
- CWE subcategory: CWE-682 [5]

Description

The Yield Delegating Vaults protocol is heavily influenced by `YFI` and architecturally is a wrapper around these `YFI` vaults. By doing so, it allows users to delegate the earnings generated on their behalf to a beneficiary. The `YieldDelegatingVault` keeps track of user funds within an ever-growing `YFI` vault which is expected to grow with additional gains.

At the core, it leverages a `YFI`-specific `strategy` to invest users' funds (transferred from the linked `YFI vault`) into certain yield-gaining pools. While reviewing the current `YieldDelegatingVault` contract, we notice there is a potential reentrancy risk in current implementation.

To elaborate, we show below the code snippet of the `deposit()` routine in `YieldDelegatingVault`. The execution logic is rather straightforward: it firstly transfers the funds from the depositing user to the delegating vault, and then mints corresponding shares to the user.

```
    function deposit(uint256 _amount) public returns (uint256) {
        if(individualDepositCap < balanceOf(address(this)).add(_amount)) {
            return fail(Error.BAD_INPUT, FailureInfo.SET_INDIVIDUAL_SOFT_CAP_CHECK);
        }

        if(globalDepositCap < totalSupply().add(_amount)) {
            return fail(Error.BAD_INPUT, FailureInfo.SET_GLOBAL_SOFT_CAP_CHECK);
        }

        uint256 pending = earned(msg.sender);
        if (pending > 0) {
            safeRallyTransfer(msg.sender, pending);
```

```
100         }
101         uint256 _pool = balance();
102 
103         uint256 _before = token.balanceOf(address(this));
104         token.safeTransferFrom(msg.sender, address(this), _amount);
105         uint256 _after = token.balanceOf(address(this));
106         _amount = _after.sub(_before);
107 
108         totalDeposits = totalDeposits.add(_amount);
109 
110         token.approve(vault, _amount);
111         Vault(vault).deposit(_amount);
112         uint256 _after_pool = balance();
113 
114         uint256 _new_shares = _after_pool.sub(_pool); //new vault tokens representing my added vault shares
115 
116         //translate vault shares into delegating vault shares
117         uint256 shares = 0;
118         if (totalSupply() == 0) {
119             shares = _new_shares;
120         } else {
121             shares = (_new_shares.mul(totalSupply())).div(_pool);
122         }
123         _mint(msg.sender, shares);
124         rewardDebt[msg.sender] = balanceOf(msg.sender).mul(accRallyPerShare).div(1e12);
125     }
```

Listing 3.1: YieldDelegatingVault.sol

However, our analysis shows that the current implementation of `deposit()` lacks re-entrancy prevention. If the underlying token faithfully implements the ERC777-like standard, then the `deposit()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend` and `tokensReceived` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In the ERC777 token case, it may be exploited to manipulate the number of shares that are calculated. Specifically, the above hook can be planted in `token.safeTransferFrom(msg.sender, address(this), _amount)` (line 104 or 111) before the actual transfer of the underlying token occurs. By doing so, we can effectively keep `balance()` intact (used for the calculation of minted shares at line 101). With a lower `_pool`, the re-entered `deposit()` is able to mint more underlying tokens. It can be repeated to exploit this vulnerability for gains, just like earlier Uniswap/imBTC hack [19].

We emphasize that those tokens supported in `YFI` are not ERC777-compliant, which means the current implementation is safe. However, with the possibility of adding new tokens, it is always our suggestion to be proactive in filtering out unsupported tokens in the first place.

Recommendation Add necessary reentrancy guards (e.g., `nonReentrant`) to prevent unwanted reentrancy risks.

Status The issue has been fixed by adding necessary `nonReentrant` in this commit: `ebdbcea`.

3.2 The Mismatched Addition of Amounts Denominated in Different Tokens

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `YieldDelegatingVault`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [6]

Description

The Yield Delegating Vaults protocol has defined a number of system parameters to regulate the protocol behavior. Two example parameters are `globalDepositCap` and `individualDepositCap`. As the names indicate, the first parameter specifies the global deposit threshold that effectively limits the total deposit amount allowed into the protocol; and the second parameter provides a more fine-grained threshold at the individual user level.

To elaborate, we show below the related enforcement logic of these two system parameters.

```
88      function deposit(uint256 _amount) public returns (uint256) {
89          if(individualDepositCap < balanceOf(address(this)).add(_amount)) {
90              return fail(Error.BAD_INPUT, FailureInfo.SET_INDIVIDUAL_SOFT_CAP_CHECK);
91          }
92
93          if(globalDepositCap < totalSupply().add(_amount)) {
94              return fail(Error.BAD_INPUT, FailureInfo.SET_GLOBAL_SOFT_CAP_CHECK);
95          }
96          ...
97      }
```

Listing 3.2: YieldDelegatingVault.sol

We notice that the system parameter `globalDepositCap` is enforced via the following statement: `globalDepositCap < totalSupply().add(_amount)` (line 93). For each specific deposit, it ensures the new deposit amount plus the current `totalSupply()` will not exceed `globalDepositCap`. However, the deposit amount is denominated at the deposit token that will be relayed to the `YFI vault`, and the `totalSupply()` is denominated at its own token, which is different from the deposit token. The addition of these two does not make any sense. The enforcement of `individualDepositCap` shares a very similar issue.

Recommendation Revise the current validation logic to ensure both system parameters, i.e., `globalDepositCap` and `individualDepositCap`, are enforced with denomination at the same deposit token.

Status The issue has been fixed by removing these two parameters in this commit: `ebdbcea`.

3.3 Inconsistent Enforcement of Individual/Global DepositCap

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `YieldDelegatingVault`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [6]

Description

In Section 3.2, we have elaborated two specific system-wide parameters, i.e., `globalDepositCap` and `individualDepositCap`. In this section, we further explore the enforcement logic of these two parameters. As discussed earlier, the `globalDepositCap` parameter specifies the global deposit threshold allowed into the protocol; while the `individualDepositCap` parameter provides a more fine-grained threshold at the individual user level.

Users can stake their assets into Yield Delegating Vaults via two functions, i.e., `deposit()` and `deposityToken()`. The first function directly moves the depositing assets from the staking user to the protocol while the second one indirectly moves the depositing assets by transferring the `YFI vault` token. The analysis in Section 3.2 shows these two parameters are indeed evaluated at `deposit()`. However, the evaluation of these two parameters is currently missing in `deposityToken()`.

```
    function deposityToken(uint256 _yamount) public returns (uint256) {
        uint256 _pool = balance();

        uint256 pending = earned(msg.sender);
        if (pending > 0) {
            safeRallyTransfer(msg.sender, pending);
        }

        uint256 _before = IERC20(vault).balanceOf(address(this));
```

```
        IERC20(vault).safeTransferFrom(msg.sender, address(this), _yamount);
        uint256 _after = IERC20(vault).balanceOf(address(this));
        _yamount = _after.sub(_before);

        uint _underlyingAmount = _yamount.div(Vault(vault).getPricePerFullShare());
        totalDeposits = totalDeposits.add(_underlyingAmount);

        //translate vault shares into delegating vault shares
        uint256 shares = 0;
        if (totalSupply() == 0) {
            shares = _yamount;
        } else {
            shares = (_yamount.mul(totalSupply())).div(_pool);
        }
        _mint(msg.sender, shares);
        rewardDebt[msg.sender] = balanceOf(msg.sender).mul(accRallyPerShare).div(1e12);
    }
```

Listing 3.3: YieldDelegatingVault.sol

Recommendation Add necessary validation logic in `depositToken()` to properly enforce both individual and global deposit thresholds.

Status The issue has been fixed by removing these two parameters in this commit: `ebdbcea`.

3.4 Inaccurate yToken Conversion in depositToken()

- ID: PVE-004
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `YieldDelegatingVault`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [6]

Description

As mentioned in Section 3.3, users can stake their assets into Yield Delegating Vaults via two functions, i.e., `deposit()` and `depositToken()`. The first function directly moves the depositing assets from the staking user to the protocol while the second one indirectly moves the depositing assets by transferring the `YFI vault` token. In either case, there is an internal state `totalDeposits` that is designed to keep track of the total deposits so far.

To elaborate, we show below the `depositToken()` routine. Our analysis shows that the amount conversion from the deposited `yToken` to the underlying token could lead to an inaccurate calculation of `totalDeposits`. Specifically, for the deposited `_yamount` of `yToken`, the corresponding amount of the underlying token is calculated as follows: `_underlyingAmount = _yamount.div(Vault(vault).getPricePerFullShare())` (line 140).

```
127     function depositToken(uint256 _yamount) public returns (uint256) {
128         uint256 _pool = balance();
129 
130         uint256 pending = earned(msg.sender);
131         if (pending > 0) {
132             safeRallyTransfer(msg.sender, pending);
133         }
134 
135         uint256 _before = IERC20(vault).balanceOf(address(this));
136         IERC20(vault).safeTransferFrom(msg.sender, address(this), _yamount);
137         uint256 _after = IERC20(vault).balanceOf(address(this));
138         _yamount = _after.sub(_before);
139 
140         uint _underlyingAmount = _yamount.div(Vault(vault).getPricePerFullShare());
141         totalDeposits = totalDeposits.add(_underlyingAmount);
142 
143         //translate vault shares into delegating vault shares
144         uint256 shares = 0;
145         if (totalSupply() == 0) {
146             shares = _yamount;
147         } else {
148             shares = (_yamount.mul(totalSupply())).div(_pool);
149         }
150         _mint(msg.sender, shares);
151         rewardDebt[msg.sender] = balanceOf(msg.sender).mul(accRallyPerShare).div(1e12);
152     }
```

Listing 3.4: YieldDelegatingVault.sol

This calculation makes a wrong reverse interpretation of the meaning of `Vault(vault).getPricePerFullShare()`. In fact, the proper conversion should be `_underlyingAmount = _yamount.mul(Vault(vault).getPricePerFullShare()).div(1e18)`.

Recommendation Correct the inaccurate calculation of `_underlyingAmount`. Note that an inaccurate conversion could mess up the internal state of `totalDeposits` and cascadingly corrupt both calculations on `availableYield()` and `harvest()`.

Status The issue has been fixed by this commit: `ebdbcea`.

3.5 Improved Precision in YieldDelegatingVault::harvest()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `YieldDelegatingVault`
- Category: Numeric Errors [11]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

Specifically, the Yield Delegating Vaults protocol implements an incentive mechanism that distributes its own `RLY` tokens based on the yields calculated in `harvest()`. To elaborate, we show below the related `harvest()` routine. The harvested reward amount is calculated as `rallyReward = _availableYield.mul(delegatePercent).div(10000).mul(rewardPerToken).div(1e18)` (line 226).

```
220     //transfer accumulated yield to treasury, update totalDeposits to ensure
            availableYield following
221     //harvest is 0, and increase accumulated rally rewards
222     //harvest fails if we're unable to fund rewards
223     function harvest() public {
224         uint256 _availableYield = availableYield();
225         if (_availableYield > 0) {
226             uint256 rallyReward = _availableYield.mul(delegatePercent).div(10000).mul(
                    rewardPerToken).div(1e18);
227             rewards.transferReward(rallyReward);
228             IERC20(vault).safeTransfer(treasury, _availableYield.mul(delegatePercent).
                    div(10000));
229             accRallyPerShare = accRallyPerShare.add(rallyReward.mul(1e12).div(
                    totalSupply()));
230             totalDeposits = balance().mul(Vault(vault).getPricePerFullShare()).div(1e18)
                    ;
231         }
232     }
```

Listing 3.5: YieldDelegatingVault.sol

We notice the above calculation of the reward amount of `rallyReward` involves two multiplications and two devisions. For improved precision, it is better to calculate the multiplication before the division, i.e., `rallyReward = _availableYield.mul(delegatePercent).mul(rewardPerToken).div(1e22)` (line

226). Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been confirmed.

3.6 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `YieldDelegatingVault/YFI`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [6]

Description

As mentioned earlier, the Yield Delegating Vaults protocol wraps the `YFI` vaults and thus shares similar restrictions on the set of tokens that can be supported. In `YFI`, the `yVault` contract is designed to be the main entry for interaction with farming users. In particular, one entry routine, i.e., `deposit()`, accepts user deposits of supported assets (e.g., `DAI`). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the vault. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function earn() public {
        uint256 _bal = available();
        token.safeTransfer(controller, _bal);
        IController(controller).earn(address(token), _bal);
    }

    function depositAll() external {
        deposit(token.balanceOf(msg.sender));
    }

    function deposit(uint256 _amount) public {
        uint256 _pool = balance();
        uint256 _before = token.balanceOf(address(this));
        token.safeTransferFrom(msg.sender, address(this), _amount);
        uint256 _after = token.balanceOf(address(this));
        _amount = _after.sub(_before); // Additional check for deflationary tokens
        uint256 shares = 0;
        if (totalSupply() == 0) {
            shares = _amount;
```

```
        } else {
            shares = (_amount.mul(totalSupply())).div(_pool);
        }
        _mint(msg.sender, shares);
    }
```

Listing 3.6: yVault.sol

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every transfer() or transferFrom(). (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines.

Note that the deposit() routine in yVault is enhanced to properly support deflationary tokens. However, other functions are not! For example, the earn() routine transfers the assets from the current vault to the controller and the amount involved in transfer() has not been properly adjusted for the support of deflationary tokens.

Therefore, we consider there is a lack of support in terms of deflationary/rebasing tokens in current YFI protocol. With that, we need to make it explicit that the Yield Delegating Vaults protocol does not support deflationary/rebasing tokens either.

One possible mitigation is to regulate the set of ERC20 tokens that are permitted into the Yield Delegating Vaults. In our case, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., USDT) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the transfer()/transferFrom() call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is widely-adopted USDT.

Status This issue has been confirmed. However, considering the fact that this specific issue does not affect the normal operation, the team decides to address it when the need of supporting deflationary/rebasing tokens arises.

3.7 Duplicate Pool Detection and Prevention

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `NoMintLiquidityRewardPools`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [6]

Description

The Yield Delegating Vaults protocol provides incentive mechanisms that reward the staking of supported assets with `RLY` tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its `allocPoint*100%/totalAllocPoint` share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded `RLY` tokens and more can be scheduled for addition (via a proper governance procedure). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `add()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accRallyPerShare: 0
        }));
    }
```

Listing 3.7: NoMintLiquidityRewardPools.sol

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IERC20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    function add(uint256 _allocPoint, IERC20 _lpToken, bool _withUpdate) public
        onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock = block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(PoolInfo({
            lpToken: _lpToken,
            allocPoint: _allocPoint,
            lastRewardBlock: lastRewardBlock,
            accRallyPerShare: 0
        }));
    }
```

Listing 3.8: NoMintLiquidityRewardPools.sol (revised)

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

Status The issue has been confirmed. Given that the reward pools contract had been deployed, the team agrees to exercise care in adding or updating pools instead of deploying a new contract and asking users to migrate deposits.

3.8 Recommended Explicit Pool Validity Checks

- ID: PVE-008
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `NoMintLiquidityRewardPools`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [3]

Description

The reward mechanism in Yield Delegating Vaults has a central contract – `NoMintLiquidityRewardPools` that has been tasked with the pool management, staking/unstaking support, as well as the reward distribution to various pools and stakers. In the following, we show the key `pool` data structure. Note all added pools are maintained in an array `poolInfo`.

```
    // Info of each pool.
    struct PoolInfo {
        IERC20 lpToken;           // Address of LP token contract.
        uint256 allocPoint;       // How many allocation points assigned to this pool.
            RLYs to distribute per block.
        uint256 lastRewardBlock;  // Last block number that RLYs distribution occurs.
        uint256 accRallyPerShare; // Accumulated RLYs per share, times 1e12. See below.
    }
    ...
    // Info of each pool.
    PoolInfo[] public poolInfo;
```

Listing 3.9: NoMintLiquidityRewardPools.sol

When there is a need to add a new pool, set a new `allocPoint` for an existing pool, stake (by depositing the supported assets), unstake (by redeeming previously deposited assets), query pending `RLY` rewards, there is a constant need to perform sanity checks on the pool validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the pool index stays within the array range `[0, poolInfo.length-1]`. However, considering the importance of validating given pools and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say `validatePool`. This new modifier essentially ensures the given`_pool_id` or `_pid` indeed points to a valid, live pool, and additionally give semantically meaningful information when it is not!

```
    // Deposit LP tokens to pool for RLY allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accRallyPerShare).div(1e12).sub(user.
                rewardDebt);
            if(pending > 0) {
                safeRallyTransfer(msg.sender, pending);
            }
        }
        if(_amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accRallyPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.10: NoMintLiquidityRewardPools.sol

We highlight that there are a number of functions that can be benefited from the new pool-validating modifier, including `set()`, `deposit()`, `withdraw()`, `emergencyWithdraw()`, `pendingRally()` and `updatePool()`.

Recommendation Apply necessary sanity checks to ensure the given `_pid` is legitimate. Accordingly, a new modifier `validatePool` can be developed and appended to each function in the above list.

```
     modifier validatePool(uint256 _pid) {
        require(_pid < poolInfo.length, "chef: pool exists?");
        _;
    }

    // Deposit LP tokens to pool for RLY allocation.
    function deposit(uint256 _pid, uint256 _amount) public validatePool(_pid) {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accRallyPerShare).div(1e12).sub(user.
                rewardDebt);
            if(pending > 0) {
                safeRallyTransfer(msg.sender, pending);
            }
        }
        if(_amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accRallyPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.11: NoMintLiquidityRewardPools.sol (revised)

Status The issue has been confirmed. For the same reason outlined in Section 3.7, the team decides to leave it as it is for the time being.

3.9 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NoMintLiquidityRewardPools`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

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` [21] exploit, and the recent `Uniswap/Lendf.Me` hack [19].

We notice there are several occasions the `checks-effects-interactions` principle is violated. Using the `NoMintLiquidityRewardPools` as an example, the `emergencyWithdraw()` 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 185) starts before effecting the update on internal states (lines 187−188), 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 very same `emergencyWithdraw()` function.

```
181     // Withdraw without caring about rewards. EMERGENCY ONLY.
182     function emergencyWithdraw(uint256 _pid) public {
183         PoolInfo storage pool = poolInfo[_pid];
184         UserInfo storage user = userInfo[_pid][msg.sender];
185         pool.lpToken.safeTransfer(address(msg.sender), user.amount);
186         emit EmergencyWithdraw(msg.sender, _pid, user.amount);
187         user.amount = 0;
188         user.rewardDebt = 0;
189     }
```

Listing 3.12: NoMintLiquidityRewardPools.sol

Another similar violation can be found in the `deposit()` and `withdraw()` routines within the same contract.

```
    // Deposit LP tokens to pool for RLY allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
```

```
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending = user.amount.mul(pool.accRallyPerShare).div(1e12).sub(user.rewardDebt);
            if(pending > 0) {
                safeRallyTransfer(msg.sender, pending);
            }
        }
        if(_amount > 0) {
            pool.lpToken.safeTransferFrom(address(msg.sender), address(this), _amount);
            user.amount = user.amount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accRallyPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    // Withdraw LP tokens from pool.
    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        require(user.amount >= _amount, "withdraw: not good");
        updatePool(_pid);
        uint256 pending = user.amount.mul(pool.accRallyPerShare).div(1e12).sub(user.rewardDebt);
        if(pending > 0) {
            safeRallyTransfer(msg.sender, pending);
        }
        if(_amount > 0) {
            user.amount = user.amount.sub(_amount);
            pool.lpToken.safeTransfer(address(msg.sender), _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accRallyPerShare).div(1e12);
        emit Withdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.13: NoMintLiquidityRewardPools.sol

In the meantime, we should mention that the supported tokens in `YFI` vaults implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` best practice. An example revision on the `emergencyWithdraw` routine is shown below:

```
    // Withdraw without caring about rewards. EMERGENCY ONLY.
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        uint256 _amount=user.amount
        user.amount = 0;
        user.rewardDebt = 0;
```

```
        pool.lpToken.safeTransfer(address(msg.sender), _amount);
        emit EmergencyWithdraw(msg.sender, _pid, _amount);
    }
```

Listing 3.14: NoMintLiquidityRewardPools.sol (revised)

Status The issue has been confirmed. For the same reason outlined in Section 3.7, the team decides to leave it as it is for the time being.

3.10 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `NoMintLiquidityRewardPools`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [6]

Description

As mentioned in Section 3.7, the Yield Delegating Vaults protocol provides incentive mechanisms that reward the staking of supported assets with `RLY` tokens.The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    // Update the given pool's RLY allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.15: NoMintLiquidityRewardPools.sol

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```
    // Update the given pool's RLY allocation point. Can only be called by the owner.
    function set(uint256 _pid, uint256 _allocPoint) public onlyOwner {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.16: NoMintLiquidityRewardPools.sol (revised)

Status The issue has been confirmed. Given that the reward pools contract had been deployed, the team agrees to exercise care in always using `massUpdatePools = true`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Yield Delegating Vaults protocol. The audited system presents a unique addition to current DeFi offerings by providing a wrapper to current `YFI` vaults and allowing for the collected yields in a designated treasury to fund various projects. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

As a final precaution, 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.

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [14, 15, 16, 17, 20].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [22] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

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

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

[10] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[11] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

[14] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[15] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[16] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[17] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

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

[20] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

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

[22] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.