SMART CONTRACT AUDIT REPORT

for

YEARN.FINANCE

Prepared By: Shuxiao Wang

Hangzhou, China

January 17, 2021

Document Properties

Client	Yearn.Finance
Title	Smart Contract Audit Report
Target	Hegic Strategies
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 17, 2021	Xuxian Jiang	Final Release
1.0-rc2	December 31, 2020	Xuxian Jiang	Release Candidate #2
1.0-rc1	December 27, 2020	Xuxian Jiang	Release Candidate #1
0.2	December 21, 2020	Xuxian Jiang	Additional Findings
0.1	December 18, 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 design document and related smart contract source code of `Hegic Strategies` in the `Yearn.Finance` protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Yearn.Finance/Hegic Strategies

`Yearn.Finance` is a yield aggregating platform on Ethereum. It has grown into an ecosystem of protocols that aims to maximize annual percentage yields (APY) for its users. Specifically, it utilizes a number of yield-generating DeFi protocols such as `Curve`, `Compound`, `Aave`, and `dYdX` to optimize token lending. In a nutshell, it is a sophisticated protocol that diverts liquidity to different sectors of the DeFi universe and provides its users with access to the highest yields on deposits of ether, stablecoins, and altcoins. The audited four `Hegic strategies` are new additions that make use of the popular `Hegic` protocol to provide new yielding opportunities with minimized risk.

The basic information of the Hegic Strategies is as follows:

Table 1.1: Basic Information of Hegic Strategies

Item	Description
Issuer	Yearn.Finance
Website	https://yearn.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 17, 2021

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit.

- https://github.com/Macarse/yhegic.git (5376205)
- https://github.com/Grandthrax/YearnV2-Generic-Lev-Comp-Farm.git (1a30040)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/Macarse/yhegic.git (5f95f8e)
- https://github.com/Grandthrax/YearnV2-Generic-Lev-Comp-Farm.git (eba1a94)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;

- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further 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) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s), 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

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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of Hegic Strategies. 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	3	■ ■ ■
Low	2	■ ■
Informational	3	■ ■ ■
Total	9	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 3 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key Audit Findings of Hegic Strategies

ID	Severity	Title	Category	Status
PVE-001	Informational	Asset Consistency Check Between Vault And Strategy	Coding Practices	Resolved
PVE-002	Medium	Improved protectedTokens() For sweep() Exclusion	Business Logic	Resolved
PVE-003	Low	Unclaimed Profits in prepareMigration() And exitPosition()	Business Logic	Resolved
PVE-004	Low	Possible Front-Running For Reduced Return/Profit	Time and State	Resolved
PVE-005	Medium	Potential Denial-of-Service in _withdrawSome() And exitPosition()	Business Logic	Resolved
PVE-006	Informational	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Resolved
PVE-007	High	Business Logic Error in StrategyWbtcHegicLP::adjustPosition()	Business Logic	Resolved
PVE-008	Informational	Redundant Code Removal in Multiple Strategies	Coding Practices	Resolved
PVE-009	Medium	Business Logic Error in Strategy::adjustPosition()	Business Logic	Resolved

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 Asset Consistency Check Between Vault And Strategy

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-1099 [1]

Description

For each audited Hegic-related strategy, there is a one-to-one mapping with the related vault. To properly link a vault with its strategy, it is natural for the two to operate on the same underlying asset. For example, StrategyHegicETH allows for HEGIC-based deposits and withdraws for investment. The strategy naturally has HEGIC as the underlying want asset. If these two are different, the strategy should not be successful.

```
contract StrategyHegicETH is BaseStrategy {
    using SafeERC20 for IERC20;
    using Address for address;
    using SafeMath for uint256;

    address public hegic;
    address public hegicStaking;
    uint256 public constant LOT_PRICE = 888e21;
    address public unirouter;
    string public constant override name = "StrategyHegicETH";

    constructor(
        address _vault,
        address _hegic,
        address _hegicStaking,
        address _unirouter
    ) public BaseStrategy(_vault) {
        hegic = _hegic;
        hegicStaking = _hegicStaking;
```

```
        unirouter = _unirouter;
        IERC20(hegic).safeApprove(hegicStaking, uint256(-1));
    }
```

Listing 3.1: StrategyHegicETH::constructor()

For elaboration, we show above the `constructor()` routine of the `StrategyHegicETH` contract. It is important to ensure the deposited `HEGIC` asset is consistent with the expected `want` asset in receiving strategies for investment. In other words, it is suggested to impose another requirement, i.e., `require(hegic == want)`. By doing so, we can ensure users may not mistakenly sent wrong assets for investment.

The same issue is applicable to both `StrategyHegicETH` and `StrategyHegicWBTC` contracts.

Recommendation Ensure the consistency of the underlying asset within each new strategy.

Status The issue has been fixed in this commit: `8135d28`.

3.2 Improved protectedTokens() For sweep() Exclusion

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

The supported strategies in `Yearn.Finance` allow users to invest their assets for yields and gains with high-assurance. Since its deployment, `Yearn.Finance` has gained increasing popularity and adoption. In the meantime, we notice that there is always non-trivial possibilities that non-related tokens may be accidentally sent to various `Yearn.Finance` contracts. To avoid unnecessary loss of `Yearn.Finance` users, the base `BaseStrategy` contract provides the necessary support of rescuing tokens accidentally sent to the contract. This is a design choice for the benefit of `Yearn.Finance` users.

To elaborate, we show below the code snippet of the `sweep()` routine in `BaseStrategy`. This routine is tasked with rescuing these non-related tokens accidentally sent to the strategy contract.

```
    /**
     * @notice
     *  Removes tokens from this Strategy that are not the type of tokens
     *  managed by this Strategy. This may be used in case of accidentally
     *  sending the wrong kind of token to this Strategy.
     *
     *  Tokens will be sent to `governance()`.
     *
```

```
     *  This will fail if an attempt is made to sweep `want`, or any tokens
     *  that are protected by this Strategy.
     *
     *  This may only be called by governance.
     * @dev
     *  Implement `protectedTokens()` to specify any additional tokens that
     *  should be protected from sweeping in addition to `want`.
     * @param _token The token to transfer out of this vault.
     */
    function sweep(address _token) external onlyGovernance {
        require(_token != address(want), "!want");
        require(_token != address(vault), "!shares");

        address[] memory _protectedTokens = protectedTokens();
        for (uint256 i; i < _protectedTokens.length; i++) require(_token !=
            _protectedTokens[i], "!protected");

        IERC20(_token).transfer(governance(), IERC20(_token).balanceOf(address(this)));
    }
```

Listing 3.2: BaseStrategy::sweep()

The routine has a rather straightforward execution logic in excluding related tokens (defined via `protectedTokens()`) and sending those non-related tokens back to `governance()`. However, if we examine the `protectedTokens()` support in the four new strategies, such support needs to be improved in accurately defining the set of tokens for exclusion.

Using the `StrategyHegicWBTC` as an example, we show below its `protectedTokens()` routine

```
44    function protectedTokens() internal override view returns (address[] memory) {
45        address[] memory protected = new address[](3);
46        protected[0] = address(want);
47        protected[1] = hegic;
48        protected[2] = hegicStaking;
49        return protected;
50    }
```

Listing 3.3: StrategyHegicWBTC::protectedTokens()

This routine duplicates the `HEGIC` token (line 47) and lacks the much desired `WBTC` token. Similarly, the `StrategyHegicETH` duplicates the `HEGIC` token as well. The remaining two strategies, i.e., `StrategyEthHegicLP` and `StrategyEthHegicLP`, miss the `want` token in `protectedTokens()`.

Recommendation Properly address the above issue by improving `protectedTokens()` routine in all supported four strategies for `sweep()` exclusion.

Status The issue has been fixed in this commit: `8157b78`.

3.3 Unclaimed Profits in prepareMigration() And exitPosition()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

The base strategy contract has defined a set of standard interfaces that need to be followed by every `Yearn.Finance` strategy. Specifically, `BaseStrategy` implements all of the required functionality to interoperate closely with the vault contract. This `BaseStrategy` contract should be inherited and the standard interfaces or methods implemented to adapt the strategy to the particular needs it has to create a return.

Among the set of interfaces or methods, we examine below two of them, i.e., `prepareMigration()` and `exitPosition()`. The first one prepares this strategy for migration, such as transferring any reserve or LP tokens, CDPs, or other tokens or stores of value, to the new strategy while the second one divests in all necessary means to make as much capital as possible "free" for the linked vault to take.

To elaborate, we show below the `prepareMigration()` routine in `StrategyHegicETH`. In essence, it migrates two types of assets – `want` and `hegicStaking`. However, it completely leaves any possible unclaimed gains behind.

```
    function prepareMigration(address _newStrategy) internal override {
        want.transfer(_newStrategy, balanceOfWant());
        IERC20(hegicStaking).transfer(_newStrategy, IERC20(hegicStaking).balanceOf(
            address(this)));
    }
```

Listing 3.4: StrategyHegicETH::prepareMigration()

The same issue is also applicable to other routines, including `StrategyHegicWBTC::prepareMigration()`, `StrategyEthHegicLP::prepareMigration()`, `StrategyEthHegicLP::exitPosition()`, `StrategyWbtcHegicLP::prepareMigration()`, and `StrategyWbtcHegicLP::exitPosition()`.

Recommendation Properly claim those uncollected gains in all these four strategies. Note that the `exitPosition()` routine in `StrategyHegicETH` and `StrategyHegicWBTC` are not affected.

Status The issue has been fixed in this commit: `4b501e4`.

3.4 Possible Front-Running For Reduced Return/Profit

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [4]

Description

As mentioned in Section 3.1, four new strategies have been designed and implemented to invest farmers' assets in `Hegic` and harvest growing yields. To elaborate, we show below the `prepareReturn()` routine from the `StrategyHegicETH` strategy. By calling this routine, the strategy can collect any pending rewards and swap them to the designated `want` token for the next round of investment.

```
    function prepareReturn(uint256 _debtOutstanding) internal override returns (uint256 _profit, uint256 _loss, uint256 _debtPayment) {
        // We might need to return want to the vault
        if (_debtOutstanding > 0) {
            uint256 _amountFreed = liquidatePosition(_debtOutstanding);
            _debtPayment = Math.min(_amountFreed, _debtOutstanding);
        }

        uint256 balanceOfWantBefore = balanceOfWant();

        // Claim profit only when available
        uint256 ethProfit = ethFutureProfit();
        if (ethProfit > 0) {
            IHegicStaking(hegicStaking).claimProfit();
            _swap(address(this).balance);
        }

        // Final profit is want generated in the swap if ethProfit > 0
        _profit = balanceOfWant().sub(balanceOfWantBefore);
    }
```

Listing 3.5: StrategyHegicETH::prepareReturn()

```
    function _swap(uint256 _amountIn) internal returns (uint256[] memory amounts) {
        address[] memory path = new address[](2);
        path[0] = address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2); // weth
        path[1] = address(want);

        return Uni(unirouter).swapExactETHForTokens{value: _amountIn}(_amountIn, path, address(this), now.add(1 days));
    }
```

Listing 3.6: StrategyHegicETH::_swap()

We notice the collected yields are routed to `UniswapV2` in order to swap them to `want` as rewards. 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. A similar issue also exists in the `prepareReturn()` routine of other `strategies`.

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 sandwich attack to better protect the interests of farming users.

Status This issue has been confirmed. However, as mentioned earlier, the front-running attack is inherent in current DEXes and there is still a need to search for more effective countermeasures. And the team has decided to change the swap path from `wbtc -> dai -> hegic` to `wbtc -> weth -> hegic`.

3.5 Potential Denial-of-Service in _withdrawSome() And exitPosition()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

`Hegic` is an on-chain peer-to-pool options trading protocol built on Ethereum. The pool has well-defined APIs that allow for liquidity providers ("writers") to efficiently add or remove funds. By doing so, funds from liquidity providers can be distributed among many hedge contracts simultaneously. It not only diversifies the liquidity allocation and makes efficient use of funds in the pool, but collectively shares the associated risks from one particular writer to all active liquidity providers.

The four new audited strategies rely on the `Hegic` protocol. Especially, two defined interfaces in

BaseStrategy, i.e., _withdrawSome() and exitPosition(), will interact with Hegic to withdraw all funds invested so far. Accordingly, the functionality of these two interfaces depend on the the defined APIs in Hegic. Our analysis with Hegic shows that its pool management mainly provide provide() and withdraw(): The provide() routine is used to add funds into the pool while the withdraw() routine is used to withdraw funds from the pool. In the meantime, Hegic imposes a lockup period for new funds into the pool. Specifically, for each liquidity provider, the associated lockup period is recorded as [lastProvideTimestamp[account], lastProvideTimestamp[account].add(lockupPeriod)]. Moreover, when any transfer() or transferFrom() action occurs, there is an accompanying lockup verification routine, i.e., _beforeTokenTransfer(). In the following, we outline the code logic of the three related functions: provide(), withdraw(), and _beforeTokenTransfer().

```
    function withdraw(uint256 amount, uint256 maxBurn) external returns (uint256 burn) {
        require(
            lastProvideTimestamp[msg.sender].add(lockupPeriod) <= block.timestamp,
            "Pool: Withdrawal is locked up"
        );
        require(
            amount <= availableBalance(),
            "Pool Error: Not enough funds on the pool contract. Please lower the amount."
        );
        burn = amount.mul(totalSupply()).div(totalBalance());

        require(burn <= maxBurn, "Pool: Burn limit is too small");
        require(burn <= balanceOf(msg.sender), "Pool: Amount is too large");
        require(burn > 0, "Pool: Amount is too small");

        _burn(msg.sender, burn);
        emit Withdraw(msg.sender, amount, burn);
        msg.sender.transfer(amount);
    }
```

Listing 3.7: HegicETHPool.sol

```
    function withdraw(uint256 amount, uint256 maxBurn) external returns (uint256 burn) {
        require(
            lastProvideTimestamp[msg.sender].add(lockupPeriod) <= block.timestamp,
            "Pool: Withdrawal is locked up"
        );
        require(
            amount <= availableBalance(),
            "Pool Error: Not enough funds on the pool contract. Please lower the amount."
        );
        burn = amount.mul(totalSupply()).div(totalBalance());

        require(burn <= maxBurn, "Pool: Burn limit is too small");
        require(burn <= balanceOf(msg.sender), "Pool: Amount is too large");
        require(burn > 0, "Pool: Amount is too small");
```

```
102
103         _burn(msg.sender, burn);
104         emit Withdraw(msg.sender, amount, burn);
105         msg.sender.transfer(amount);
106     }
```

Listing 3.8: HegicETHPool.sol

```
194     function _beforeTokenTransfer(address from, address to, uint256) internal override {
195         if (
196             lastProvideTimestamp[from].add(lockupPeriod) > block.timestamp &&
197             lastProvideTimestamp[from] > lastProvideTimestamp[to]
198         ) {
199             require(
200                 !_revertTransfersInLockUpPeriod[to],
201                 "the recipient does not accept blocked funds"
202             );
203             lastProvideTimestamp[to] = lastProvideTimestamp[from];
204         }
205     }
```

Listing 3.9: HegicETHPool.sol

By examining these three routines, we identify a possible front-running attack that may block an ongoing withdrawal attempt. Specifically, when a `transfer()` or `transferFrom()` action occurs, the lockup period of the receiver, i.e.,`lastProvideTimestamp[to]`, might be accordingly updated (line 203). Therefore, upon the observation of a `withdraw()` attempt from a victim, a malicious actor could intentionally transfer 1 `WEI` to the victim. By doing so, the `lastProvideTimestamp` of the victim is updated with the `lastProvideTimestamp` of the malicious actor. As a result, the specific `withdraw()` attempt is blocked as it occurs in the lockup period (line 90). We emphasize this attack will not work for those victims who do turn on the `lastProvideTimestamp` flag. However, most victims likely will not turn the flag on since it requires an extra transaction to achieve that.

In addition, the staking support in Hegic Strategies (implemented in `HegicStakingETH` and `HegicStakingWBTC`) shares a similar issue as the $address(0)$ could be contaminated, hence blocking all `buy()` attempts from legitimate stakers who turn on the `_revertTransfersInLockUpPeriod` flag. Note this attack does not work for victims who have not turned the flag on, which is contrary to the pool case.

Recommendation A mitigation to the above front-running attacks need to turn on (the pool front-running) or off (the stake front-running) the victim's flag, i.e., `_revertTransfersInLockUpPeriod`. By doing so, we can prevent the `lastProvideTimestamp` flag from being manipulated by others.

Status After the discussion, the team considers that the cost for the malicious actor to launch the denial-of-service is non-trivial (and economically not feasible) and therefore decides to leave it as is.

3.6 Improved Precision By Multiplication And Division Reordering

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact:Low
- Target: `Multiple Contracts`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

Among the new four strategies, two of them have defined a `calculateRate()` routine to calculate rewards rate in (invested) tokens per year from them. For illustration, we show below this routine in `StrategyEthHegicLP`. This routine is not complicated by basically retrieving current return rate from the staking pool (line 291), then dividing the current total supply (line 292), and finally computing the share in the full-year scale as the `return on investment` or ROI (line 292).

After performing necessary sanity checks on the input arguments, the repurchasing quota requirement is performed at lines 263 – 265. In essence, it verifies the following: `amount.mul(1e18).div(price)< bal`.

```
289     // calculates rewards rate in tokens per year for this address
290     function calculateRate() public view returns(uint256) {
291         uint256 rate = IHegicEthPoolStaking(ethPoolStaking).userRewardPerTokenPaid(
                address(this));
292         uint256 supply = IHegicEthPoolStaking(ethPoolStaking).totalSupply();
293         uint256 roi = IERC20(ethPoolStaking).balanceOf(address(this)).div(supply).mul(
                rate).mul((31536000));
294         return roi;
295     }
```

Listing 3.10: StrategyEthHegicLP::calculateRate()

It is important to note that the lack of `float` support in `Solidity` may introduce subtle, but troublesome issue: precision loss. One possible precision loss stems from the computation when both multiplication (`mul`) and division (`div`) are involved. Specifically, the compuation at line 292 is performed as follows: `IERC20(ethPoolStaking).balanceOf(address(this)).div(supply).mul(rate).mul((31536000))`.

A better approach is to perform the multiplication operation before division to avoid introducing unnecessary precision loss. In other words, the above computation can be revised as the following: `IERC20(ethPoolStaking).balanceOf(address(this)).mul(rate).mul((31536000)).div(supply)`. Certainly, the reordering should not introduce any unwanted overflow in the multiplication operations.

Recommendation Avoid unnecessary precision loss due to the lack of floating support in `Solidity`. If there is no concern in introducing the overflow risk, it is always preferred to perform multiply-before-divide in the computation.

Status The issue has been fixed in this commit: `150fe5b`.

3.7 Business Logic Error in StrategyWbtcHegicLP::adjustPosition()

- ID: PVE-007
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `StrategyWbtcHegicLP`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.3, the `BaseStrategy` contract has defined a set of standard interfaces or methods for every `Yearn.Finance` strategy in order to properly interoperate with the vault contract. And we have examined two of them, i.e., `prepareMigration()` and `exitPosition()`. In this section, we examine another routine – `adjustPosition()`. This routine is designed to perform necessary adjustments to the core position(s) of the strategy given whatever change that may be made in the vault, including new "investable capital" available to the strategy.

To elaborate, we use the `StrategyWbtcHegicLP` contract as an example and show its`adjustPosition()` routine. From this routine, we notice it basically invests the new allowed amount into the configured staking pool for investment (line 105).

```
92      // adjusts position.
93      function adjustPosition(uint256 _debtOutstanding) internal override {
94         //emergency exit is dealt with in prepareReturn
95          if (emergencyExit) {
96            return;
97         }
98
99          // Invest the rest of the want
100         uint256 _wantAvailable = balanceOfWant().sub(_debtOutstanding);
101         if (_wantAvailable > 0) {
102             uint256 _availableFunds = address(this).balance;
103             IHegicWbtcPool(wbtcPool).provide(_availableFunds, 0);
104             uint256 writeWbtc = IERC20(wbtcPool).balanceOf(address(this));
105             IHegicWbtcPoolStaking(wbtcPoolStaking).stake(writeWbtc);
106         }
```

```
107     }
```

Listing 3.11: StrategyWbtcHegicLP::adjustPosition()

However, the current method in computing the allowed amount has implemented an incorrect logic and should be revised. The reason is that the `_availableFunds` (line 102) is calculated from the `ETH` balance, not the intended `WBTC` balance. Therefore, in the likely situation with always 0 `ETH` balance, the current logic may not return any new yields, rendering this strategy not fully functional.

Recommendation Properly compute the right amount for staking. An example revision is shown below:

```
92      // adjusts position.
93      function adjustPosition(uint256 _debtOutstanding) internal override {
94         //emergency exit is dealt with in prepareReturn
95          if (emergencyExit) {
96              return;
97         }
98
99          // Invest the rest of the want
100         uint256 _wantAvailable = balanceOfWant().sub(_debtOutstanding);
101         if (_wantAvailable > 0) {
102             uint256 _availableFunds = IERC20(wbtc).balanceOf(this);
103             IHegicWbtcPool(wbtcPool).provide(_availableFunds, 0);
104             uint256 writeWbtc = IERC20(wbtcPool).balanceOf(address(this));
105             IHegicWbtcPoolStaking(wbtcPoolStaking).stake(writeWbtc);
106         }
107     }
```

Listing 3.12: Revised StrategyWbtcHegicLP::adjustPosition()

Status The issue has been fixed in this commit: `66e45c6`.

3.8 Redundant Code Removal in Multiple Strategies

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `StrategyWBTCHegicLP`, `Strategy`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The new strategies make good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `Math`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the `StrategyWbtcHegicLP` 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 `StrategyWbtcHegicLP` contract, there is a local variable that is defined, but not used anymore: `_wbtcBalance`, This variable is apparently left behind from a deprecated feature and can be safely removed.

```
    // N.B. this will only work so long as the various contracts are not timelocked
    // each deposit into the WBTC pool restarts the 14 day counter on the entire value.
    function exitPosition(uint256 _debtOutstanding)
        internal
        override
        returns (
          uint256 _profit,
          uint256 _loss,
          uint256 _debtPayment
        )
    {
        // Shouldn't we revert if we try to exitPosition and there is a timelock?

        uint256 writeWbtc = IHegicWbtcPool(wbtcPool).shareOf(address(this));
        uint256 stakingBalance = IHegicWbtcPoolStaking(wbtcPoolStaking).balanceOf(
            address(this));

        // by doing this before the timelock check, we will trigger the timelock
        if (stakingBalance > 0) {
            IHegicWbtcPoolStaking(wbtcPoolStaking).exit();
        }

        // timelock will never be negative now that we've changed it to boolean
        bool unlocked = withdrawUnlocked();
        if (unlocked = true) {
            uint256 writeBurn = IERC20(wbtcPool).balanceOf(address(this));
            IHegicWbtcPool(wbtcPool).withdraw(writeWbtc, writeBurn);
            uint256 _wbtcBalance = address(this).balance;
```

```
        }
        else {
            revert("funds timelocked");
        }
    }
```

Listing 3.13: StrategyWbtcHegicLP::exitPosition()

Also, the `Strategy` contract in the `YearnV2-Generic-Lev-Comp-Farm` repository contains a function named `prepareMigration()`. And this function has a redundant transfer operation, i.e., `want.safeTransfer(_newStrategy, want.balanceOf(address(this)))` (line 657), which can also be safely removed.

```
    //lets leave
    //if we can't deleverage in one go set collateralFactor to 0 and call harvest
        multiple times until delevered
    function prepareMigration(address _newStrategy) internal override {
        (uint256 deposits, uint256 borrows) = getLivePosition();
        _withdrawSome(deposits.sub(borrows), false);

        (, , uint256 borrowBalance, ) = cToken.getAccountSnapshot(address(this));

        require(borrowBalance == 0, "DELEVERAGE_FIRST");

        want.safeTransfer(_newStrategy, want.balanceOf(address(this)));

        IERC20 _comp = IERC20(comp);
        uint _compB = _comp.balanceOf(address(this));
        if(_compB > 0){
            _comp.safeTransfer(_newStrategy, _compB);
        }
    }
```

Listing 3.14: Strategy :: prepareMigration()

Recommendation Consider the removal of the unused code and the unused constants.

Status The issue has been fixed in this commit: `2254984`.

3.9 Business Logic Error in Strategy::adjustPosition()

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Strategy`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.3, the `BaseStrategy` contract has defined a set of standard interfaces or methods for every `Yearn.Finance` strategy in order to properly interoperate with the vault contract. And we have examined two of them, i.e., `prepareMigration()` and `exitPosition()`. In this section, we examine another routine – `adjustPosition()` in the `Strategy` contract from the `YearnV2-Generic-Lev-Comp-Farm` repository. This routine is designed to perform necessary adjustments to the core position(s) of the strategy given whatever change that may be made in the vault, including new "investable capital" available to the strategy.

To elaborate, we show below the `adjustPosition()` routine that is designed to compute the new desired position for farming.

```
    function adjustPosition(uint256 _debtOutstanding) internal override {
        //emergency exit is dealt with in prepareReturn
        if (emergencyExit) {
            return;
        }

        //we are spending all our cash unless we have debt outstanding
        uint256 _wantBal = want.balanceOf(address(this));
        if(_wantBal < _debtOutstanding){
            //this is graceful withdrawal. dont use backup
            //we use more than 1 because withdrawunderlying causes problems with 1 token due to different decimals
            if(cToken.balanceOf(address(this)) > 1){
                _withdrawSome(_debtOutstanding - _wantBal, false);
            }

            return;
        }

        (uint256 position, bool deficit) = _calculateDesiredPosition(_wantBal - _debtOutstanding, true);

        //if we are below minimun want change it is not worth doing
        //need to be careful in case this pushes to liquidation
        if (position > minWant) {
            //if dydx is not active we just try our best with basic leverage
```

```
438             if (!DyDxActive) {
439                 uint i = 5;
440                 while(position > 0){
441                     position = position.sub(_noFlashLoan(position, deficit));
442                     i++;
443                 }
444             } else {
445                 //if there is huge position to improve we want to do normal leverage. it
                        is quicker
446                 if (position > want.balanceOf(SOLO)) {
447                     position = position.sub(_noFlashLoan(position, deficit));
448                 }
449 
450                 //flash loan to position
451                 if(position > 0){
452                     doDyDxFlashLoan(deficit, position);
453                 }
454 
455             }
456         }
457     }
```

Listing 3.15: Strategy :: adjustPosition ()

However, the current method in implementing the basic leverage when the `dYdX` integration is inactive is flawed and should be revised. The reason is that the `while`-loop (lines 439 – 443) has an incorrect termination condition, hence leading to an out-of-gas situation. Therefore, the current logic may not return any new yields, rendering this strategy not fully functional.

Recommendation Properly revise the termination condition for the internal `while`-loop. An example revision is shown below:

```
    function adjustPosition(uint256 _debtOutstanding) internal override {
        //emergency exit is dealt with in prepareReturn
        if (emergencyExit) {
            return;
        }

        //we are spending all our cash unless we have debt outstanding
        uint256 _wantBal = want.balanceOf(address(this));
        if(_wantBal < _debtOutstanding){
            //this is graceful withdrawal. dont use backup
            //we use more than 1 because withdrawunderlying causes problems with 1 token
                due to different decimals
            if(cToken.balanceOf(address(this)) > 1){
                _withdrawSome(_debtOutstanding - _wantBal, false);
            }

            return;
        }

```

```
        (uint256 position, bool deficit) = _calculateDesiredPosition(_wantBal -
            _debtOutstanding, true);

        //if we are below minimun want change it is not worth doing
        //need to be careful in case this pushes to liquidation
        if (position > minWant) {
            //if dydx is not active we just try our best with basic leverage
            if (!DyDxActive) {
                uint i = 0;
                while(position > 0){
                    position = position.sub(_noFlashLoan(position, deficit));
                    if(i >= 6){
                        break;
                    }
                    i++;
                }
            } else {
                //if there is huge position to improve we want to do normal leverage. it
                    is quicker
                if (position > want.balanceOf(SOLO)) {
                    position = position.sub(_noFlashLoan(position, deficit));
                }

                //flash loan to position
                if(position > 0){
                    doDyDxFlashLoan(deficit, position);
                }

            }
        }
    }
```

Listing 3.16: Revised Strategy::adjustPosition()

Status The issue has been fixed in this commit: `eba1a94`.

4 Conclusion

In this audit, we have analyzed the design and implementation of four `Hegic Strategies` in the `Yearn.Finance` protocol. The audited system presents a unique addition to current DeFi offerings in maximizing yields for users. As part of `Yearn.Finance`, the four new strategies work with the popular `Hegic` protocol for new yielding opportunities. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

As a final precaution, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

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

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

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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

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

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

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