Ninja Yield Security Review

Pashov Audit Group

Conducted by: pashov

December 6th, 2022

Contents

1. About pashov

Krum Pashov, or **pashov**, is an independent smart contract security researcher. Having found numerous security vulnerabilities in various protocols, he does his best to contribute to the blockchain ecosystem and its protocols by putting time and effort into security research & reviews. Check his previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **Ninja Yield** protocol was done by **pashov**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Ninja Yield

The code was reviewed for a total of 6 hours.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hash **-** **9e9367120d45fdd6144328964fabb8f57610661c**

7. Executive Summary

Over the course of the security review, pashov engaged with Ninja Yield to review Ninja Yield. In this period of time a total of **7** issues were uncovered.

Protocol Summary

Protocol Name	Ninja Yield
Date	December 6th, 2022

Findings Count

Severity	**Amount**
High	2
Medium	2
Low	3
Total Findings	**7**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Incorrect user accounting in withdraw method	High	Resolved
[H-02]	First vault depositor can steal subsequent depositors’ tokens	High	Resolved
[M-01]	MEV can sandwich every harvest due to missing slippage tolerance value	Medium	Resolved
[M-02]	Hardcoded swap path might not be the most optimal/liquid one	Medium	Resolved
[L-01]	Missing nonReentrant modifier in functions with external calls	Low	Resolved
[L-02]	If underlying or rewardToken is a two-address token then inCaseTokensGetStuck method can be used to rug users	Low	Resolved
[L-03]	The getPricePerFullShare method returns a wrong value when totalSupply is 0	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Incorrect user accounting in `withdraw` method

Proof of Concept

The `withdraw` method in `NYProfitTakingVaultBaseV1` does incorrect user accounting in the following line:

```
user.amount = user.amount - _shares;
```

In the `deposit` method we use the `user.amount` to store the amount of `underlying` tokens deposited, but in `withdraw` instead of subtracting the `underlying` tokens the code subtracts the shares burned.

Impact

Since `shares` are not 1:1 with `underlying` this will completely mess up the user accounting on each withdraw. It is possible to be in two directions - if `_shares` was less than the amount withdrawn, then the user will be able to withdraw more than he deposited, essentially a possibility to deplete the vault to zero. If `_shares` was more than the amount withdrawn, then the user will be able to withdraw less than he deposited, essentially a loss of value for users.

Recommendation

Make the following change:

```diff
- user.amount = user.amount - _shares;
+ user.amount = user.amount - r;
```

[H-02] First vault depositor can steal subsequent depositors' tokens

Proof of Concept

Imagine the following scenario:

1. A new vault has been deployed and configured, no depositors yet
2. Alice wants to deposit 10 ether(10e18) worth of `underlying` and sends a transaction to the public mempool
3. A MEV bot sees Alice's transaction and front runs it by depositing 1 wei(1e-18) of `underlying`, resulting in him receiving 1 wei(1e-18) of vault tokens (shares)
4. The MEV bot also front runs Alice's transaction with a transfer of 10 ether(10e18) of `underlying` to the vault via `ERC20::transfer`
5. Now the code calculates Alice's shares as `shares = (_amount * totalSupply()) / _pool;` which is 10e18 * 1 / (10e18 + 1) which is 0
6. Alice gets minted 0 shares, but she deposited 10e18 of `underlying`
7. Now the MEV bot backruns Alice's transaction calling `withdraw` with his 1e-18 (1 wei) of share, which is the total supply, so he withdraws his deposit + Alice's whole deposit

This can be replayed multiple times until the depositors notice the problem.

Impact

The result of this is 100% value loss for all subsequent depositors.

Recommendation

UniswapV2 fixed this with two types of protection:

First, on the first `mint` it actually mints the first 1000 shares to the zero-address

Second, it requires that the minted shares are not 0

Implementing them both will resolve this vulnerability.

8.2. Medium Findings

[M-01] MEV can sandwich every harvest due to missing slippage tolerance value

Proof of Concept

In `NyPtvFantomWftmBooSpookyV2StrategyToUsdc` each time the `_harvestCore` method is called (on each harvest) it will call the `_swapFarmEmissionTokens` method which itself has the following code:

```
IUniswapV2Router02
  (SPOOKY_ROUTER).swapExactTokensForTokensSupportingFeeOnTransferTokens(
      booBalance,
      0,
      booToUsdcPath,
      address(this),
      block.timestamp
    );
```

The “0” here is the value of the `amountOutMin` argument which is used for slippage tolerance. 0 value here essentially means 100% slippage tolerance. This is a very easy target for MEV and bots to do a flash loan sandwich attack on each of the strategy’s swaps, resulting in very big slippage on each trade.

Impact

100% slippage tolerance can be exploited in a way that the strategy (so the vault and the users) receive much less value than it should had. This can be done on every trade if the trade transaction goes through a public mempool.

Recommendation

The best solution here is to make the `harvest` method of the vault be callable only by a list of trusted addresses which will send the transaction through a private mempool. This, combined with an on-chain calculation for an `amountOutMin` that is off from the expected `amountOut` by a slippage tolerance percentage (that might be configurable through a setter in the strategy) should be good enough to protect you from MEV sandwich attacks.

[M-02] Hardcoded swap path might not be the most optimal/liquid one

Proof of Concept

Currently in `NyPtvFantomWftmBooSpookyV2StrategyToUsdc` the value of the `booToUsdcPath` trade path is not configurable and is basically hardcoded to be `[BOO, USDC]`. It is the same for the swap router, as it is currently hardcoded to point to the SpookySwap router. The problem is that the `BOO/USDC` pool on SpookySwap might not be the most optimal and liquid one, and maybe instead it would be better to go `BOO/USDT` and then `USDT/USDC`. If the `BOO/USDC` pair for example loses most of its liquidity (maybe LPs are not incentivised as much or they decided to move elsewhere) then the strategy will still be forced to do its swaps on `harvest` through the illiquid/non-optimal `BOO/USDC` pair on SpookySwap.

Impact

This can result in a loss of value for vault users, as if a more liquid pool was used for swaps it could have resulted in less slippage so a bigger reward.

Recommendation

Add setter functions for both the trade router and the trade path - make them configurable. One possible option is to hardcode the 3 most liquid Fantom exchanges and 3 possible trade paths and switch through them via the setter.

8.3. Low Findings

[L-01] Missing `nonReentrant` modifier in functions with external calls

Proof of Concept

The `NYProfitTakingVaultBaseV1` contract inherits from OpenZeppelin's `ReentrancyGuard` contract and has marked most of its state-changing methods that do ERC20 external calls with the `nonReentrant` modifier. The problem is the modifier is missing on some of the functions that also do ERC20 external calls - the `depositOutputTokenForUsers` and `earn` methods. Currently the methods are not exploitable, but ERC777 tokens (which are ERC20 compatible) can reenter a method call because of their pre and post hooks, so the `nonReentrant` modifier is an important security measure when doing unsafe ERC20 external calls.

Impact

If ERC777 tokens were used as `rewardToken` or `underlying` they can reenter the `depositOutputTokenForUsers` and `earn` methods, which currently is not exploitable, but might become a big problem when new code is added.

Recommendation

Add the `nonReentrant` modifier to the `depositOutputTokenForUsers` and `earn` methods

[L-02] If `underlying` or `rewardToken` is a two-address token then `inCaseTokensGetStuck` method can be used to rug users

Proof of Concept

Some ERC20 tokens on the blockchain are deployed behind a proxy, so they have at least 2 entrypoints (the proxy and the implementation) for their functionality. Example is Synthetix's `ProxyERC20` contract from where you can interact with `sUSD, sBTC etc). If such a token was used as the underlying token in a vault, then the owner will be able to rug all depositors with the`inCaseTokensGetStuck` method, even though it has the following checks

```
if (_token == address(underlying)) {
      revert NYProfitTakingVault__CannotWithdrawUnderlying();
}
if (_token == address(rewardToken)) {
      revert NYProfitTakingVault__CannotWithdrawRewardToken();
}
```

Since the tokens have multiple addresses the admin can give another address and pass those checks.

Impact

The potential impact is 100% loss of deposited tokens for users, but it requires a malicious/compromised owner and a special type of ERC20 token used in the vault.

Recommendation

Instead of checking the address of the withdrawn token, it is a better approach to check the balance of `underlying` and `rewardToken` before and after the transfer and to verify it is the same.

[L-03] The `getPricePerFullShare` method returns a wrong value when `totalSupply` is 0

Proof of Concept

The `getPricePerFullShare` method currently computes the result by the following formula:

```
return totalSupply() == 0 ? 1e18 : (balance() * 1e18) / totalSupply();
```

The problem is that when the underlying token used is with less or more than 18 decimals and the `totalSupply` is still 0, then the price returned won't be correct, as it will be 1e18. The intention to return 1e18 when `totalSupply` is 0 looks like it comes from the amount of shares minted on the first deposit

```
if (totalSupply() == 0) {
      shares = _amount;
}
```

so it looks like 1 share will equal 1 token, but if the token's decimals are not 18 then 1 token is not 1e18 wei of this token.

Impact

Since this function is market with `public view` and is not used anywhere in the protocol, it will probably be used in front ends only, impacting the initial pricing of a vault share.

Recommendation

Make the following change:

```
- return totalSupply() == 0 ? 1e18 : (balance() * 1e18) / totalSupply();

+ return totalSupply() == 0 ? 10**underlying.decimals() : (balance
+ () * 1e18) / totalSupply();
```