Silo Finance

Silo Vault

Security Assessment Report

Version: 2.0

March, 2025

Contents

Introduction 2
Disclaimer 2
Document Structure 2
Overview 2

Security Assessment Summary 3
Scope 3
Approach 3
Coverage Limitations 4
Findings Summary 4

Detailed Findings 5

Summary of Findings 6
Missing `owner` Data In Cloned `VaultIncentivesModule` 7
`skim()` Function Allows Unintended Removal Of Share Tokens 8
Potential Market Manipulation Through `reallocateTo()` 9
Possible Market Removal With Non-Zero Token Supply 11
Possibility Of Revert In `_claimRewards()` Due To Hitting Block Gas Limit 12
Malicious Guardian Can Prevent Removal By Exploiting `revokePendingGuardian()` . . 13
Lack of Zero-Address Checks 14
Lack Of Upper Limit On `_notificationReceiver` May Lead To Expensive Token Transfers . . 15
Potential Excessive Cost For `deposit()` And `withdraw()` Operations 16
Miscellaneous General Comments 18

A Test Suite 19

B Vulnerability Severity Classification 20

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Silo Finance components. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract in scope. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Silo Finance components contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Silo Finance components in scope.

Overview

The Silo Project is a collection of smart contracts designed for decentralised lending and asset management. Its primary goal is to allow users to deposit tokens into isolated "silos", which minimise the cross-contamination of risk across multiple markets. By keeping different assets and their associated liabilities separate, the Silo system aims to provide safer and more efficient lending and borrowing strategies.

The `SiloVault` is a core contract within this system, responsible for managing deposits and withdrawals, handling reward claims, and securely interacting with external contracts. It acts as a specialised vault that holds user funds and orchestrates various operations, such as claiming rewards from incentive programs, while enforcing important constraints like reentrancy protection. By simulating withdrawals and carefully distributing rewards, the `SiloVault` supports safe asset transfers and maintains the integrity of the overall Silo lending framework.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Silo Finance repository.

The scope of this time-boxed review was strictly limited to the following files at commit 5549ca2:

- `IdleVault.sol`
- `PublicAllocator.sol`
- `SiloVault.sol`
- `SiloVaultsFactory.sol`
- `ConstantsLib.sol`
- `ErrorsLib.sol`
- `EventsLib.sol`
- `PendingLib.sol`
- `VaultIncentivesModule.sol`
- `SiloIncentivesControllerCL.sol`
- `SiloIncentivesControllerCLFactory.sol`
- `SiloVaultsContracts.sol`
- `IIncentivesClaimingLogic.sol`
- `INotificationReceiver.sol`
- `IPublicAllocator.sol`
- `ISiloIncentivesControllerCLFactory.sol`
- `ISiloVault.sol`
- `ISiloVaultsFactory.sol`
- `IVaultIncentivesModule.sol`

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

The manual review focused on identifying issues associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout).

Additionally, the manual review process focused on identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`
- Aderyn: `https://github.com/Cyfrin/aderyn`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 10 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 1 issue.
- Medium: 2 issues.
- Low: 2 issues.
- Informational: 4 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Silo Finance components in scope. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
SILV-01	Missing `owner` Data In Cloned `VaultIncentivesModule`	Critical	Resolved
SILV-02	`skim()` Function Allows Unintended Removal Of Share Tokens	High	Resolved
SILV-03	Potential Market Manipulation Through `reallocateTo()`	Medium	Closed
SILV-04	Possible Market Removal With Non-Zero Token Supply	Medium	Closed
SILV-05	Possibility Of Revert In `_claimRewards()` Due To Hitting Block Gas Limit	Low	Closed
SILV-06	Malicious Guardian Can Prevent Removal By Exploiting `revokePendingGuardian()`	Low	Closed
SILV-07	Lack of Zero-Address Checks	Informational	Resolved
SILV-08	Lack Of Upper Limit On `_notificationReceiver` May Lead To Expensive Token Transfers	Informational	Closed
SILV-09	Potential Excessive Cost For `deposit()` And `withdraw()` Operations	Informational	Closed
SILV-10	Miscellaneous General Comments	Informational	Closed

SILV-01	Missing `owner` Data In Cloned `VaultIncentivesModule`		
Asset	`SiloVaultsFactory.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

The `createSiloVault()` function clones an implementation contract of `VaultIncentivesModule`, where the constructor initialises the `owner` data. However, in the cloned contract, the `owner` data is not accessible.

As a result, all functions with the `onlyOwner` modifier in the cloned `VaultIncentivesModule` become unusable, including the core function `addIncentivesClaimingLogic()`.

Recommendations

Implement an initialisation function to set the `owner` address in the cloned contract.

This function should be called immediately after the contract is cloned.

Resolution

The Silo team has addressed this issue by implementing the `_VaultIncentivesModule_init()` function to set the `owner` address in the cloned contract.

This issue has been resolved in PR #1029.

SILV-02	`skim()` Function Allows Unintended Removal Of Share Tokens		
Asset	`SiloVault.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

The `skim()` function's implementation does not prevent share tokens, which should not be removed from `SiloVault`, from being transferred to the `skimRecipient` address. As a result, the internal accounting of shares can become inaccurate.

The `skim()` function allows administrators to remove tokens that have accidentally ended up in the `SiloVault` contract. This function is generally safe for anyone to call as the tokens are transferred to the predefined `skimRecipient` address, however, share tokens should be excluded from this process.

An attacker could exploit this by repeatedly calling the function to disrupt the operation of `SiloVault`, particularly if system administrators attempt to rectify the issue by returning share tokens from the `skimRecipient` to `SiloVault`.

Recommendations

Modify implementation of the `skim()` function to explicitly exclude share tokens from the skimming process.

Resolution

The Silo team has address this issue by removing the `skim()` function from the `SiloVault` contract.

This issue has been resolved in PR #1027.

SILV-03	Potential Market Manipulation Through `reallocateTo()`		
Asset	`PublicAllocator.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `reallocateTo()` function introduces a potential vulnerability that could allow malicious actors to manipulate market conditions, leading to unfair advantages or financial gains.

This issue arises due to the ability of users to influence supply allocation within specific markets, which can directly impact interest rates and create opportunities for exploitation.

1. Scenario 1: **Interest Rate Suppression for Borrowers**
   If a user holds a borrow position in a specific market, they can exploit the `reallocateTo()` function to flood the market with additional supply. This sudden increase in supply artificially lowers the interest rate for that market. For users with large borrow positions, the interest savings achieved through this manipulation could far exceed the associated fees, making the attack economically viable.
2. Scenario 2: **Interest Rate Spikes for Liquidation**
   Conversely, if a user aims to liquidate a borrow position in a specific market, they can use `reallocateTo()` to remove a significant portion of the market's supply. This drastic reduction in supply causes a sharp increase in the interest rate. As a result, the affected borrow position may fall below the liquidation threshold, triggering liquidation. The attacker can then submit a liquidation request and claim the liquidation reward. If the reward exceeds the cost of executing the attack (e.g., fees), the manipulation becomes profitable.

Both scenarios are particularly feasible if the vault holds a substantial amount of deposits in the targeted markets, as the attacker can leverage the vault's liquidity to amplify the impact of their actions.

The possible impacts are as follows:

- **Market Manipulation:** Attackers can artificially suppress or spike interest rates, disrupting the normal functioning of the protocol.
- **Financial Losses:** Legitimate users may incur losses due to unfair liquidations or distorted interest rates.
- **Protocol Instability:** Repeated exploitation of this vulnerability could undermine trust in the protocol and destabilize its operations.

Recommendations

To mitigate this issue, implement the following measures:

1. **Deposit/Withdrawal Caps:** Set the optimal value of `flowCaps` to mitigate the unintended effects of the feature.
2. **Fee Adjustments:** Dynamically adjust fees based on the size and impact of reallocation transactions to make exploitation economically unfeasible.

Resolution

This issue has been acknowledged by the Silo team.

SILV-04	Possible Market Removal With Non-Zero Token Supply		
Asset	`SiloVault.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

There are no checks to ensure that `_ERC20BalanceOf(address(market), address(this)) == 0` before market deletion.

In `updateWithdrawQueue()` on line [**344**], the code checks to ensure that, if the vaults's token balance for the market is non-zero, that a pending removal was set and the `timelock` for removal has not passed. This is to give sufficient time for any tokens for the market belonging to the vault to be removed before deletion on line [**352**]. However, the zero balance check is missing before market deletion.

This means that it is possible for a market to be deleted even if a pending removal was set and the `timelock` for removal has passed, but the vault still holds a positive token balance. This will in effect leave unmanaged tokens belonging to the vault for the market which will cause a reduction in available liquidity.

The impact of this reduction in liquidity is a drop in the share price, which will result in withdrawers experiencing a loss. This is due to the fact that `totalAssets()` has been reduced while `totalSupply()` remained unchanged, thus causing each share to be now worth less assets.

Recommendations

As a safety precaution, update the code to check the token balance for the vault in the market before deletion.

This can be achieved by adding the following:

```
if (_ERC20BalanceOf(address(market), address(this)) != 0) {
    revert ErrorsLib.InvalidMarketRemovalNonZeroSupply(market);
}
delete config[market];
```

Resolution

The Silo team has acknowledged this issue with the following response:

"This is by Morpho design we decided not to change it. See comments here and here".

SILV-05	Possibility Of Revert In `_claimRewards()` Due To Hitting Block Gas Limit		
Asset	`SiloVault.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `_claimRewards()` function may revert due to exhausting block gas limit when iterating through large number of logic controllers.

The function has a `for` loop that iterates over multiple incentive claiming logic controllers. Each logic controller calls via `delegatecall()` the `claimRewardsAndDistribute()` function, which has two `for` loops processing `accruedRewards`.

If the function loops over `N` logic controllers, and each logic controller calls `claimRewardsAndDistribute()` with its two loops, the first loop iterating over `M1` rewards and the second loop iterating over `M2` rewards, this results in `O(N × (M1 + M2))` iterations in the worst case.

Since `M1` is equal to `M2` (both `accruedRewards`) the final time complexity is `O(N * M)`.

This may cause the following issues:

1. **High Gas Costs**: The nested loops increase gas consumption exponentially if `N` (the number of logic controllers) and `M` (the number of `accruedRewards`) are large. If the gas limit is exceeded, the entire transaction reverts.
2. **Risk of Hitting the Block Gas Limit**: If the function handles many incentive logic controllers, the increased gas consumption from the nested loops may exceed the block gas limit.
3. **Call Stipends & Failures**: `delegatecall` inherits the gas limit from the caller, so if a single iteration consumes too much gas, subsequent calls may fail mid-loop, leading to failure.

Recommendations

Consider placing an upper limit on the number of incentive claiming logic controllers for each market.

Resolution

This issue has been acknowledged by the Silo team.

SILV-06	Malicious Guardian Can Prevent Removal By Exploiting `revokePendingGuardian()`		
Asset	`Contract.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

A malicious guardian can exploit the `revokePendingGuardian()` function to prevent their removal from the system.

By continuously revoking any pending guardian updates, the malicious guardian can indefinitely block the owner or other authorised party from replacing them.

This creates a situation where even the owner is unable to remove or replace the guardian, effectively locking the system into an undesirable state.

This issue exists because the `revokePendingGuardian()` function allows a guardian to cancel any pending updates to their role. If a guardian acts in bad faith, they can abuse this functionality to maintain their position indefinitely, undermining the governance and security of the protocol.

Recommendations

To address this issue, allow the `owner` to override the revocation process in exceptional cases, ensuring that malicious guardians can still be removed.

Resolution

This issue has been acknowledged by the Silo team with the following response:

"Adding more power to the owner will create a power misbalance".

SILV-07	Lack of Zero-Address Checks
Asset	SiloIncentivesControllerCL.sol
Status	**Resolved:** See Resolution
Rating	Informational

Description

In the constructor of `SiloIncentivesControllerCL` contract, there is no check to ensure that the parameters `_vaultIncentivesController` and `_siloIncentivesController` are not the zero address.

The additional checks are advised as both `VAULT_INCENTIVES_CONTROLLER` and `SILO_INCENTIVES_CONTROLLER` are immutable.

Recommendations

Consider adding a check to ensure that `_vaultIncentivesController` and `_siloIncentivesController` are not the zero address.

Resolution

The Silo team has addressed this issue by adding a check to ensure that `_vaultIncentivesController` and `_siloIncentivesController` are not the zero address.

This issue has been resolved in PR #1058.

SILV-08	Lack Of Upper Limit On `_notificationReceiver` May Lead To Expensive Token Transfers
Asset	`VaultIncentivesModule.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The function `addNotificationReceiver()` allows the contract owner to add a notification receiver, which will be called by `SiloVault._afterTokenTransfer()` whenever a token transfer occurs. However, the current implementation does not impose an upper limit on the number of `_notificationReceiver` addresses that can be added.

This lack of a limit can lead to significant gas costs during token transfers, as each transfer triggers a call to `INotificationReceiver.afterTokenTransfer()` for every registered notification receiver. If a large number of receivers are added, the cumulative gas cost of these calls could make token transfers prohibitively expensive, negatively impacting the usability and efficiency of the `SiloVault` contract.

Recommendations

To mitigate this issue, consider introducing a reasonable upper limit on the number of `_notificationReceiver` addresses that can be added. This limit should balance functionality with gas efficiency.

Resolution

This issue has been acknowledged by the Silo team.

SILV-09	Potential Excessive Cost For `deposit()` And `withdraw()` Operations
Asset	`SiloVault.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `SiloVault` contracts are designed to support connections to a maximum of 30 markets. While this design allows for flexibility and scalability, it introduces significant gas cost implications when the maximum number of markets is reached. Under normal network conditions, the gas costs for key functions such as `deposit()` and `withdraw()` are already substantial, but they can become prohibitively expensive during periods of gas price volatility.

Gas Cost Breakdown

- **`deposit()`**: When the maximum of 30 markets is connected, a single `deposit()` call consumes **2,393,330 gas**.
- **`withdraw()`**: Similarly, a `withdraw()` call consumes **1,625,161 gas** under the same conditions.

Cost Implications

1. **Normal Network Conditions (1 GWei gas price):**
   - `deposit()` costs approximately **US$6**.
   - `withdraw()` costs approximately **US$4**.
2. **High Gas Price Scenarios (50 GWei gas price):**
   - `deposit()` costs rise to approximately **US$300**.
   - `withdraw()` costs rise to approximately **US$200**.

These costs can become a significant barrier for users, especially during periods of network congestion or gas price spikes. The high gas costs may deter users from interacting with the protocol, reducing its usability and accessibility.

Impact

- **User Experience:** High gas costs can discourage users from depositing or withdrawing funds, limiting the protocol's functionality and adoption.
- **Financial Burden:** Users may face unexpectedly high transaction fees, particularly during periods of gas price volatility, leading to potential financial losses.
- **Protocol Efficiency:** The high gas costs may reduce the efficiency of the protocol, as users may delay or avoid transactions altogether.

Recommendations

To address this issue, consider implementing the following measures:

1. **Gas Optimisation:** Review and optimise the `SiloVault` contract code to reduce gas consumption for `deposit()` and `withdraw()` functions, especially when the maximum number of markets is connected.
2. **Batch Transactions:** Enable batch processing for deposits and withdrawals to reduce the number of transactions and overall gas costs.
3. **User Alerts:** Implement a system to notify users of current gas prices and estimated transaction costs, allowing them to make informed decisions.

Resolution

This issue has been acknowledged by the Silo team.

SILV-10	Miscellaneous General Comments
Asset	All contracts
Status	**Closed:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Inconsistent Comments and Code for `MAX_SETTABLE_FLOW_CAP`**

   Related Asset(s): IPublicAllocator.sol

   The comments and code related to the `MAX_SETTABLE_FLOW_CAP` constant in the contract are inconsistent, which could lead to confusion or misinterpretation. Specifically:

   - The comment on line [9] states that the maximum cap is `type(uint128).max - 1`, which equals `340282366920938463463374607431768211454`.
   - However, the comment on line [10] and the actual code on line [11] use `type(uint128).max / 2`, which equals `170141183460469231731687303715884105727`.

   This discrepancy creates ambiguity about the intended value of `MAX_SETTABLE_FLOW_CAP`. If the comments and code are not aligned, developers may misunderstand the constraints or behaviour of the contract, potentially leading to errors in implementation or usage.

   To resolve this issue, consider updating the comment on line [9] to match the actual implementation, clarifying that `MAX_SETTABLE_FLOW_CAP` is set to `type(uint128).max / 2`.

2. **Suggested Renaming of Modifier `onlyAllocatorRole` for Clarity**

   Related Asset(s): SiloVault.sol

   The current modifier `onlyAllocatorRole` may be misleading because it grants access not only to allocators but also to curator. This naming inconsistency can cause confusion, especially when compared to other modifiers with similar behaviour, such as `onlyCuratorOrGuardianRole`, which explicitly lists all roles it grants access to curator and guardian, including the owner.

   To improve clarity and maintain consistency across the codebase, it is recommended to rename the `onlyAllocatorRole` modifier to `onlyAllocatorOrCuratorRole`. This change would accurately reflect the roles that have access and align with the naming conventions used for other modifiers in the system.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The Silo team has chosen to address the first issue by updating the comment to match the code. This has been resolved in PR #1059.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are given along with this document. The `forge` framework was used to perform these tests and the output is given below.

```
Ran 1 test for test/tests-local/Ownable2Step.sigp.t.sol:Ownable2StepChildSigpTest
[PASS] test_sigp_owner() (gas: 12628)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 978.46us (133.87us CPU time)

Ran 3 tests for test/tests-local/UtilsLib.sigp.t.sol:UtilsLibTestSigp
[PASS] test_sigp_zeroFloorSub_fuzz(uint256,uint256) (runs: 260, μ: 6062, ~: 6062)
[PASS] test_sigp_zeroFloorSub_single_x() (gas: 2188)
[PASS] test_sigp_zeroFloorSub_single_y() (gas: 2133)
Suite result: ok. 3 passed; 0 failed; 0 skipped; finished in 13.85ms (13.15ms CPU time)

Ran 1 test for test/tests-local/MarketTest.sigp.t.sol:MarketTestSigp
[PASS] test_sigp_testUpdateWithdrawQueueInvalidMarketRemovalNonZeroSupply() (gas: 443231)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 182.07ms (1.07ms CPU time)
proptest: Aborting shrinking after the PROPTEST_MAX_SHRINK_ITERS environment variable or ProptestConfig.max_shrink_iters
      ↪  iterations (set 0 to a large(r) value to shrink more; current configuration: 0 iterations)

Ran 2 tests for test/tests-local/SiloVaultsFactory.sigp.t.sol:SiloVaultsFactoryTestSigp
[PASS] testFail_sigp_createSiloVault_zero_owner_SiloVault(uint256,string,string) (runs: 260, μ: 223111, ~: 212441)
[FAIL. Reason: incentivesModule.owner() should be equal to initialOwner:
      0x0000000000000000000000000000000000000000 != 0x5a8D0d351b347E9997dC656c964BB92bcb1C5B39;
      counterexample: calldata=0x91378b070000000000000000000000005a8d0d351b347e9997dc656c964bb92bcb1c5b3900
      0000000000000000000000000002a690dd4e774d919b1a21d8d158b4b79f3400000000000000000000000
      0000000000000000000000000000000000000008000000000000000000000000000000000000000000000
      000000000000000000e00000000000000000000000000000000000000000000000000000000000000002f2
      2e18d97e0bd8f2ec8bae0a1a5f09e8aa6c2a5f09f8990eaa0b2443fc2a5efacb93ae0b6bd3cf09f95b44c
      f096bfa3000000000000000000000000000000000000000000000000000000000000000000000000000000
      000000000000000000018ce8c2e73e385b0e0b1ae27f09eb9973dc8bac2a4e1898c360000000000000000
            ↪  args=[0x5a8D0d351b347E9997dC656c964BB92bcb1C5B39, 230904002692875149107248733348236042411828 [2.309e41], "", ""]]
            ↪  test_sigp_createSiloVault_zero_owner_VaultIncentivesModule_Vuln(address,uint256,string,string) (runs: 0, μ: 0, ~:
            ↪  0)
Suite result: FAILED. 1 passed; 1 failed; 0 skipped; finished in 289.86ms (203.44ms CPU time)

Ran 13 tests for test/tests-local/SiloVault.sigp.t.sol:SiloVaultTestSigp
[PASS] testFail_sigp_inflation() (gas: 871232)
[PASS] testFail_sigp_setCap_max() (gas: 38368)
[PASS] test_sigp_borrow() (gas: 943722)
[PASS] test_sigp_borrow_validMarkets(uint256[100]) (runs: 260, μ: 46144386, ~: 46144509)
[PASS] test_sigp_claimRewards() (gas: 12475)
[PASS] test_sigp_deposit_withdraw_one_market(uint256,uint256) (runs: 260, μ: 773132, ~: 773188)
[PASS] test_sigp_deposit_withdraw_valid_markets(uint256,uint256) (runs: 260, μ: 16461505, ~: 18381264)
[PASS] test_sigp_market_removal_share_price_change(uint256[100]) (runs: 260, μ: 46092856, ~: 46092666)
[PASS] test_sigp_setFee_setFeeRecipient() (gas: 43057)
[PASS] test_sigp_submitCap() (gas: 208267)
[PASS] test_sigp_supplyQueue_withdrawQueue() (gas: 4462193)
[PASS] test_sigp_vault_deposit() (gas: 8159512)
[PASS] test_sigp_withdraw_markets() (gas: 15959014)
Suite result: ok. 13 passed; 0 failed; 0 skipped; finished in 42.78s (101.71s CPU time)

Ran 5 test suites in 42.78s (43.27s CPU time): 19 tests passed, 1 failed, 0 skipped (20 total tests)

Failing tests:
Encountered 1 failing test in test/tests-local/SiloVaultsFactory.sigp.t.sol:SiloVaultsFactoryTestSigp
[FAIL. Reason: incentivesModule.owner() should be equal to initialOwner: 0x0000000000000000000000000000000000000000 !=
      ↪  0x5a8D0d351b347E9997dC656c964BB92bcb1C5B39;]
      ↪  test_sigp_createSiloVault_zero_owner_VaultIncentivesModule_Vuln(address,uint256,string,string) (runs: 0, μ: 0, ~: 0)

Encountered a total of 1 failing tests, 19 tests succeeded
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact	High	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Low	Low	Medium
		Low	Medium	High
			Likelihood	

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].