Least Authority
PRIVACY MATTERS

Harvest Smart Contracts

Security Audit Report

Harvest Finance

Final Report Version: 17 February 2021

Table of Contents

Overview

Background

Harvest Finance has requested that Least Authority perform a security audit of the Harvest Smart Contracts. Harvest is a tool that aims to help users get automatic exposure to the highest yield available across select Decentralized Finance (DeFi) protocols and optimizes the yields that are received using the latest farming techniques. FARM is the cashflow token for Harvest.

Project Dates

- **November 30 - December 28**: Code review *(Completed)*
- **December 29**: Delivery of Initial Audit Report *(Completed)*
- **January 12:** Delivery of Updated Initial Audit Report *(Completed)*
- **February 8 - 11:** Verification completed *(Completed)*
- **February 12:** Delivery of Final Audit Report *(Completed)*
- **February 17**: Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Nathan Ginnever, Security Researcher and Engineer
- Dominc Tarr, Security Researcher and Engineer
- Bryan White, Security Researcher and Engineer
- Alex Lewis, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Harvest Smart Contracts followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

In-Scope

The specified contracts contained within the following code repository are considered in-scope for the review:

- Harvest: https://github.com/harvest-finance/harvest
    - Vault.sol
    - VaultProxy.sol
    - VaultStorage.sol
    - strategiesV2/*
    - strategies/idle/IdleFinanceStrategy.sol
    - strategies/compound/CompoundWETHFoldStrategy.sol
    - strategies/SNXRewards/SushiMasterChefLPStrategy.sol
    - strategies/upgradability/*
    - strategies/LiquidityRecipient.sol

Specifically, we examined the Git revisions for our initial review:

```
De5cd3479d4419516887b9cf16ba3343bdc59fdf
```

For the verification, we examined the Git revision:

`5d1fd6fce5badd6934d6414612ab099371d24365`

This subdirectory was cloned for use during the audit and is linked for reference in this report:

https://github.com/LeastAuthority/harvest

All file references in this document use Unix-style paths relative to the project's root directory.

Out of Scope

The following contracts and components are considered out of scope for this review:

- RewardToken.sol
- DelayMinter.sol
- DepositHelper.sol
- DoHardWorkBatch
- Grain.sol
- NotifyHelper.sol
- RewardPool.sol
- ExclusiveRewardPool.sol
- strategies/SplitterStrategy.sol
- strategies/curve/*
- strategies/ProfitNotifier.sol
- strategies/RewardTokenProfitNotifier.sol
- AutoStakeMultiAsset.sol
- strategies/StableVaultMigrator.sol
- strategies/VaultMigratorStrategy.sol
- strategies/VaultWithdrawDisabledStrategy.sol
- strategies/SNXRewards/DEGORewardInterface.sol
- strategies/SNXRewards/DEGOSimpleStrategy.sol
- strategies/SNXRewards/SNXRewardKittenStrategy.sol
- strategies/SNXRewards/SNXRewardUniLPStrategy.sol
- strategies/compound/WETHCreamNoFoldStrategy.sol
- strategies/compound/CreamNoFoldStrategyWETHMainnet.sol
- strategies/compound/CompoundStrategy.sol

In addition, any dependency and third party code was considered out of scope for this review, unless specifically mentioned as in-scope.

Supporting Documentation

The following documentation was available to the review team:

- Harvest_Finance_Documentation (1).pdf (*provided to Least Authority by the Harvest Finance team via email on 21 October 2020*)
- AuditRequests.pdf (*provided to Least Authority by the Harvest Finance team via Telegram on 30 November 2020*)
- Harvest Finance Community Wiki: https://farm.chainwiki.dev/en/home
- README: https://github.com/harvest-finance/harvest/blob/De5cd3479d4419516887b9cf16ba3343bdc59fdf/README.md
- Vault redesign: https://github.com/harvest-finance/harvest/blob/De5cd3479d4419516887b9cf16ba3343bdc59fdf/VaultRedesign.MD

- Harvest Flashloan Economic Attack Post-Mortem: https://medium.com/harvest-finance/harvest-flashloan-economic-attack-post-mortem-3cf900d65217

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the contracts;
- Potential misuse and gaming of the smart contracts;
- Attacks that impacts funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Economic incentives: ensure token economics (monetary incentives to punish bad behavior and reward good behavior) are included and functional;
- Denial of Service (DoS) and other security exploits that would impact the contracts intended use or disrupt the execution of the contract;
- Vulnerabilities in the smart contracts code;
- Protection against malicious attacks and other ways to exploit contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Review Scope

In reviewing the Harvest Smart Contracts system, our team primarily focused on the files considered to be of the highest priority by the Harvest Finance team, which was communicated to us in the AuditRequest.pdf scope document. In order for our team to gain a full understanding and appreciation of the larger system and abstracted functionality of the smart contracts (i.e., complex contract interactions), it was necessary and pertinent to reference the out of scope components of the system. For example, the upgradeable system that `SushiMasterChefLPStrategy.sol` inherits was quickly examined, as recent analysis has shown that initializers and the `delegatecall` functionality can be prone to error. We found that the Harvest Finance system utilizes the OpenZeppelin framework version 2.5.0, however, the current release of OpenZeppelin is version 3.3.0. Given that Harvest Finance implements an older version, we checked to ensure that the initializer did not contain a revision bug similar to the one discovered in Aave. We recommend that future reviews broaden the scope to incorporate all aspects of the system that are security critical and may be prone to vulnerabilities, even in instances where those components have undergone previous audits as it has been demonstrated that issues can potentially be missed.

Code Quality

The code is well-organized, well-structured, and consistent, and the directory structure is relatively flat. We found the code style to be intuitive and the variable and function names to be logical and human-readable, which makes the code easy to navigate and understand for both contributors and reviewers. We commend the Harvest Finance team for these efforts as they have reduced the opportunity for code being misunderstood and time consuming to review, thus minimizing the risk of errors and bugs being overlooked.

There is a large test suite which encompases many simple and happy path test cases. In general, we suggest the test coverage be expanded to include revert cases and strategy integration tests. This will prevent errors or potential vulnerabilities from being introduced with the use of complex and changing strategies (Suggestion 3). In addition, we recommend implementing tests in TypeScript rather than JavaScript, as TypeScript has become an industry standard and facilitates development speed while helping to eliminate common classes of bugs found in pure JavaScript, where types are non-existent.

Additionally, we suggest upgrading the compiler version from 0.5.16 to 0.7.0 or higher. However, we recommend against upgrading to a version higher than 0.7.4, as more recent versions are more likely to have the potential for unknown issues (Suggestion 2).

Dependency Concerns

There are many dependencies and external contracts included in the Harvest Finance project, adding further complexity to the system at large. While a considerable number of the external dependencies are OpenZeppelin, which is considered to be a trusted and well-established source, we recommend that dependencies continue to be closely monitored and updated regularly to the most current versions, to ensure that fixes for recent bugs and vulnerabilities have been integrated into the system.

Another area of concern is the use of out-dated `npm` packages. The Harvest Finance system is currently using version 2.5.0 while, at present, OpenZeppelin is on version 3.3.0. The `npm` packages will need to be updated when the compiler version is updated, thus addressing both concerns.

To allow for upgradeability, Harvest Finance makes use of a fairly high number of dependencies and separations of logic, which makes reviewing functions difficult at times, as the execution body of some functions is distributed across multiple files. The probability for error is decreased as the complexity of the code and the number of moving parts is limited within the implementation. As a result, we recommend considering a restructure of the code or using more of the stack within functions, which will increase the readability of the code and reduce general complexity.

There are repeat occurrences within the Harvest Finance system where the execution of the smart contracts extends into the strategy that Harvest Finance is farming, requiring a comprehensive understanding of the specific DeFi system that the strategy relies on. For example, there are Uniswap and SushiSwap token exchanges that require an understanding of those exchanges and how they provide swaps. This particular dependency concern has led to the most prevalent issues that we were tasked to review by the Harvest Finance team, namely flash loans and large token holder price manipulation (Issue A; Issue B). While Uniswap and SushiSwap are well known and did not require much effort to review, less familiar farming strategies such as Idle Finance, that work with other lending platforms to generate yields with different pricing mechanisms, required a more in-depth review and were more unfamiliar to our team.

Documentation

Throughout the code base, we found that comments are absent in several areas and that many of the existing comments require clarification. We suggest adding Solidity-style comments, NatSpec Format, to all contract functions in addition to increasing overall comment coverage in the code base (Suggestion 1).

Furthermore, our team found the Vault redesign documentation to be helpful as it describes the previous attack, along with the mitigation and its side effects that was implemented by the Harvest Finance team. While the documentation does not provide an example, one was provided in the post mortem blog post published by the Harvest Finance team. We recommend that this level of detail be further incorporated into the Vault redesign documentation (Suggestion 5).

In order to gain a comprehensive understanding of strategies, and since strategies use other DeFi systems, considerable effort was required in order to fully understand these systems, which facilitated our ability to review and reason about the strategies. We recommend improving the documentation so that it encompases a clear description of the strategies, which would allow a lower barrier of entry to understanding them, without requiring reviewers to look beyond the Harvest Finance system (Suggestion 5).

System Design

The Harvest Finance system is large and complex, and contains many moving parts. Specifically, the size and complexity of the Vault and various strategy classes, including super-classes, requires that extra attention be given to the surrounding code, in order to optimize the security of all methods. In particular, upgradability, controllability, and modular strategies all add to this complexity and should be continued to be closely monitored and complexity should be reduced where possible.

Farming Strategies

Harvest Finance incorporates many underlying DeFi systems. Consequently, there are complex farming strategies and many variations that must be tracked in order to farm such a large number of platforms. The differences and nuances between the underlying foundation of these systems require that the corresponding strategies be specially purposed and customized for each platform. Given the complexity of such a system, many moving pieces create the opportunity for a wider attack surface and multiple points of failure. We suggest continuing to explore and pursue opportunities to reduce complexity and facilitate an easier understanding of the system at large, which would allow reviewers to more effectively reason about and analyze the potential for security vulnerabilities.

Price Checkpoint

Two of the issues we have identified are known attacks (Issue A; Issue B). The current remediation strategy implemented by the Harvest Finance team is an economic one, using price checkpoints to prevent gains from being realized in these types of attacks. For flash loan attacks (Issue A), we recommend a technical solution with two main advantages: having no negative impact on the user experience and having no arbitrary distinction between legitimate and illegitimate prices. Furthermore, the technical remediation does not enable the attack vector (Issue C) that is made possible by the currently proposed solution. We recommend that the alternative technical remediation is given due consideration by the Harvest Finance team.

Initialization Functions

The Harvest Finance system consists of upgradeable contracts that will allow the organization or governance to swap strategies, which will keep the contract system flexible as new strategies become more or less profitable. They implement initialization functions on contracts to allow for upgradeability (e.g. the `SushiMasterChefLPStrategy.sol`), with a modifier called initializer. This modifier is a tool provided by OpenZeppelin to prevent an initialization function from running more than once. This abstraction is used in many places and is a good example of code reuse. Extra care must be taken to initialize all other contracts and their parents, as these functions are not automatically called the way in which the constructor function is called.

Governance

The Harvest Smart Contracts possess significant amounts of control over the governance mechanisms, including a salvage method. Although we assume that the Harvest Finance team is benevolent, this centralization does increase the surface for an attack. We recommend the Harvest Finance team consider multisignature where critical (e.g. strategy upgrades) and otherwise limit the scope of the Governance role.

Verification: Abandoning Virtual Prices

Following the completion of the audit and delivery of the Updated Initial Audit Report, the Harvest Finance team has decided that the virtual price model had insurmountable challenges and they would revert to the previous design. In response to our report, the Harvest team has stated their intention to discontinue the strategies which were vulnerable to price manipulations, which includes any contract that requires a conversion of the underlying asset on withdrawal. Specifically, the Harvest team has noted that this includes the following strategies: `CRVStrategyWRenBTC.sol`, `CRVStrategySwerve.sol`, `CRVStrategyStable.sol`, and their respective subclasses. We commend the Harvest Finance team for taking such significant actions to reduce complexity and, as a result, minimize the potential attack surface. However, as a result of this decision, the code that is currently running in production has not been audited by our team.

The Vault redesign that our team reviewed within the scope of this audit has not been deployed in production, thus the issues we identified are no longer applicable to the current Harvest Finance production system.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Flash Loan Attack (Known Issue)	Not Applicable
Issue B: Sandwich Attack (Known Issue)	Not Applicable
Issue C: Price Checkpoint DoS	Not Applicable
Suggestion 1: Improve Code Comments	Unresolved
Suggestion 2: Update Compiler Version	Unresolved
Suggestion 3: Expand Test Coverage	Unresolved
Suggestion 4: Improve Documentation	Unresolved

Issue A: Flash Loan Attack (Known Issue)

Location

/contracts/strategies/curve/CRVStrategyStable.sol

Synopsis

Using a flash loan, a contract is able to make a Harvest deposit while also manipulating the share price of the underlying token, and then withdraw the deposit to make a profit, and steal from the other depositors via the Harvest buffer.

Impact

Funds can be manipulated and withdrawn. A previous flash loan attack on Harvest Finance, which occured on 26 October 2020, resulted in the theft of $33.6 million worth of funds, representing 3% of the total value in the Harvest Finance protocol.

Preconditions

Harvest Finance supports strategies that invest in DeFi protocols whose price can be manipulated by a flash loan.

Feasibility

Attacks utilizing flash loans are sometimes easy, depending on the approach taken, and interactions between Harvest Finance and the DeFi protocol used by a strategy. As the Harvest Smart Contracts incorporate other DeFi protocols, it is possible that new flash loan attacks will become feasible.

Technical Details

With the previous flash loan attack, the attacker took out a flash loan, and used it to acquire a large amount of USDC and USDT. They then converted the USDT to USDC inside the curve yPool, increasing the price of USDC within yPool. The attacker then deposited a large amount of USDC into Harvest Finance. Due to the inflated price of USDC at the time, the share price relative to USDC had lowered and the attacker obtained slightly more shares than usual. They then reversed their swap of USDT back to USDC and withdrew from Harvest Finance. Since the USDC price had now returned to normal and the share price increased, they were able to sell the shares back for slightly more than they brought them. This extra amount was paid from the Harvest Finance buffer.

Remediation

Existing Remediation: Price Checkpointing

These attacks require a flash loan, which depends on multiple technical aspects of Ethereum contracts, in addition to the ability to manipulate Vault share prices and get a positive financial return. In response to the previous attack, the Harvest Finance team has designed the remediation strategy described in the Vault redesign document. When calculating a share value, instead of using the current price (which may be manipulated via a flash loan), the system compares the current price with the last price checkpoint and uses the minimum of the two in the withdraw function and the maximum of the two prices in the deposit function. This means that a price manipulation will not affect the share price. A flash loan will lose money, fail to be repaid, and be reverted.

We agree that the Vault redesign should be sufficient to prevent flash loans from being used against Harvest Finance. However, as noted in the Vault redesign document, it negatively affects user experience in that it causes a delay in transactions or a discrepancy in price:

> *"As a consequence of the virtual price calculation, if an honest user deposits funds into a vault, they should wait for at least one `doHardWork()` call before withdrawing, otherwise they will notice a slight discrepancy between the deposited amount and the amount withdrawn. "*

Consequently, honest users would need to be aware of price checkpointing being used, in order to avoid being subject to price discrepancies between the times in which `doHardWork` is called.

After exploring the issue during our review, we suggest several alternative ways flash loans could be prevented in purely technical, non-economic ways, as detailed below.

Alternative Potential Remediations

Restrict all contracts

The first possibility is to restrict all contracts, disallowing the use of a contract to invest in Harvest Finance. A flash loan can only be made out to a contract and not an Externally Owned Account (EOA) because the flash loan calls back to the recipient, who must repay the loan before it returns. An EOA can call EVM functions but cannot itself be called, thus it is unable to receive flash loans except by deploying a contract. While this remediation would prevent flash loan attacks, creating a system where no one can use a contract to invest in Harvest Finance may be seen as too severe a limitation.

Two-step process

A second strategy is to target the synchronicity attribute of flashs loans, specifically, that they must be repaid before the flash loan call returns. If a contract is not able to both deposit and withdraw within a single transaction, it would be impossible to repay the flash loan, causing it to be reverted. In a post-mortem blog post about the attack, the Harvest Finance team suggested using a commit-and-reveal mechanism for deposits. In this case, a user would transfer funds into Harvest Finance in a single step and then claim it in the next. As the Harvest Finance team points out, this has the negative side effect of changing the user experience and API by requiring two transactions, as well as increasing gas costs.

However, simply requiring that a deposit and a withdrawal are in two separate blocks would interrupt flash loans in the same way, but with the positive benefit of not altering the user experience. To implement this method of resistance to a flash loan, a contract could keep a map of the address to the block height of the last interaction with the contract and whether it was a deposit or withdrawal. If the previous interaction was before the current block height, then the call is allowed to proceed, but if the previous interaction equals the current block height, then the transaction is reverted. By requiring the withdrawal after the deposit to be made in at least the next block, it will be impossible to use the contract via a flash loan.

Return Funds At Deposit Price If Withdrawn Before `doDardWork` Call

Alternatively, there is another way to interrupt flash loans to prevent an attack. If the deposited funds are withdrawn before `doHardWork` is called, it makes sense to simply return the funds at the price paid on deposit. With a flash loan (or other price manipulation attacks), the deposit is made, then the attacker performs a price manipulation, and then withdraws. Prior to calling `doHardWork`, these funds are sitting in the Vault and they have not done any *work*. If they are withdrawn before the funds have done anything, they can simply be returned at the same price.

In the edge case where attackers deposit funds in multiple transactions during price changes but before calling `doHardWork`, a simple solution would be to record the price as the average of the two prices paid, weighted by the amounts brought in each time. This is a simple calculation:

```
price_new = (price1*deposit1+price2*deposit2)/(deposit1+deposit2)

deposit_new = deposit1+deposit2
```

If they then withdraw both deposits, it is done at `price_new`, and they get the same amount back as if they had withdrawn `deposit1` before they made `deposit2`. Although it might seem necessary to record this in a complicated data structure, this should be avoided in Ethereum contracts. The important thing is that any funds that do not enter the actual DeFi exchange are withdrawn at the same price.

Verification

While we agree that the currently implemented price-checkpoint remediation should prevent flash loan attacks, we recommend considering this last remediation option. By returning the funds at the price paid at deposit, it makes an attack uneconomical. The simplicity of using this price would not introduce the complexities of determining legitimate and illegitimate prices, and without impacting the user experience as a two step process requires. Furthermore, contracts depositing and withdrawing in the same transaction would still work, but neither making a profit nor a loss.

Status

Not Applicable. The Harvest Finance team has informed us they have discontinued the strategies which were vulnerable to price manipulations and have reverted to a previous code design that was out-of-scope for our review. As a result of this decision, the code that is currently running in production has not been audited by our team and we recommend it be reviewed for the presence of this or similar issues.

Issue B: Sandwich Attack (Known Issue)

Synopsis

We believe this type of attack first surfaced in the form of front-running attacks against exchanges. The creator of Uniswap has coined this “The Dark Forest” of Ethereum, where transaction ordering and the public nature of the blockchain create front-running buy and sell attacks. In the context of Harvest Finance, a sandwich attack is an attempt to manipulate prices and commit a similar attack to Issue A by manipulating the price before *and* after the call to `doHardWork` (which encompasses the price checkpoint), thus "sandwiching" the `doHardWork` transaction. Even if a remediation to Issue A prevents price manipulations within a single transaction, it is still possible that one is performed over multiple transactions.

As flash loans should no longer be possible after the existing remediation was implemented for Issue A, the actor carrying out a successful sandwich attack needs to be a “whale”, that is, a market participant with enough funds to make large transactions that move the market on their own. The whale knows about these market movements ahead of other participants because they create them. Therefore, they can profit from them by taking funds from other participants.

Impact

Funds can be manipulated and withdrawn.

Preconditions

A price checkpoint based mitigation to flash loans has been implemented and the attacker has enough funds to move the market on their own.

Feasibility

This attack is feasible if the attacker has a large amount of funds and sufficient understanding of the system. From the point of view of the attacker, there is also a risk associated with a sandwich attack, which is not present in the case of a flash loan based attack. There are typically no guarantees that the sandwich attack will not lose funds to arbitrage bots, since attacks cannot be conducted in the safety of one transaction if the remediation in Issue A is in place. The attacker must broadcast a transaction that gets mined before the `doHardWork` call and, since miners order transactions by gas price, this is as simple as using a higher gas price. They must also make a second transaction with a lower gas price (and there is a risk for them there that transactions will get chosen by miners instead).

Technical Details

With the price checkpoint remediation to Issue A, a new price checkpoint is set when `doHardWork` is called. A sandwich attack can be successful if an attacker is able to insert a price manipulation after this transaction is observed but before it is mined, and then append another transaction that takes advantage of the manipulation that is mined soon after the `doHardWork` transaction. The actual attack transactions would be similar to those used in Issue A, with the only difference being that they are performed in separate transactions, one inserted before the `doHardWork` call which sets the price checkpoint, and one appended as soon as possible after it.

Mitigation

The Harvest Finance team has implemented a mitigation to this attack with a New Vault Design, using a price hint that is passed to the `doHardWork` call on the `Controller` contract. When calling `doHardWork`, an expected price and acceptance ratio is checked, and if the measured price at the time the contract is mined is outside that price by more than the acceptable ratio, then the transaction is reverted. The price hint and ratio is chosen by Harvest governance before making the call. If there is an unexpected change in price before the transaction is mined, it will be reverted. Thus, the most the attacker could achieve is causing the doHardWork call to revert.

We consider this a mitigation (and not a remediation) because it reduces the impact but does not eliminate the problem. Because an acceptance ratio is used, an attacker may still profit if their attack is small enough. The documented flash loan attack used only a 1% price manipulation, while still extracting a large amount, so this seems possible. This mitigation also has the downside that Harvest Finance now carries the responsibility to approve prices. Choosing an acceptance ratio that is too small makes it easy to block `doHardWork` (see Issue C), but choosing an acceptance ratio that is too high will make it easier to make small but still profitable manipulations.

To strengthen this mitigation, it may be worth considering making the `doHardWork` call with a low gas price. Paradoxically, this will make it more difficult to sandwich. It will still be easy to insert a transaction before the doHardWork call, however, appending a transaction after it becomes more risky. Most Ethereum users prefer to save money using the average gas price. If there are many other transactions in the mempool using the same price, it becomes harder to insert the bottom layer of the sandwich. If miners must choose between many transactions, then it's possible that transaction is bumped to the next block. This significantly increases the arbitrage risk for the attacker.

Remediation

Direct Withdrawals

Another approach to fixing the sandwich attack would be to avoid tracking the price entirely. In the Harvest Finance post mortem blog post, a design is discussed whereby instead of calculating a share price based on the current market price, the user simply withdraws the underlying asset and then exchanges it themselves. By not involving the price, this completely avoids price manipulations. Notably, this approach would be a very significant change to Harvest Finance.

This remediation could be improved by automating the exchange, but by using the actual exchange contract rather than calculating an exchange rate. The documented attack used 17 million units to manipulate the price, causing a 1% price manipulation, then withdrew 50 million units. However, if this 50 million unit withdrawal went through the actual exchange contract, it would have just caused another price manipulation in the opposite direction. The outcome would depend on how the DeFi exchange in question operates, but it would likely not be profitable for the attacker.

Random ordering of Ethereum transactions

The sandwich attack is possible because the ordering of Ethereum transactions is deterministic. Transactions are selected by highest gas price, and then ordered from highest to lowest price in the block. This makes front running possible. If miners chose to order transactions randomly, it would not be possible to perform this attack, except as a miner. The ordering of transactions is not fixed in the Ethereum specification, it is simply the default behavior of the clients to order them from highest to lowest price. Changing this would not require a hard fork, but it would require campaigning to have this change made to clients and adopted by miners. This would prevent sandwich attacks for Harvest Finance, and also eliminate front running attacks for a large number of Ethereum contracts. This is not a short term fix, but it is also not mutually exclusive to the mitigation already implemented or the direct withdrawal remediation suggested above.

Verification

We understand that possibly neither of these suggested remediations are immediately viable for the Harvest Finance team, however, we encourage these to be considered as the system and the DeFi ecosystem on Ethereum continue to grow.

Status

Not Applicable. The Harvest Finance team has informed us they have discontinued the strategies which were vulnerable to price manipulations and have reverted to a previous code design that was out-of-scope for our review. As a result of this decision, the code that is currently running in production has not been audited by our team and we recommend it be reviewed for the presence of this or similar issues.

Issue C: Price Checkpoint DoS

Location

/contracts/Controller.sol#L56-L76

Synopsis

This issue arises from the mitigation strategy of the Vault redesign for sandwich attacks, specifically the `Controller` contract calling `doHardWork` and the control over the expected slippage of price from the `Controller`. If an attacker is able to predict the acceptable slippage of the `Controller`, they may attempt to front-run every `Controller` call to `doHardWork`. The attacker must manipulate the price of the assets such that the `Controller` transaction will revert for being outside of the acceptable limit. The governance of Harvest Finance will attempt to reduce the allowed slippage percentage as much as possible to prevent high slippage manipulations, but in doing so, they will make it easier to block `doHardWork` price checkpoints by making the percentage of slippage small and minimizing the amount of manipulation needed to block the call.

Impact

An incorrect recorded price for an extended period of time could cause any deposits after the attack to be withdrawn at an incorrect rate. However, if the last remediation that we propose in Issue A is implemented, there will no longer be a price checkpoint that a DoS attack can affect. Such an attack would only block the call to `doHardWork`, which should reduce the profitability of engaging in a checkpoint DoS attack.

Preconditions

The attacker must be able to front-run the `Controller`'s attempt to call `doHardWork`. They must also be able to manipulate the market to place the price hint outside of an acceptable range. Additionally, they must be able to tolerate potential losses due to arbitrage for manipulating the price many times in a row.

Feasibility

While this attack is feasible, it presents risks to the attacker that may make it much more difficult to carry out, since it requires both the capital to move prices frequently and the ability to time the sandwich in a way that reduces the risk of arbitrage losses. It is also not clear how the attacker will weigh the cost of the attack against the profit, as profit must be made from a stale price being recorded over a period of time where prices will fluctuate naturally outside of the control of the attacker. This would require constant scanning of the mempool to catch calls to `doHardWork`, similar to the sandwich attack discussed in Issue B, and would require a good deal of expenditure in transaction fees to always order before Harvest Finance transactions. Generally, there is no guarantee that an attacker will have their transactions ordered correctly and the Harvest Finance team intends to obfuscate the time when `doHardWork` is called, making it more difficult for selfish mining to ensure proper ordering.

Technical Details

The Vault redesign document states:

> *"As a consequence of the virtual price calculation, if an honest user deposits funds into a vault, they should wait for at least one `doHardWork()` call before withdrawing, otherwise they will notice a slight discrepancy between the deposited amount and the amount withdrawn."*

If the share price hint is small, this attack will be easier, but it is still possible to block `doHardWork` calls if the attacker can tolerate moving the market price with a large amount of funding. The attacker would need to make a deposit at a normally recorded price from a previous `doHardWork` call. Assuming that the value of the shares fluctuates, the attacker would then continuously sandwich attack the `Controller` with a price slippage that is outside of its acceptable range. This will effectively block the call to `doHardWork` and any update to the recorded price used to prevent flash loan attacks. This blockage will lock the price in at an outdated value and increase the discrepancy mentioned in the Vault redesign document. As more deposits enter the strategy, this price discrepancy could be in favor of the early deposit from the attacker.

Mitigation

While there are currently no industry standard best practices to address sandwich attacks, in this case, the attack can only succeed if the attacker is able to perpetually block the price update at the `Controller` level. Harvest Finance could always call `doHardWork` with a high slippage amount, but a front-running bot could detect this and commit a normal sandwich attack with a high slippage. In addition, any mitigations for the previous sandwich attack (Issue B) should help in preventing this price checkpoint DoS attack.

Controlling or purchasing a miner that can discover blocks at a rate that `doHardWork` needs to be called could fully remediate this issue. Clearing a full block to ensure that no sandwich attacks will throw the expected price out of range will ensure that the Harvest Finance team can have a successful transaction to the `Controller` contract and update the price checkpoint. However, the cost for doing so may be prohibitive to the organization and, as a result, this approach should be only considered as a potential mitigation.

We recommend that the Harvest Finance team further explore the potential alternative remediations outlined in Issue A, specifically, the remediation that will return the funds to the attacker without affecting the pool if it is called before `doHardWork`. This will prevent flash loan attacks and will potentially remove further profits that could be gained by delaying the price checkpoint, as there will be no need to price checkpoint within doHardWork, thus reducing the incentive to DoS this function.

Status

Not Applicable. The Harvest Finance team has informed us they have discontinued the strategies which were vulnerable to price manipulations and have reverted to a previous code design that was out-of-scope for our review. As a result of this decision, the code that is currently running in production has not been audited by our team and we recommend it be reviewed for the presence of this or similar issues.

Additionally, we note that the `Vault.sol` still has an option `allowSharePriceDecrease` which, if set to false, causes the doHardWork function to revert if the share price decreases. This means that an attacker could block doHardWork, but only by manipulating the share price down. The Harvest Finance Governance can unset this option.

Suggestions

Suggestion 1: Improve Code Comments

Location

Examples:

/contracts/strategies/SNXRewards/SushiMasterChefLPStrategy.sol#L14-L45

/contracts/strategies/SNXRewards/SushiMasterChefLPStrategy.sol#L59

/contracts/strategies/SNXRewards/SushiMasterChefLPStrategy.sol#L83

Synopsis

Code comments were absent in several areas of the code base and descriptive comments that do exist would benefit from further clarification. At present, most functions do not follow the style guidelines for Solidity, the NatSpec Format, and the comments do not provide any information to the inputs or goals of each function.

Mitigation

We recommend that code comment coverage be expanded, existing comments be edited for clarity, and all functions be updated to follow Solidity style guidelines.

Verification

The suggestion mitigation has not been implemented.

Status

Unresolved.

Suggestion 2: Update Compiler Version

Location

/contracts

Synopsis

The compiler version is set to version 0.5.16, which does not incorporate newer compiler fixes and updates.

Mitigation

We suggest updating the compiler to version 0.7.0 or higher, but no higher than version 0.7.4 at this current time. Version 0.7.4 is a more recent release than what is currently being used by the Harvest Finance team, however, it is not the most recent release. We advise against using the latest release as it may contain a higher probability of unknown issues.

Verification

The suggested mitigation has not been implemented.

Status

Unresolved.

Suggestion 3: Expand Test Coverage

Location

[/test]

Synopsis

There is a large test suite which covers many simple and happy path test cases. However, the code base would benefit from increasing test coverage with specific attention to revert cases and strategy integration tests. Comprehensive test coverage helps to identify simple errors and prevents functionality from breaking when new code changes are introduced.

Furthermore, tests are currently implemented in JavaScript. TypeScript has become an industry standard for tests as it facilitates development speed and helps to eliminate common classes of bugs found in pure JavaScript, where types are non-existent.

Mitigation

We recommend expanding test coverage for revert cases and strategy integration tests. In addition, we recommend implementing tests in TypeScript instead of using JavaScript.

Verification

The suggested mitigation has not been implemented.

Status

Unresolved.

Suggestion 4: Improve Documentation

Location

/VaultRedesign.MD

Synopsis

The Vault redesign was helpful in that it provides insight into the previous attack, along with the mitigation and its side effects that was implemented by the Harvest Finance team. The document would further benefit from examples, similar to what has been provided by the Harvest Finance team in the post mortem blog post.

In order to gain a comprehensive understanding of strategies, and since strategies use other DeFi systems, an understanding of those systems facilitates the ability to review and understand the strategies. As a result, clear documentation describing the strategies would allow for an easier understanding of the system at large, without requiring reviewers to look beyond the Harvest Finance system.

Mitigation

We recommend increasing the level of detail in the Vault redesign documentation so that it includes examples. In addition, we recommend improving the documentation so that it includes a clear description of the various strategies.

Verification

The suggested mitigation has not been implemented.

Status

Unresolved.

Recommendations

We recommend that the currently implemented code base be audited for the *Issues* and that the *Suggestions* stated above are addressed as soon as possible and followed up with the necessary verification.

Although the previous design and remediation strategy implemented by the Harvest Finance team for flash loan attacks (Issue A) should be suitable, it is an economic one, using price checkpoints to prevent gains from being realized in these types of attacks. We recommend the Harvest Finance team implement the suggested technical solution with two main advantages, including having no negative impact on the user experience and having no arbitrary distinction between legitimate and illegitimate prices. Furthermore, this technical remediation does not enable the attack vector noted as Issue C, which is currently possible. We recommend that the alternative technical remediation is given due consideration by the Harvest Finance team and resolved if acceptable.

However, these recommendations are not applicable. The Harvest Finance team has informed us they have discontinued the strategies which were vulnerable to price manipulations and have reverted to a previous code design that was out-of-scope for our review. As a result of this decision, the code that is currently running in production has not been audited by our team and we recommend it be immediately reviewed for the presence of this or similar issues to the current production implementation

In addition, we recommended improving code comments and ensuring they conform to the Solidity guidelines, thus providing more clarity on intended behavior of the individual components for users and reviewers of the code. Furthermore, expanding test coverage to revert cases, strategy integration, and expected failures would help identify errors and bugs in the code while maintaining up-to-date versions of dependencies would reduce the probability of unknown issues being introduced by them.

Finally, we commend the Harvest Finance team for their continued diligence and focus on security by engaging in and publishing several security audits in recent months. We recommend that they continue to engage in independent security reviews of the Harvest Finance system, which incorporate all components of the system that are considered to be security critical.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.