SMART CONTRACT AUDIT REPORT

for

HARVEST FINANCE

Prepared By: Shuxiao Wang

Hangzhou, China
October 1, 2020

Document Properties

Client	Harvest Finance
Title	Smart Contract Audit Report
Target	Harvest
Version	1.0
Author	Xuxiang Jiang
Auditors	Xuxian Jiang, Huaguo Shi, Jeff Liu
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 1, 2020	Xuxiang Jiang	Final Release
1.0-rc2	September 30, 2020	Xuxian Jiang	Final Release (Release Candidate #2)
1.0-rc1	September 28, 2020	Xuxian Jiang	Final Release (Release Candidate #1)
0.5	September 22, 2020	Xuxian Jiang	Additional Findings #4
0.4	September 20, 2020	Xuxian Jiang	Additional Findings #3
0.3	September 18, 2020	Xuxian Jiang	Additional Findings #2
0.2	September 16, 2020	Xuxian Jiang	Additional Findings #1
0.1	September 14, 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

Given the opportunity to review the **Harvest Protocol** design document and related smart contract source code, 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 Harvest Finance

Similar to YFI, the Harvest Protocol helps farmers of all shapes and sizes get automatic exposure to the highest yield available across a selection of decentralized finance protocols. The harvesting strategies are flexible and designed to be compatible with current and upcoming assets. A variety of strategies can be developed for adoption. In addition to the yields from harvesting, the protocol provides incentives to its depositing users with additional `FARM` tokens. Protocol profits are distributed to the `FARM` holders so that the interests and incentives are better aligned for Harvest users to govern and hold a stake.

The basic information of Harvest protocol is as follows:

Table 1.1: Basic Information of Harvest

Item	Description
Issuer	Harvest Finance
Website	https://harvest.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 1, 2020

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

- https://github.com/harvest-finance/harvest (037d6e3)

1.2 About PeckShield

PeckShield Inc. [23] 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 [18]:

- 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

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

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) [17], 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 Harvest Protocol 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	1	■
Medium	2	■■
Low	3	■■■
Informational	11	■■■■■■■■■■■
Total	17	

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 1 high-severity vulnerability, 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 11 informational recommendations.

Table 2.1: Key Audit Findings of Harvest Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Sanity Checks And Less Friction in NotifyHelper	Coding Practices	Fixed
PVE-002	Informational	Suggested Adherence of Checks-Effects-Interactions Pattern in HardRewards and Vaults	Business Logics	Confirmed
PVE-003	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logics	Fixed
PVE-004	Informational	Improved Event Generation	Error Conditions, Return Values, Status Codes	Fixed
PVE-005	Informational	Unused Import Removal in RewardToken	Coding Practices	Fixed
PVE-006	Informational	Simplified Logic in getReward()	Business Logics	Confirmed
PVE-007	Informational	Consistent Handling of Vault Investment Fraction	Business Logics	Fixed
PVE-008	Informational	Possible Revert in withdrawToVault()	Business Logics	Confirmed
PVE-009	Medium	Logic Error in CRVStrategyStable::depositArbCheck()	Business Logics	Fixed
PVE-010	Informational	Inconsistent/Misplaced Comments Among Multiple Contracts	Coding Practices	Fixed
PVE-011	Informational	Improved Asset Consistency Between Vaults and Strategies	Coding Practices	Confirmed
PVE-012	Medium	Possible Partial Withdrawal With withdrawAllToVault()	Business Logics	Confirmed
PVE-013	Informational	Authenticated salvage() From onlyGovernance	Security Features	Confirmed
PVE-014	Informational	Gas Optimization in withdrawToVault()	Coding Practices	Confirmed
PVE-015	Low	Possible Front-Running For Reduced Return	Time and State	Confirmed
PVE-016	Informational	Optimized claimAndLiquidateCrv() For Improved Investment	Business Logics	Confirmed
PVE-017	High	Overly-Privileged Governance-Controlling EOA	Security Features	Partially Fixed

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 Sanity Checks And Less Friction in NotifyHelper

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `NotifyHelper`
- Category: Coding Practices [14]
- CWE subcategory: CWE-1050 [2]

Description

The Harvest protocol defines unique incentive mechanisms to encourage early adoption. To facilitate the distribution of the inherent protocol token, i.e., `FARM`, across multiple pools in a secure manner, Harvest introduces a new contract named `NotifyHelper`, which defines the main notifier routine of reward amounts to incentivized pools, i.e., `notifyPools()`.

```
/**
* Notifies all the pools, safe guarding the notification amount.
*/
function notifyPools(uint256[] memory amounts, address[] memory pools) public
    onlyGovernance {
  require(amounts.length == pools.length, "Amounts and pools lengths mismatch");
  for (uint i = 0; i < pools.length; i++) {
    require(amounts[i] > 0, "Notify zero");
    NoMintRewardPool pool = NoMintRewardPool(pools[i]);
    IERC20 token = IERC20(pool.rewardToken());
    uint256 limit = token.balanceOf(pools[i]);
    require(amounts[i] <= limit, "Notify limit hit");
    NoMintRewardPool(pools[i]).notifyRewardAmount(amounts[i]);
  }
}
```

Listing 3.1: NotifyHelper.sol

To elaborate, we show the notifier logic above. This logic is restricted to governance only (with the `onlyGovernance` modifier enforcement) and takes two arguments: `amounts` and `pools`. Apparently, it intends to conveniently notify a number of pools with respective reward amounts. For safety, the notified reward amount is no larger than the pool balance. By doing so, it can greatly alleviate the concern on the potential bug that may lock user stakes if the notified reward amount is larger enough to always trigger the RewardPool SafeMath issue [26] and thus cause revert!

Specifically, this issue stems from the calculation of `rewardPerToken()` function (lines 701 – 707). If a large number of reward amount is being notified, we will obtain a large `rewardRate`, which causes the following math `lastTimeRewardApplicable().sub(lastUpdateTime).mul(rewardRate).mul(1e18).div(_totalSupply)` to overflow. Since `rewardPerToken()` is always invoked when stakers attempt to retrieve back their stakes, the always-reverted execution effectively blocks the attempt and thus locks the funds.

```
696     function rewardPerToken() public view returns (uint256) {
697         if (totalSupply() == 0) {
698             return rewardPerTokenStored;
699         }
700         return
701             rewardPerTokenStored.add(
702                 lastTimeRewardApplicable()
703                     .sub(lastUpdateTime)
704                     .mul(rewardRate)
705                     .mul(1e18)
706                     .div(totalSupply())
707             );
708     }
```

Listing 3.2: RewardPool.sol

As a solution, the proposed `notifyPools` reward notifier ensures the reward amount stays within a normal range. In particular, this amount will be required to be no greater than the remaining balance of the reward token in the pool contract. However, it should be noted that the current implementation does not check for any duplicates. As a result, it is theoretically possible to craft the function inputs in a way to notify the same pool multiple times. Each time, the reward amount is no larger than the pool, but multiple notifications to the same pool could bypass this restriction.

In the meantime, to avoid introducing unnecessary frictions, we suggest to revise the current logic in not reverting the transaction when a pools' reward amount is 0 (line 17).

Recommendation Remove possible duplicates in the given inputs and avoid unnecessary reverts for less friction. A more fundamental approach is to apply the latest patch [26].

Status This issue has been confirmed.

3.2 Suggested Adherence of Checks-Effects-Interactions Pattern in HardRewards and Vaults

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `HardRewards`, `Vaults`, `DelayMinter`
- Category: Business Logics [15]
- CWE subcategory: CWE-841 [11]

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` [27] exploit, and the recent `Uniswap/Lendf.Me` hack [24].

We notice there is a few occasions the `checks-effects-interactions` principle is violated. Using `rewardMe()` function in the `HardRewards` contract (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 53) starts before effecting the update on internal states (lines 59), hence violating the principle. In this particular case, if the `lastReward[vault]` has not been timely updated and the external contract has some hidden logic that may be capable of launching `re-entrancy`, a bad actor could continue to reward the given recipient multiple times, even draining all funds loaded into the contract.

```
33   function rewardMe(address recipient, address vault) external onlyController {
34     if (address(token) == address(0)  blockReward == 0) {
35       // no rewards now
36       emit Rewarded(recipient, vault, 0);
37       return;
38     }

40     if (lastReward[vault] == 0) {
41       // vault does not exist
42       emit Rewarded(recipient, vault, 0);
43       return;
44     }

46     uint256 span = block.number.sub(lastReward[vault]);
```

```
47     uint256 reward = blockReward.mul(span);

49     if (reward > 0) {
50       uint256 balance = token.balanceOf(address(this));
51       uint256 realReward = balance >= reward ? reward : balance;
52       if (realReward > 0) {
53         token.safeTransfer(recipient, realReward);
54       }
55       emit Rewarded(recipient, vault, realReward);
56     } else {
57       emit Rewarded(recipient, vault, 0);
58     }
59     lastReward[vault] = block.number;
60   }
```

Listing 3.3: HardRewards.sol

Specifically, in the case that the reward `token` is an ERC777 token, a malicious actor could hijack the `token.safeTransfer()` call (line 53) with a callback function. Within the callback function, they could call the `rewardMe()` function again to withdraw additional amount. The bad actor could do it again and again until all funds in `HardRewards` are drained. Fortunately, this particular case is protected with a `onlyController` modifier and the related token is not an ERC777 token.

Meanwhile, we observe similar pattern violations in other contracts, including `deposit()` and `withdraw()` in `Vaults`, `executeMint()` in `DelayMinter`, and others. The associated reentrancy risks and the notorious history bring up the necessity to implement effective reentrancy prevention in current codebase.

Recommendation Follow the known `checks-effects-interactions` best practice or apply necessary reentrancy prevention by adding the `noReentrancy`-like modifier to affected functions.

Status This issue has been confirmed.

3.3 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DepositHelper, Vault, ...`
- Category: Business Logics [15]
- CWE subcategory: CWE-841 [11]

Description

In Harvest, the `Vault` contract is designed to be the main entry for interaction with farming users. In particular, one entry routine, i.e., `deposit()`, accepts deposits of assets (e.g., `DAI`) and returns with wrapped counterparts (e.g., fDAI). Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of the Harvest protocol. 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 contracts.

```
  function _deposit(uint256 amount, address sender, address beneficiary) internal {
    require(amount > 0, "Cannot deposit 0");
    require(beneficiary != address(0), "holder must be defined");

    if (address(strategy) != address(0)) {
      require(strategy.depositArbCheck(), "Too much arb");
    }

    uint256 toMint = totalSupply() == 0
        ? amount
        : amount.mul(totalSupply()).div(underlyingBalanceWithInvestment());
    _mint(beneficiary, toMint);

    underlying.safeTransferFrom(sender, address(this), amount);

    // update the contribution amount for the beneficiary
    contributions[beneficiary] = contributions[beneficiary].add(amount);
    emit Deposit(beneficiary, amount);
  }
```

Listing 3.4: Vault.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 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. In other words, the above operations, such as deposit() 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 Harvest and affects protocol-wide operation and maintenance. A similar issue can also be found in DepositHelper, FeeRewardForwarder, and LPTokenWrapper.

One possible 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 transfer() or transferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the transfer() or transferFrom() 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 Harvest. In Harvest, 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 To accommodate the support of 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. Moreover, due to the `USDT` support in Harvest, we need to exercise extra caution in monitoring possible subversion and inconsistency if it turns into deflationary in the future.

Status The issue has been fixed by this commit: `8d464a1791a3d48d4b0318fb3c9207075cdede86`. Furthermore, it has been clarified from the team by providing the following elaboration and response:

"*The system is not designed to work with deflationary tokens. If a token is turned into a deflationary token afterwards, during withdrawal, we perform a Math.min on whatever is in the vault and the rightful share. For now, all tokens are non-deflationary (USDT switch is off). As mentioned in the security review document we provided, we do make sure that deflationary coins don't go into the reward pools, so LPTokenWrapper part is fine. (this is the reason we were using Uniswap LP Tokens for some deflationary tokens like BASED in the pool launches).*"

3.4 Improved Event Generation

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Controller`
- Category: Status Codes [16]
- CWE subcategory: CWE-682 [8]

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 monitoring tools.

`Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. For example, Harvest has a number of risk parameters that are dynamically adjustable via governance. However, the current implementation can be greatly benefited by emitting related events when they are being changed.

If the following, we use the `Controller` contract as an example. This contract is a privileged one responsible for controlling the mapping between `vaults` and `strategies`, specifying the allowed set of `hardWorkers`, and configuring the `feeRewardForwarder` address.

```
    function setFeeRewardForwarder(address _feeRewardForwarder) public onlyGovernance {
```

```
    require(_feeRewardForwarder != address(0), "new reward forwarder should not be empty");
    feeRewardForwarder = _feeRewardForwarder;
  }

  function addVaultAndStrategy(address _vault, address _strategy) external onlyGovernance {
      require(_vault != address(0), "new vault shouldn't be empty");
      require(!vaults[_vault], "vault already exists");
      require(_strategy != address(0), "new strategy shouldn't be empty");

      vaults[_vault] = true;
      // adding happens while setting
      IVault(_vault).setStrategy(_strategy);
  }
```

Listing 3.5: Controller.sol

```
  function addHardWorker(address _worker) public onlyGovernance {
    require(_worker != address(0), "_worker must be defined");
    hardWorkers[_worker] = true;
  }

  function removeHardWorker(address _worker) public onlyGovernance {
    require(_worker != address(0), "_worker must be defined");
    hardWorkers[_worker] = false;
  }
```

Listing 3.6: Controller.sol

Though these settings and their changes greatly affect the overall Harvest operations, we however notice related events are not emitted when they are being updated. As our suggestion, we feel strongly the need of emitting related events when these settings are being changed, especially when a new pair of `vault` and `strategy` are being added.

Also, when these `events` are emitted, there is a need to be precise. For example, the following event from the statement `emit Liquidating(crvBalance)` (line 155) is suggested to be relocated to be part of the `if-then` branch (right before line 157).

```
151 function claimAndLiquidateCrv() internal {
152   Mintr(mintr).mint(pool);
153   // claiming rewards and sending them to the master strategy
154   uint256 crvBalance = IERC20(crv).balanceOf(address(this));
155   emit Liquidating(crvBalance);
156   if (crvBalance > 0) {
157     uint256 daiBalanceBefore = IERC20(dai).balanceOf(address(this));
158     IERC20(crv).safeApprove(uni, 0);
159     IERC20(crv).safeApprove(uni, crvBalance);
160     // we can accept 1 as the minimum because this will be called only by a trusted worker
161     IUniswapV2Router02(uni).swapExactTokensForTokens(
162       crvBalance, 1, uniswap_CRV2DAI, address(this), block.timestamp
```

```
        );
        // now we have DAI
        // pay fee before making yCRV
        notifyProfit(daiBalanceBefore, IERC20(dai).balanceOf(address(this)));

        // liquidate if there is any DAI left
        if(IERC20(dai).balanceOf(address(this)) > 0) {
          yCurveFromDai();
        }
        // now we have yCRV
      }
    }
```

Listing 3.7: CRVStrategyYCRV.sol

Recommendation Emit necessary events to timely reflect protocol-wide setting changes.

Status The issue has been fixed by this commit: `8d464a1791a3d48d4b0318fb3c9207075cdede86`.

3.5 Unused Import Removal in RewardToken

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `RewardToken`
- Category: Coding Practices [14]
- CWE subcategory: CWE-561 [7]

Description

Harvest makes good use of a number of reference contracts, such as `ERC20`, `ERC20Detailed`, `ERC20Capped`, `ERC20Mintable`, and `Ownable`, to facilitate its code implementation and organization. For example, the `RewardToken` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `RewardToken` contract, some imports are really not necessary. Specifically, the `Ownable` import is not necessary as it is not used at all in `RewardToken`. Also the `IERC20` import needs to be replaced with `ERC20`.

```
pragma solidity 0.5.16;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20Capped.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20Detailed.sol";
import "@openzeppelin/contracts/token/ERC20/ERC20Mintable.sol";
import "@openzeppelin/contracts/ownership/Ownable.sol";
```

```
import "./Governable.sol";

contract RewardToken is ERC20, ERC20Detailed, ERC20Capped, Governable {

  uint256 public constant HARD_CAP = 5 * (10 ** 6) * (10 ** 18);

  constructor(address _storage) public
  ERC20Detailed("FARM Reward Token", "FARM", 18)
  ERC20Capped(HARD_CAP)
  Governable(_storage) {
    // msg.sender should not be a minter
    renounceMinter();
    // governance will become the only minter
    _addMinter(governance());
  }

  /**
  * Overrides adding new minters so that only governance can authorized them.
  */
  function addMinter(address _minter) public onlyGovernance {
    super.addMinter(_minter);
  }
}
```

Listing 3.8: RewardToken.sol

Recommendation Remove unnecessary imports of reference contracts and revise existing ones if necessary.

Status The issue has been fixed by this commit: `8d464a1791a3d48d4b0318fb3c9207075cdede86`.

3.6 Simplified Logic in getReward()

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StakeLPToken`
- Category: Business Logics [15]
- CWE subcategory: CWE-770 [9]

Description

In the `StakeLPToken` contract, the `getReward()` routine is intended to obtain the calling user's staking rewards. The logic is rather straightforward in calculating possible reward, which, if not zero, is then allocated to the calling (staking) user.

Our examination shows that the current implementation logic can be further optimized. In particular, the `getReward()` routine has a modifier, i.e., `updateReward(msg.sender)`, which timely updates the calling user's (earned) rewards in `rewards[msg.sender]` (line 94).

```
94    function getReward() public updateReward(msg.sender) {
95        uint reward = earned(msg.sender);
96        if (reward > 0) {
97            rewards[msg.sender] = 0;
98            core.mintReward(msg.sender, reward);
99            emit RewardPaid(msg.sender, reward);
100       }
101   }
```

Listing 3.9: StakeLPToken.sol

Having the modifier `updateReward()`, there is no need to re-calculate the earned reward for the caller `msg.sender`. In other words, we can simply re-use the calculated `rewards[msg.sender]` and assign it to the `reward` variable (line 95).

```
modifier updateReward(address account) {
    updateProtocolIncome();
    if (account != address(0)) {
        rewards[account] = _earned(rewardPerTokenStored, account);
        userRewardPerTokenPaid[account] = rewardPerTokenStored;
    }
    _;
}
```

Listing 3.10: StakeLPToken.sol

Recommendation Avoid the duplicated calculation of the caller's reward in `getReward()`, which also leads to (small) beneficial reduction of associated gas cost.

```
94    function getReward() public updateReward(msg.sender) {
95        uint reward = rewards[msg.sender];
96        if (reward > 0) {
97            rewards[msg.sender] = 0;
98            core.mintReward(msg.sender, reward);
99            emit RewardPaid(msg.sender, reward);
100       }
101   }
```

Listing 3.11: StakeLPToken.sol

Status The issue has been confirmed. However, considering this contract is directly based on `Synthetix`, the team considers the less changes there are, the better. Therefore, the team decides to intentionally keep it as is for the time being.

3.7 Consistent Handling of Vault Investment Fraction

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Vault`
- Category: Business Logics [15]
- CWE subcategory: CWE-837 [10]

Description

In Harvest, there is a one-to-one mapping between a `vault` and its `strategy`. Within the `vault`, there is a pair of parameters, i.e., `vaultFractionToInvestDenominator` and `vaultFractionToInvestNumerator`, that specify the desired percentage of funds for investment. This pair parameter can be directly specified when a new `vault` is being deployed or dynamically reconfigured via `setVaultFractionToInvest()`.

In the following, we outline the code for both `constructor()` and `setVaultFractionToInvest()`. If we pay attention to the sanity checks on the above pair of parameters, we notice `constructor()` allows for the case of `_toInvestNumerator == _toInvestDenominator`, but this case is not allowed in `setVaultFractionToInvest()`.

```
constructor(address _storage,
    address _underlying,
    uint256 _toInvestNumerator,
    uint256 _toInvestDenominator
) ERC20Detailed(
  string(abi.encodePacked("FARM_", ERC20Detailed(_underlying).symbol())),
  string(abi.encodePacked("f", ERC20Detailed(_underlying).symbol())),
  ERC20Detailed(_underlying).decimals()
) Controllable(_storage) public {
  underlying = IERC20(_underlying);
  require(_toInvestNumerator <= _toInvestDenominator, "cannot invest more than 100%");
  require(_toInvestDenominator != 0, "cannot divide by 0");
  vaultFractionToInvestDenominator = _toInvestDenominator;
  vaultFractionToInvestNumerator = _toInvestNumerator;
  underlyingUnit = 10 ** uint256(ERC20Detailed(address(underlying)).decimals());
}
```

Listing 3.12: Vault.sol

```
function setVaultFractionToInvest(uint256 numerator, uint256 denominator) external
    onlyGovernance {
  require(denominator > 0, "denominator must be greater than 0");
  require(numerator < denominator, "denominator must be greater than numerator");
  vaultFractionToInvestNumerator = numerator;
  vaultFractionToInvestDenominator = denominator;
}
```

Listing 3.13: Vault.sol

For consistency, it is suggested to apply the same criteria when validating the same parameters.

Recommendation Be consistent in validating the pair of parameters that specify the `vault`'s investment percentage, i.e., `vaultFractionToInvestDenominator` and `vaultFractionToInvestNumerator`. We suggest to modify the `constructor()` routine to disallow the equal case.

Status The issue has been fixed by this commit: `8d464a1791a3d48d4b0318fb3c9207075cdede86`.

3.8 Possible Revert in withdrawToVault()

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SNXRewardStrategy`
- Category: Coding Practices [14]
- CWE subcategory: CWE-1099 [3]

Description

To facilitate the interaction between a `vault` and its `strategy`, Harvest defines a set of standard interfaces that are required and exported in `strategy` instances. Some example interfaces include `withdrawToVault()`, `withdrawAllToVault()`, `salvage()`, `investedUnderlyingBalance()`, `doHardWork()`, and `depositArbCheck()`. Note that `withdrawToVault()` and `withdrawAllToVault()` are used to transfer funds currently held in `strategy` to the associated `vault` contract.

During our analysis of the `SNXRewardStrategy` contract, we notice a corner case that may cause `withdrawToVault()` to fail. To elaborate, we show its code snippet below.

```
/*
*   Withdraws all the asset to the vault
*/
function withdrawToVault(uint256 amount) public restricted {
  // Typically there wouldn't be any amount here
  // however, it is possible because of the emergencyExit
  if(amount > underlying.balanceOf(address(this))){
    // While we have the check above, we still using SafeMath below
    // for the peace of mind (in case something gets changed in between)
    uint256 needToWithdraw = amount.sub(underlying.balanceOf(address(this)));
    rewardPool.withdraw(Math.min(rewardPool.balanceOf(address(this)), needToWithdraw));
  }

  IERC20(underlying).safeTransfer(vault, amount);
}
```

Listing 3.14: SNXRewardStrategy.sol

The function logic is straightforward: it first determines whether the current `strategy` contract holds sufficient funds to satisfy the withdraw request. If yes, it directly transfers the request funds; Otherwise, it needs to withdraw from the `rewardPool`. However, the `rewardPool` case may bring the following scenario when `rewardPool.balanceOf(address(this))< needToWithdraw` (line 240). In other words, the total available balance, including `rewardPool`, may be smaller than the requested amount. As a result, the final transfer with the requested amount (line 243) fails.

Recommendation Detect whether the balance is sufficient and accordingly adjust the amount being transferred out if not. An example revision is shown below:

```
230 /*
231 *   Withdraws all the asset to the vault
232 */
233 function withdrawToVault(uint256 amount) public restricted {
234   // Typically there wouldn't be any amount here
235   // however, it is possible because of the emergencyExit
236   if(amount > underlying.balanceOf(address(this))){
237     // While we have the check above, we still using SafeMath below
238     // for the peace of mind (in case something gets changed in between)
239     uint256 needToWithdraw = amount.sub(underlying.balanceOf(address(this)));
240     rewardPool.withdraw(Math.min(rewardPool.balanceOf(address(this)), needToWithdraw))
        ;
241   }
242
243   IERC20(underlying).safeTransfer(vault, Math.min(amount, underlying.balanceOf(address
        (this))));
244 }
```

Listing 3.15: SNXRewardStrategy.sol

Status This issue has been through a few rounds of discussions. It has come to conclusion that by design, the `revert` is expected when the balance is insufficient to satisfy the `withdrawToVault()` demand.

3.9 Logic Error in CRVStrategyStable::depositArbCheck()

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CRVStrategyStable`
- Category: Business Logics [15]
- CWE subcategory: CWE-841 [11]

Description

As mentioned in Section 3.8, Harvest defines a number of APIs which each `strategy` is required to implement. Here, we discuss one particular interface, i.e., `depositArbCheck()`. This interface is designed to prevent arbitrage opportunities from being aggressively exploited. In particular, it detects the price slippage from previous checkpoints and block new deposits if the deviation is above the prescribed threshold `arbTolerance` (3% in current deployment).

To elaborate, we show the detection logic below. If the current price `currentPrice` is large than the latest checkpoint `curvePriceCheckpoint` and the deviation is more than `arbTolerance` percentage, it is considered an arbitrage attempt and the protocol will block it. However, we find out that the detection logic is flawed. In fact, the current `depositArbCheck()` implementation always returns `true`.

```
104   function depositArbCheck() public view returns(bool) {
105     uint256 currentPrice = underlyingValueFromYCrv(ycrvUnit);
106     if (currentPrice > curvePriceCheckpoint) {
107       return currentPrice.mul(100).div(curvePriceCheckpoint) > 100 - arbTolerance;
108     } else {
109       return curvePriceCheckpoint.mul(100).div(currentPrice) > 100 - arbTolerance;
110     }
111   }
112
113   function setArbTolerance(uint256 tolerance) external onlyGovernance {
114     require(tolerance <= 100, "at most 100");
115     arbTolerance = tolerance;
116   }
```

Listing 3.16: CRVStrategyStable.sol

Specifically, if we examine the case `currentPrice > curvePriceCheckpoint` (line 106), it is guaranteed that the following statement in line 107 is evaluated `true`. A proper implementation needs to revised as follows:

```
104   function depositArbCheck() public view returns(bool) {
105     uint256 currentPrice = underlyingValueFromYCrv(ycrvUnit);
106     if (currentPrice > curvePriceCheckpoint) {
107       return currentPrice.mul(100)) < (100 + arbTolerance).mul(curvePriceCheckpoint)
108     } else {
109       return curvePriceCheckpoint.mul(100) < (100 + arbTolerance).mul(currentPrice);
```

```
    }
  }

  function setArbTolerance(uint256 tolerance) external onlyGovernance {
    require(tolerance <= 100, "at most 100");
    arbTolerance = tolerance;
  }
```

Listing 3.17: CRVStrategyStable.sol

Recommendation Revise the `depositArbCheck()` logic to reflect the intended purpose.

Status The issue has been fixed by this commit: `48adf02d98b5bad2b426d7b833548aeddd62d2f7`.

3.10 Inconsistent/Misplaced Comments Among Multiple Contracts

- ID: PVE-010
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CRVStrategyStable`, `CRVStrategyYCRV`
- Category: Coding Practices [14]
- CWE subcategory: CWE-1116 [4]

Description

Documentations, including comments embedded in the code, are indispensable in our auditing process for a better understanding of the overall protocol design and the underlying implementation. It is also valuable when there is need to maintain, refactor, or extend the current codebase. Our review process exposes a number of occasions where certain comments are inconsistent, misleading, or completely misplaced. In the following, we show three representative cases.

Case I The first example is the `strategy` contract, i.e., `CRVStrategyYCRV`. This contract is designed to invest farmers' assets (held in `vault`), harvest growing yields, and sell any gains, if any, to the original asset. In the following, we show two helper routines, i.e., `withdrawToVault()` and `withdrawAllToVault()`, that are used to withdraw funds back to `vault`. However, both comments above respective routines can be more precise in stating the withdrawed funds are returned back to the `vault`, not `pool`. Note the `pool` here actually means the `gauge`!

```
  /**
  * Withdraws the yCRV tokens to the pool in the specified amount.
  */
  function withdrawToVault(uint256 amountUnderlying) external restricted {
```

```
    withdrawYCrvFromPool(amountUnderlying);
    if (IERC20(underlying).balanceOf(address(this)) < amountUnderlying) {
      claimAndLiquidateCrv();
    }
    uint256 toTransfer = Math.min(IERC20(underlying).balanceOf(address(this)),
        amountUnderlying);
    IERC20(underlying).safeTransfer(vault, toTransfer);
  }

  /**
  * Withdraws all the yCRV tokens to the pool.
  */
  function withdrawAllToVault() external restricted {
    claimAndLiquidateCrv();
    withdrawYCrvFromPool(maxUint);
    uint256 balance = IERC20(underlying).balanceOf(address(this));
    IERC20(underlying).safeTransfer(vault, balance);
  }
```

Listing 3.18: CRVStrategyYCRV.sol

Case II The second example is two other functions defined in the same `CRVStrategyYCRV` contact. These two functions are `doHardWork()` and `investedUnderlyingBalance()`. The comment of `doHardWork()` states that `"Claims and liquidates CRV into yCRV, and then invests all underlying."` However, this strategy indeed claims and liquidates `CRV`, but not into `yCRV`. Instead, `CRV` is claimed and liquidated into the underlying asset (e.g., `DAI` or other stablecoins).

The second function `investedUnderlyingBalance()`, as the name indicates, intends to query the invested amount denominated at the underlying token. However, the current comment reads `"Claims and liquidates CRV into yCRV, and then invests all underlying"`, which apparently is copied from the first function without any modification.

```
  /**
  * Claims and liquidates CRV into yCRV, and then invests all underlying.
  */
  function doHardWork() public restricted {
    claimAndLiquidateCrv();
    investAllUnderlying();
  }

  /**
  * Claims and liquidates CRV into yCRV, and then invests all underlying.
  */
  function investedUnderlyingBalance() public view returns (uint256) {
    return Gauge(pool).balanceOf(address(this)).add(
      IERC20(underlying).balanceOf(address(this))
    );
  }
```

Listing 3.19: CRVStrategyYCRV.sol

Case III The third example is two functions, i.e., `yTokenValueFromYCrv()` and `underlyingValueFromYCrv()` in the `CRVStrategyStable` contract. They are simply misplaced and need to switched with each other to better explain the purpose of each function!

```
/**
* Returns the value of yCRV in underlying token accounting for slippage and fees.
*/
function yTokenValueFromYCrv(uint256 ycrvBalance) public view returns (uint256) {
  return underlyingValueFromYCrv(ycrvBalance) // this is in DAI, we will convert to yDAI
  .mul(10 ** 18)
  .div(yERC20(yVault).getPricePerFullShare()); // function getPricePerFullShare() has 18 decimals for all tokens
}

/**
* Returns the value of yCRV in y-token (e.g., yCRV -> yDai) accounting for slippage and fees.
*/
function underlyingValueFromYCrv(uint256 ycrvBalance) public view returns (uint256) {
  return IPriceConvertor(convertor).yCrvToUnderlying(ycrvBalance, uint256(tokenIndex));
}
```

Listing 3.20: CRVStrategyStable.sol

Recommendation Revise the above code comments to make them coherent with respective functions.

Status The issue has been fixed by this commit: `8d464a1791a3d48d4b0318fb3c9207075cdede86`.

3.11 Improved Asset Consistency Between Vaults and Strategies

- ID: PVE-011
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CRVStrategyStable`, `CRVStrategyYCRV`
- Category: Coding Practices [14]
- CWE subcategory: CWE-1099 [3]

Description

As mentioned in Section 3.7, there is a one-to-one mapping between a `vault` and its `strategy`. To properly link a `vault` with its `strategy`, it is naturally for the two to operate on the same underlying asset. For example, the `VaultDAI` vault allows for `DAI`-based deposits and withdraws. The associated strategy, i.e., a CRVStrategyStable-based instance, naturally has DAI as the underlying asset. If these two have different underlying assets, the link should not be successful.

If we examine the setStrategy() routine in the vault contract, this routine allows for dynamic binding of the vault with a new strategy (line 138). A successful binding needs to satisfy a number of requirements. One specific one is shown as follows: require(IStrategy(_strategy).underlying()== address(underlying), "Vault underlying must match Strategy underlying") (line 130). Apparently, this requirement guarantees the consistency of the underlying asset between the vault and its associated strategy.

```
128  function setStrategy(address _strategy) public onlyControllerOrGovernance {
129    require(_strategy != address(0), "new _strategy cannot be empty");
130    require(IStrategy(_strategy).underlying() == address(underlying), "Vault underlying must match Strategy underlying");
131    require(IStrategy(_strategy).vault() == address(this), "the strategy does not belong to this vault");
132
133    if (address(_strategy) != address(strategy)) {
134      if (address(strategy) != address(0)) { // if the original strategy (no underscore) is defined
135        underlying.safeApprove(address(strategy), 0);
136        strategy.withdrawAllToVault();
137      }
138      strategy = IStrategy(_strategy);
139      underlying.safeApprove(address(strategy), 0);
140      underlying.safeApprove(address(strategy), uint256(~0));
141    }
142  }
```

Listing 3.21: Vaults.sol

However, if we examine the constructor() of various strategy contracts (e.g., CRVStrategyStable, CRVStrategySwerve, and CRVStrategyWRenBTC), the requirement of having the same underlying asset is not enforced. A new strategy deployment with an ill-provided list of arguments with an unmatched underlying asset may cause unintended consequences, including possible asset loss. With that, we suggest to maintain an invariant by ensuring the consistency of the underlying asset when a new strategy is being deployed.

```
constructor(
  address _storage,
  address _underlying,
  address _vault,
  address _ycrvVault,
  address _yVault,
  uint256 _tokenIndex,
  address _ycrv,
  address _curveProtocol,
  address _convertor
)
Controllable(_storage) public {
```

```
    vault = _vault;
    ycrvVault = _ycrvVault;
    underlying = _underlying;
    tokenIndex = TokenIndex(_tokenIndex);
    yVault = _yVault;
    ycrv = _ycrv;
    curve = _curveProtocol;
    convertor = _convertor;

    // set these tokens to be not salvageable
    unsalvagableTokens[underlying] = true;
    unsalvagableTokens[yVault] = true;
    unsalvagableTokens[ycrv] = true;
    unsalvagableTokens[ycrvVault] = true;

    ycrvUnit = 10 ** 18;
    // starting with a stable price, the mainnet will override this value
    curvePriceCheckpoint = ycrvUnit;
  }
```

Listing 3.22: CRVStrategyStable.sol

Recommendation Ensure the consistency of the underlying asset between the `vault` and its associated `strategy`. An example revision is shown below. Note three `strategy` contracts or variants need to be revisited: `CRVStrategyStable`, `CRVStrategySwerve`, and `CRVStrategyWRenBTC`.

```
  constructor(
    address _storage,
    address _underlying,
    address _vault,
    address _ycrvVault,
    address _yVault,
    uint256 _tokenIndex,
    address _ycrv,
    address _curveProtocol,
    address _convertor
  )
  Controllable(_storage) public {
    require(IVault(_vault).underlying() == _underlying, "vault does not support yCRV");
    vault = _vault;
    ycrvVault = _ycrvVault;
    underlying = _underlying;
    tokenIndex = TokenIndex(_tokenIndex);
    yVault = _yVault;
    ycrv = _ycrv;
    curve = _curveProtocol;
    convertor = _convertor;

    // set these tokens to be not salvageable
    unsalvagableTokens[underlying] = true;
    unsalvagableTokens[yVault] = true;
    unsalvagableTokens[ycrv] = true;
```

```
    unsalvagableTokens[ycrvVault] = true;

    ycrvUnit = 10 ** 18;
    // starting with a stable price, the mainnet will override this value
    curvePriceCheckpoint = ycrvUnit;
  }
```

Listing 3.23: CRVStrategyStable.sol

Status This issue has been confirmed. Note that the current implementation also performs necessary sanity checks in `setStrategy()`, which ensures that the `vault`'s underlying asset is the same as `strategy`'s underlying asset. Because of that, the team decides there is no such need.

3.12 Possible Partial Withdrawal With withdrawAllToVault()

- ID: PVE-012
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CRVStrategyStable`
- Category: Business Logics [15]
- CWE subcategory: CWE-841 [11]

Description

As mentioned in Section 3.8, Harvest defines a number of standard APIs which each `strategy` is required to implement. In this section, we discuss one particular interface, i.e., `withdrawAllToVault()`. This interface is designed to withdraw all available underlying assets back to the linked `vault` contract. This may happen when a new `strategy` is activated to replace an old one.

To elaborate, we show the `withdrawAllToVault()` logic below. The full withdrawal is implemented in a helper routine named `yCurveToUnderlying()` (line 206). However, a close examination of this helper routine shows it may not always achieve the intended full withdrawal.

```
198  /**
199  * Withdraws all assets from the vault. We ask the yCRV vault to give us our entire
       yCRV balance
200  * and then convert it to the underlying asset using the Curve protocol.
201  */
202  function withdrawAllToVault() external restricted {
203    uint256 shares = IERC20(ycrvVault).balanceOf(address(this));
204    IVault(ycrvVault).withdraw(shares);
205    // withdraw everything until there is only dust left
206    yCurveToUnderlying(uint256(~0));
207    uint256 actualBalance = IERC20(underlying).balanceOf(address(this));
208    if (actualBalance > 0) {
209      IERC20(underlying).safeTransfer(vault, actualBalance);
210    }
```

```
}
```

Listing 3.24: CRVStrategyStable.sol

To understand the reason, we show below the current `yCurveToUnderlying()` implementation. Its logic is not complicated: Firstly it determines the available amount of `yToken` for withdrawal; Next it removes the corresponding liquidity of that amount from the `curve` pool; Finally the returned amount of `yToken` is converted (line 171 – 173) back to the underlying asset. (In our case, it is a stablecoin.)

```
144 /**
145 * Uses the Curve protocol to convert the yCRV back into the underlying asset. If it cannot acquire
146 * the limit amount, it will acquire the maximum it can.
147 */
148 function yCurveToUnderlying(uint256 underlyingLimit) internal {
149   uint256 ycrvBalance = IERC20(ycrv).balanceOf(address(this));
150
151   // this is the maximum number of y-tokens we can get for our yCRV
152   uint256 yTokenMaximumAmount = yTokenValueFromYCrv(ycrvBalance);
153   if (yTokenMaximumAmount == 0) {
154     return;
155   }
156
157   // ensure that we will not overflow in the conversion
158   uint256 yTokenDesiredAmount = underlyingLimit == uint256(~0) ?
159     yTokenMaximumAmount : yTokenValueFromUnderlying(underlyingLimit);
160
161   uint256[4] memory yTokenAmounts = wrapCoinAmount(
162     Math.min(yTokenMaximumAmount, yTokenDesiredAmount));
163   uint256 yUnderlyingBalanceBefore = IERC20(yVault).balanceOf(address(this));
164   IERC20(ycrv).safeApprove(curve, 0);
165   IERC20(ycrv).safeApprove(curve, ycrvBalance);
166   ICurveFi(curve).remove_liquidity_imbalance(
167     yTokenAmounts, ycrvBalance
168   );
169   // now we have yUnderlying asset
170   uint256 yUnderlyingBalanceAfter = IERC20(yVault).balanceOf(address(this));
171   if (yUnderlyingBalanceAfter > yUnderlyingBalanceBefore) {
172     // we received new yUnderlying tokens for yCRV
173     yERC20(yVault).withdraw(yUnderlyingBalanceAfter.sub(yUnderlyingBalanceBefore));
174   }
175 }
```

Listing 3.25: CRVStrategyStable.sol

As we are calling `yCurveToUnderlying(uint256(~0))` with the intent of a full withdrawal, the withdrawal of only the difference between `yUnderlyingBalanceBefore` and `yUnderlyingBalanceAfter`, i.e., `yUnderlyingBalanceAfter.sub(yUnderlyingBalanceBefore)` (line 173), is not sufficient. In fact, to enable full withdrawal back to `vault`, we need to initialize `yUnderlyingBalanceBefore` to be 0 (line 163).

Recommendation Revise the `yCurveToUnderlying()` logic to be compatible with full withdrawal.

An example revision is shown below:

```
/**
* Uses the Curve protocol to convert the yCRV back into the underlying asset. If it cannot acquire
* the limit amount, it will acquire the maximum it can.
*/
function yCurveToUnderlying(uint256 underlyingLimit) internal {
  uint256 ycrvBalance = IERC20(ycrv).balanceOf(address(this));

  // this is the maximum number of y-tokens we can get for our yCRV
  uint256 yTokenMaximumAmount = yTokenValueFromYCrv(ycrvBalance);
  if (yTokenMaximumAmount == 0) {
    return;
  }

  // ensure that we will not overflow in the conversion
  uint256 yTokenDesiredAmount = underlyingLimit == uint256(~0) ?
    yTokenMaximumAmount : yTokenValueFromUnderlying(underlyingLimit);

  uint256[4] memory yTokenAmounts = wrapCoinAmount(
    Math.min(yTokenMaximumAmount, yTokenDesiredAmount));
  uint256 yUnderlyingBalanceBefore = underlyingLimit == uint256(~0) ? 0: IERC20(yVault).balanceOf(address(this));
  IERC20(ycrv).safeApprove(curve, 0);
  IERC20(ycrv).safeApprove(curve, ycrvBalance);
  ICurveFi(curve).remove_liquidity_imbalance(
    yTokenAmounts, ycrvBalance
  );
  // now we have yUnderlying asset
  uint256 yUnderlyingBalanceAfter = IERC20(yVault).balanceOf(address(this));
  if (yUnderlyingBalanceAfter > yUnderlyingBalanceBefore) {
    // we received new yUnderlying tokens for yCRV
    yERC20(yVault).withdraw(yUnderlyingBalanceAfter.sub(yUnderlyingBalanceBefore));
  }
}
```

Listing 3.26: CRVStrategyStable.sol

Status This issue has been confirmed. The team has made extra efforts in clarifying that "*this is not a security issue as this doesn't cause any accounting issue for the users and the contract would not have yVault tokens normally. If you follow the logic, the investment converts the underlying to yTokens, then from yTokens to yCRV immediately to the yCRV vault. The only way that there would be some yTokens here is that someone intentionally send it to the strategy. It is also simple to resolve and digest the tokens here. We only need to call doHardwork that converts all yTokens to yCRV along the execution, then it's gone.*"

3.13 Authenticated salvage() From onlyGovernance

- ID: PVE-013
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: SNXRewardStrategy
- Category: Security Features [12]
- CWE subcategory: CWE-285 [5]

Description

Besides a variety of normal operations, Harvest also gives considerations of accidentally transferred assets into various Harvest contracts. In particular, it develops a helper routine named `salvage()` . As the name indicates, it is mainly used for the very purpose of retrieving back other locked assets. To avoid legitimate assets held in current contract from being affected, `salvage()` imposes two restrictions: the first one is the token being retrieved must not belong to `unsalvagableTokens`; the second one is the function must be invoked via governance. The `unsalvagableTokens` array defines the list tokens that should not participate in the salvage.

This is a nice feature to recover otherwise lost funds. Our analysis shows certain inconsistency across different contracts with the `salvage()` support. In the following, we show the comparison between `CRVStrategyStable` and `SNXRewardStrategy`. Apparently, the `CRVStrategyStable`'s version of `salvage()` is properly enforced by `onlyGovernance` while the `SNXRewardStrategy` version is enforced by `onlyControllerOrGovernance`. While both `controller` and `governance` are considered trustworthy, it is important to be consistent!

```
/**
* Salvages a token. We cannot salvage the shares in the yCRV pool, yCRV tokens, or
    underlying
* assets.
*/
function salvage(address recipient, address token, uint256 amount) public
    onlyGovernance {
  // To make sure that governance cannot come in and take away the coins
  require(!unsalvagableTokens[token], "token is defined as not salvageable");
  IERC20(token).safeTransfer(recipient, amount);
}
```

Listing 3.27: CRVStrategyStable.sol

```
/*
*   Governance or Controller can claim coins that are somehow transferred into the
    contract
*   Note that they cannot come in take away coins that are used and defined in the
    strategy itself
*   Those are protected by the "unsalvagableTokens". To check, see where those are
    being flagged.
```

```
*/
function salvage(address recipient, address token, uint256 amount) external
    onlyControllerOrGovernance {
  // To make sure that governance cannot come in and take away the coins
  require(!unsalvagableTokens[token], "token is defined as not salvagable");
  IERC20(token).safeTransfer(recipient, amount);
}
```

Listing 3.28: SNXRewardStrategy.sol

Moreover, we notice current `savage()` supports a number of ERC20-compliant tokens. However, it does not support of salvaging `ETH` tokens.

Recommendation Make a consistent access control policy on `salvage` by strictly enforcing it with the `onlyGovernance` modifier. Also, add the new `ETH` support in `savage()`.

Status This issue has been confirmed. And here is the team's response "*Our design has progressed over time, and we now rely only on governance to perform the salvage. Only the authorized party is allowed to salvage the tokens, and only a subset that does not take immediate part in the reward liquidation flow.*" Considering the fact that this routine is already guarded by trusted entities, we agree there is no need for further revision.

3.14 Gas Optimization in withdrawToVault()

- ID: PVE-014
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CRVStrategyStable`
- Category: Coding Practices [14]
- CWE subcategory: CWE-1050 [2]

Description

While reviewing the `withdrawToVault()` execution logic, we notice a potential optimization to execute `investAllUnderlying()` only when necessary. Especially, the `withdrawToVault()` logic is straightforward in firstly determining the current `ycrvVault` balance, then withdrawing and converting all of them into the underlying asset, next handling the withdraw request, and finally re-investing all remaining funds accordingly to the current `strategy`.

Note that if the withdraw request completely drains current funds, there is no need to further call `investAllUnderlying()`. Therefore, we can simply add a check up-front to detect this case. By doing so, we can not only simplify the execution logic, but also reduce the gas cost.

```
/**
* Withdraws an underlying asset from the strategy to the vault in the specified amount
    by asking
```

```
* the yCRV vault for yCRV (currently all of it), and then removing imbalanced liquidity from
* the Curve protocol. The rest is deposited back to the yCRV vault. If the amount requested cannot
* be obtained, the method will get as much as we have.
*/
function withdrawToVault(uint256 amountUnderlying) external restricted {
  // If we want to be more accurate, we need to calculate how much yCRV we will need here
  uint256 shares = IERC20(ycrvVault).balanceOf(address(this));
  IVault(ycrvVault).withdraw(shares);
  yCurveToUnderlying(amountUnderlying);
  // we can transfer the asset to the vault
  uint256 actualBalance = IERC20(underlying).balanceOf(address(this));
  if (actualBalance > 0) {
    IERC20(underlying).safeTransfer(vault, Math.min(amountUnderlying, actualBalance));
  }

  // invest back the rest
  investAllUnderlying();
}
```

Listing 3.29: CRVStrategyStable.sol

Recommendation Optimize the execution path to invoke `investAllUnderlying()` only when necessary.

```
/**
* Withdraws an underlying asset from the strategy to the vault in the specified amount by asking
* the yCRV vault for yCRV (currently all of it), and then removing imbalanced liquidity from
* the Curve protocol. The rest is deposited back to the yCRV vault. If the amount requested cannot
* be obtained, the method will get as much as we have.
*/
function withdrawToVault(uint256 amountUnderlying) external restricted {
  // If we want to be more accurate, we need to calculate how much yCRV we will need here
  uint256 shares = IERC20(ycrvVault).balanceOf(address(this));
  IVault(ycrvVault).withdraw(shares);
  yCurveToUnderlying(amountUnderlying);
  // we can transfer the asset to the vault
  uint256 actualBalance = IERC20(underlying).balanceOf(address(this));
  if (actualBalance > 0) {
    IERC20(underlying).safeTransfer(vault, Math.min(amountUnderlying, actualBalance));
    // invest back the rest
    If (actualBalance>amountUnderlying) { investAllUnderlying();}
  }
}
```

Listing 3.30: CRVStrategyStable.sol

Status This issue has been confirmed.

3.15 Possible Front-Running For Reduced Return

- ID: PVE-015
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `CRVStrategyYCRV`
- Category: Time and State [13]
- CWE subcategory: CWE-362 [6]

Description

In Harvest, a number of `strategy` contracts have been designed and implemented to invest farmers' assets (held in `vaults`), harvest growing yields, and sell any gains, if any, to the original asset.

Using the `CRVStrategyYCRV` strategy as an example, a pre-configured worker can call `doHardWork()` that basically collects any pending rewards (via `claimAndLiquidateCrv()-- line 180`) and swaps them to the underlying asset for re-investment (via `investAllUnderlying()` – line 181).

```
176 /**
177 * Claims and liquidates CRV into yCRV, and then invests all underlying.
178 */
179 function doHardWork() public restricted {
180   claimAndLiquidateCrv();
181   investAllUnderlying();
182 }
```

Listing 3.31: CRVStrategyYCRV.sol

The `claimAndLiquidateCrv()` routine deserves our attention. Since this one directly collects rewards, if any, and brings actual gains for previous investment. We notice the collected rewards are routed to `UniswapV2` in order to swap them to the underlying token for next-round of re-investment and yielding. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

```
function claimAndLiquidateCrv() internal {
  Mintr(mintr).mint(pool);
  // claiming rewards and sending them to the master strategy
  uint256 crvBalance = IERC20(crv).balanceOf(address(this));
  emit Liquidating(crvBalance);
  if (crvBalance > 0) {
    uint256 daiBalanceBefore = IERC20(dai).balanceOf(address(this));
    IERC20(crv).safeApprove(uni, 0);
    IERC20(crv).safeApprove(uni, crvBalance);
    // we can accept 1 as the minimum because this will be called only by a trusted
        worker
```

```
      IUniswapV2Router02(uni).swapExactTokensForTokens(
        crvBalance, 1, uniswap_CRV2DAI, address(this), block.timestamp
      );
      // now we have DAI
      // pay fee before making yCRV
      notifyProfit(daiBalanceBefore, IERC20(dai).balanceOf(address(this)));

      // liquidate if there is any DAI left
      if(IERC20(dai).balanceOf(address(this)) > 0) {
        yCurveFromDai();
      }
      // now we have yCRV
    }
  }
```

Listing 3.32: CRVStrategyYCRV.sol

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user or the `strategy` contract in our case because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status This issue has been confirmed. Note that this is a common issue for any trade in a decentralized exchange and there is no effective mitigation that is currently available. However, certain best practices can be applied, including the use of limiting slippages or setting necessary expiry timestamps.

3.16 Optimized claimAndLiquidateCrv() For Improved Investment

- ID: PVE-016
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CRVStrategyStable`
- Category: Business Logics [15]
- CWE subcategory: CWE-841 [11]

Description

In last section, we have examined the `claimAndLiquidateCrv()` routine for a possible front-running attack. In this section, we continue our focus on this routine and report a possible optimization.

To elaborate, we show below the implementation of the `claimAndLiquidateCrv()` routine. It proceeds by firstly claiming possible `CRV` rewards (line 152) and then liquidating the rewards (lines 156 – 173). The liquidation process is basically a swap trade (on the popular `UniswapV2` platform) to sell `CRV` for `DAI`. After that, the returned `DAI` gains will be deposited into the `curve` pool as our next-round of investment (line 170).

```
151   function claimAndLiquidateCrv() internal {
152     Mintr(mintr).mint(pool);
153     // claiming rewards and sending them to the master strategy
154     uint256 crvBalance = IERC20(crv).balanceOf(address(this));
155     emit Liquidating(crvBalance);
156     if (crvBalance > 0) {
157       uint256 daiBalanceBefore = IERC20(dai).balanceOf(address(this));
158       IERC20(crv).safeApprove(uni, 0);
159       IERC20(crv).safeApprove(uni, crvBalance);
160       // we can accept 1 as the minimum because this will be called only by a trusted
          worker
161       IUniswapV2Router02(uni).swapExactTokensForTokens(
162         crvBalance, 1, uniswap_CRV2DAI, address(this), block.timestamp
163       );
164       // now we have DAI
165       // pay fee before making yCRV
166       notifyProfit(daiBalanceBefore, IERC20(dai).balanceOf(address(this)));
167
168       // liquidate if there is any DAI left
169       if(IERC20(dai).balanceOf(address(this)) > 0) {
170         yCurveFromDai();
171       }
172       // now we have yCRV
173     }
174   }
```

Listing 3.33: CRVStrategyYCRV.sol

Notice that the conversion (line 170) of any pending `DAI` balance to `yCRV` only occurs on the condition when non-zero `CRV` rewards are claimed. However, there is no need to restrict ourselves to this condition only. The fact of defining `yCurveFromDai()` as a public function motivates us to move its call from inside the `if-then` branch (line 156) in `claimAndLiquidateCrv()` to outside. By doing so, every call to `doHardWork()` will automatically and timely convert any pending `DAI` balance to `yCRV` to maximize the investment return.

Recommendation Optimize the implementation by moving the `yCurveFromDai()` call outside.

```
151  function claimAndLiquidateCrv() internal {
152    Mintr(mintr).mint(pool);
153    // claiming rewards and sending them to the master strategy
154    uint256 crvBalance = IERC20(crv).balanceOf(address(this));
155    emit Liquidating(crvBalance);
156    if (crvBalance > 0) {
157      uint256 daiBalanceBefore = IERC20(dai).balanceOf(address(this));
158      IERC20(crv).safeApprove(uni, 0);
159      IERC20(crv).safeApprove(uni, crvBalance);
160      // we can accept 1 as the minimum because this will be called only by a trusted
         worker
161      IUniswapV2Router02(uni).swapExactTokensForTokens(
162        crvBalance, 1, uniswap_CRV2DAI, address(this), block.timestamp
163      );
164      // now we have DAI
165      // pay fee before making yCRV
166      notifyProfit(daiBalanceBefore, IERC20(dai).balanceOf(address(this)));
167    }
168
169    // liquidate if there is any DAI left
170    if(IERC20(dai).balanceOf(address(this)) > 0) {
171      yCurveFromDai();
172    }
173    // now we have yCRV
174  }
```

Listing 3.34: CRVStrategyYCRV.sol

Status This issue has been confirmed. In the meantime, the team considers that it may not make much difference, because the excess `DAI` would be traded on the next iteration regardless.

3.17 Overly-Privileged Governance-Controlling EOA

- ID: PVE-017
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: Controller
- Category: Security Features [12]
- CWE subcategory: CWE-285 [5]

Description

In Harvest, governance is a privileged role that sets up new vaults and new strategies, adjusts a variety of protocol parameters, and assigns other system-wide other roles, e.g., controller, minter, and hardworkers. With great privilege comes great responsibility. In this section, we examine the governance subsystem in Harvest.

Our analysis shows that the governance role is indeed privileged, but it is currently controlled by an externally owned account (EOA), which raises necessary concerns from the community. In the following, we show the current state of privilege assignment in Harvest. Specifically, we kept track of the current deployment of various contracts in Harvest, including six active vaults (and their respective strategies), the controller, as well as current governance. Also, for each deployment, we further extract the controlling contract or governance, if any. Our results are shown in Table 3.1.

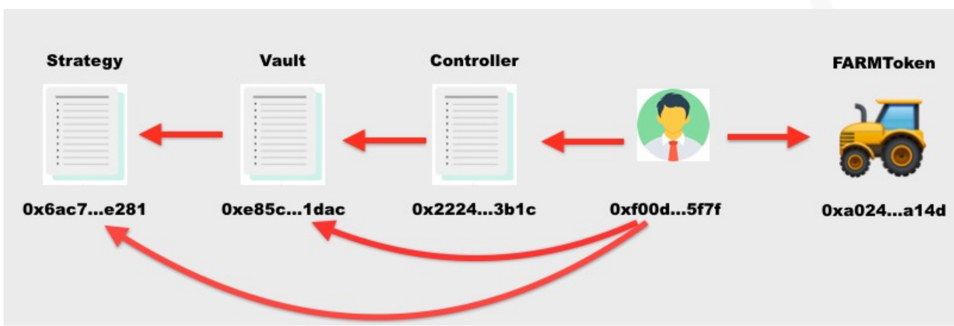

Figure 3.1: The Current Governance Chain of Harvest

To further elaborate, we draw the governance chain based on the current deployment of Harvest in Figure 3.1. We emphasize that the FARM token contract is also administrated by the governance EOA that is authorized to add new minters to mint new FARM tokens. The current setup, if not changed, may pose serious concerns for the future development and wider community adoption.

We notice that the mapping between current vaults and their strategies is properly set up. Currently, they are all administrated by the same controller contract and this administration is appropriate as the controller contract is indeed authorized to configure various aspects of vaults

Table 3.1: Current Contract Deployment of `Harvest` (as of September 16, 2020)

Contract	Address	Controller/Governance
FARMToken	0xa0246c9032bc3a600820415ae600c6388619a14d	0xf00dd244228f51547f0563e60bca65a30fbf5f7f
DAI Vault	0xe85c8581e60d7cd32bbfd86303d2a4fa6a951dac	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
USDC Vault	0xc3f7ffb5d5869b3ade9448d094d81b0521e8326f	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
USDT Vault	0xc7ee21406bb581e741fbb8b21f213188433d9f2f	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
WBTC Vault	0xc07eb91961662d275e2d285bdc21885a4db136b0	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
renBTC Vault	0xfbe122d0ba3c75e1f7c80bd27613c9f35b81feec	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
crvRenBTC Vault	0x192e9d29d43db385063799bc239e772c3b6888f3	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
CRVStrategy SwerveDAIMainnet	0x6ac7575a340a3dab2ae9ca07c4dbfc6bf8e7e281	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
CRVStrategy SwerveUSDCMainnet	0x18c4325ae10fc84895c77c8310d6d98c748e9533	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
CRVStrategy SwerveUSDTMainnet	0x01fcb5bc16e8d945ba276dccfee068231da4ce33	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
CRVStrategy WBTCMainnet	0xe7048e7186cb6f12c566a6c8a818d9d41da6df19	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
CRVStrategy RENBTCMainnet	0x2eadfb06f9d890eba80e999eaba2d445bc70f006	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
CRVStrategy WRenBTCMixMainnet	0xaf2d2e5c5af90c782c008b5b287f20334eeb308e	0xf00dd244228f51547f0563e60bca65a30fbf5f7f 0x222412af183bceadefd72e4cb1b71f1889953b1c
Deployer	0xf00dd244228f51547f0563e60bca65a30fbf5f7f	N/A

and their `strategies`, including the addition of new `vaults`, the configuration of new grey members, the setting of `hardworkers` etc.

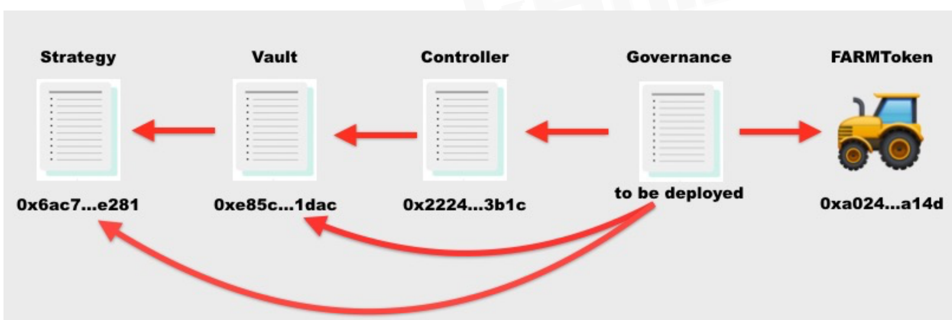

Figure 3.2: The Expected Governance Chain of `Harvest`

However, it is indeed worrisome that the final authority of the entire governance chain is still controlled by an EOA account, i.e., `0xf00dd244228f51547f0563e60bca65a30fbf5f7f`. This EOA address happens to be the same deployer address of `Harvest` and also has the authority of adding `minters` to inflate `FARM` tokens. It is also important to point out that the activation of a privileged operation does not go through an appropriate timelock. In other words, current prototype setup assumes trusted strategies. If a malicious one is introduced and linked with a current vault by successfully bypassing the governance EOA's scrutiny, it may immediately cause serious consequences, including jeopardizing users' funds.

The current administration setup may have the short-term benefit for rapid development and iteration. However, such setup, if unchanged, will eventually hurt its own progress by negatively impacting community engagement and adoption. With a proper community-based on-chain governance, we expect its governance chain should be revised as follows:

Recommendation Promptly transfer the governance privileges from current EOA to the intended governance contract, including the mintability of FARM and the administration of both vaults and strategies. In addition, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed. The team is taking measures to move away from the EOA-based governance, but the details are still forthcoming. As one mitigation step, the team is currently working on adding a timelock to reduce the amount of power the governance has over the vaults.

4 Conclusion

In this audit, we thoroughly analyzed the design and implementation of the Harvest protocol. The system presents a clean and consistent design that makes it distinctive and valuable alternative to current yield farming offerings. The current code base is well organized and those identified issues are promptly confirmed and fixed. However, one main concern is due to the fact that under current deployment, the protocol-wide privilege management, including the mintability of `FRAM`, is not under the control of a community-based governance.

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 [19, 20, 21, 22, 25].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [28] 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-1050: Excessive Platform Resource Consumption within a Loop. https://cwe.mitre.org/data/definitions/1050.html.

[3] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[4] MITRE. CWE-1116: Inaccurate Comments. https://cwe.mitre.org/data/definitions/1116.html.

[5] MITRE. CWE-285: Improper Authorization. https://cwe.mitre.org/data/definitions/285.html.

[6] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[7] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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

[9] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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

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

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

[13] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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

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

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

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

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

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

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

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

[26] Synthetix Improvement Proposals. SIP 77: StakingRewards bug fix's and Pausable stake(). https://sips.synthetix.io/sips/sip-77.

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

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