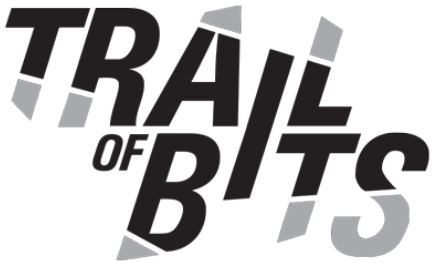

Opyn Gamma Protocol

Security Assessment

May 28, 2021

Prepared For:
Aparna Krishnan | *Opyn*
aparna@opyn.co

Haythem Sellami | *Opyn*
haythem@opyn.co

Andrew Leone | *Opyn*
andrew@opyn.co

Prepared By:
Dominik Teiml | *Trail of Bits*
dominik.teiml@trailofbits.com

Mike Martel | *Trail of Bits*
mike.martel@trailofbits.com

Changelog:
May 28, 2021: Initial report delivered
June 7, 2021: Added Appendix D. Fix Log

Executive Summary

From May 10 to May 28, 2021, Trail of Bits conducted a review of the Opyn Gamma Protocol smart contracts, working from commit 9a75da2a of the `GammaProtocol` repository.

During the first week of the audit, we primarily focused on gaining an understanding of the Gamma Protocol codebase and documentation. In the second week, we continued our manual code review and used static analysis tools such as Slither to uncover issues. Finally, in the last week of the audit, we completed our manual code review and tested margin calculations by using Echidna to carry out property-based fuzzing.

Our review resulted in 17 findings ranging from high to informational severity. Most issues, including those of high severity, were related to data validation; these include a lack of signature verification for data forwarded to an external party (TOB-OPYN-002), potentially incorrect assumptions on the value of stablecoins (TOB-OPYN-010), potential mismatches between fixed-point integers and raw values (TOB-OPYN-015), and incorrect assumptions on the data returned by the Yearn price oracle (TOB-OPYN-017). Similarly, one function returns the absolute value when converting a signed integer to an unsigned integer (TOB-OPYN-012, TOB-OPYN-013).

We also found that under the new partial collateralization scheme, adverse market conditions could disincentivize users from carrying out liquidations, which would affect the solvency of the Opyn Gamma Protocol `MarginPool` (TOB-OPYN-004). The risk parameters set by Opyn are conservative and could prove to be insufficient in the case of a black swan event. However, because black swan events are inherently unpredictable, it was not possible for us to judge the likelihood of such an event. Additionally, there is no reserve fund to backstop these types of events, which is a common feature of other DeFi protocols that have partial collateralization and liquidation schemes.

Although we discovered issues in the currently deployed code, our audit was focused on the code added to the Gamma Protocol to allow for partial collateralization and the new liquidation functionality. Many of the issues in the newer code stemmed from incorrect or improperly documented assumptions. Opyn should consider generalizing parts of the Gamma Protocol code (where appropriate) to make the invariants more obvious and easier to verify.

Trail of Bits recommends that Opyn take the following steps:

- Address all reported issues.
- Remove the hard-coded assumptions about the types of assets used in the Gamma Protocol. If the assumptions are left in, over time, they will complicate the token integration process.
- Simplify the statement and verification of invariants to make the code more conducive to automated verification.

- Conduct a security review to verify the properties and invariants of all external components integrated into the protocol, such as tokens and oracles; similarly, review any reserve fund mechanisms introduced into the protocol at a later time.

Project Dashboard

Application Summary

Name	Opyn Gamma Protocol
Version	9a75da2a
Type	Solidity
Platform	Ethereum

Engagement Summary

Dates	May 10, 2021 – May 28, 2021
Method	Full knowledge
Consultants Engaged	2
Level of Effort	6 person-weeks

Vulnerability Summary

Total High-Severity Issues	6	■ ■ ■ ■ ■ ■
Total Medium-Severity Issues	3	■ ■ ■
Total Low-Severity Issues	1	■
Total Informational-Severity Issues	7	■ ■ ■ ■ ■ ■ ■
Total	17	

Category Breakdown

Data Validation	12	■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
Auditing and Logging	1	■
Configuration	1	■
Denial of Service	1	■
Patching	1	■
Undefined Behavior	1	■
Total	17	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory**. The vast majority of functions have appropriate access controls. We did find that it would be possible for an attacker to execute trades on behalf of a user who has approved the `TradeCallee` contract (TOB-OPYN-002); however, we were informed that this module has been removed.
Arithmetic	**Moderate**. We did not find any critical arithmetic vulnerabilities. However, improving the handling of the arithmetic would reduce the chance of vulnerabilities (TOB-OPYN-014). For example, a library function turns a negative integer into its absolute value, which leads to unexpected results (TOB-OPYN-012, TOB-OPYN-013).
Assembly Use	**Strong**. The system does not use assembly code.
Centralization	**Moderate.** The protocol relies heavily on the liveness of privileged actors (TOB-OPYN-011), who play a significant role in the system. If one of these privileged actors were compromised, the system would likely become. This risk is exacerbated by the lack of strict validation of data submitted by privileged actors (TOB-OPYN-015).
Upgradeability	**Strong.** Two of the modules, `oToken` and `Controller`, could be upgraded in the future. However, we did not find any issues arising from their respective upgradeability mechanisms.
Function Composition	**Moderate.** The composition of the system's functions and contracts is generally sound. However, there are many points in the system in which the architecture could be improved (TOB-OPYN-007, TOB-OPYN-008, TOB-OPYN-009).
Front-Running	**Satisfactory**. The system is no more vulnerable to front-running than the execution context into which it is embedded.
Monitoring	**Moderate.** The system emits events in most modules. However, the `MarginCalculator` defines four new events but does not use any of them (TOB-OPYN-005).
Specification	**Moderate.** The client provided a specification and a significant amount of documentation with the codebase. However, certain code comments do not accurately reflect the implementation (TOB-OPYN-015).
Testing &	**Moderate.** The project contains hard-coded unit tests and

Verification	integration tests but no fuzz testing. Furthermore, the test suite was unable to catch bugs such as TOB-OPYN-013. The system would benefit from the addition of fuzzing tests as well as invariant assertions in function bodies. Specifically, it would benefit from invariants that check that fixed-point integers assumed to be non-negative are indeed non-negative. These assertions would trigger an exception if they failed during development or testing and could be filtered out for production.

Engagement Goals

The engagement was scoped to provide a security assessment of the upgrade to the Gamma Protocol, which introduces a partial collateralization scheme and a new liquidation mechanism.

Specifically, we sought to answer the following questions:

- Is the arithmetic correct?
- Are the system's assumptions about external contracts such as oracles correct?
- Does the system suffer from access control vulnerabilities?
- Is it possible for users to steal funds (such as through reentrancy)?
- Are the mechanisms for minting and redeeming oTokens correct?
- Does the mechanism that allows a vault owner to settle a vault work as it should?
- Does the mechanism that enables any user to liquidate a vault work correctly?
- Is the calculation of required collateral correct for all vault types?

Coverage

`Oracle` and pricers. We assessed the access controls of the oracle and pricers (confirming that only privileged parties would be able to post price feeds), checked for centralization issues (analyzing what would happen if a privileged party violated the liveness assumptions), and checked that the Chainlink, Compound, and Yearn APIs were used correctly.

`Whitelist` and `AddressBook`. We assessed the access controls to ensure that only owners would be able to add a module to an "allow list" or a "ban list." We also assessed the data validation that occurs when a new product is whitelisted and checked that the contracts returned correct data when called through a view.

`oToken`. We checked the access controls on minting and burning (processes that should be restricted to the `Controller`) and checked that token symbols and names were set correctly. We also verified that there was no additional logic that would prevent the system from functioning correctly.

`MarginPool`. We checked that only the `Controller` could execute token transfers and that tokens would be handled correctly regardless of whether they provide return data. We also checked that the farmer mechanism worked correctly and did not introduce any security vulnerabilities.

`MarginCalculator`. We checked that the calculator would return correct data on fully collateralized vaults, fully collateralized spread vaults, and naked margin vaults. In

particular, we checked the correctness of decimals and division operations. We also assessed the correctness of the contract's interactions with external oracles and the auction mechanism. We verified that the appropriate collateral requirements were returned for all vault types, that vault data validation was performed correctly, and that the correct cash value of a vault would be used in the settling process. Finally, we reviewed the access controls (particularly the controls on the setting of configuration values) and the related data validation.

`Controller`. We verified that the `Controller` appropriately handled the process of executing multiple actions, and that arguments were parsed correctly for all nine action types. Additionally, we checked that an undercollateralized vault would provide the proper liquidation incentives and mechanisms. We also verified whether it would be possible to correctly redeem oTokens and settle an expired vault. Lastly, we assessed whether invoking `_call` or updating a vault would lead to unintended behavior.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Review the codebase and document the source and version of each dependency.** Include third-party sources as submodules in your Git repository to maintain internal path consistency and ensure that dependencies are updated periodically. TOB-OPYN-001

❑ **Consider implementing a signature verification step that occurs before orders are forwarded to 0x from `TradeCallee`, either by having `tx.origin` sign the data passed to `callFunction` or by using the provided signatures for orders to validate that they were signed by `tx.origin`.** Alternatively, use the `_sender` parameter instead of the `tx.origin` validation to prevent attackers from transferring users' funds. TOB-OPYN-002

❑ **Ensure that all functions that should return a value do.** TOB-OPYN-003

❑ **Consider requiring that margin users pay fees to subsidize the default risk incurred by the Gamma Protocol.** With this requirement, the system would be similar to other DeFi systems that charge interest or stability fees when a user borrows funds. TOB-OPYN-004

❑ **Ensure that all functions that modify key variables emit events corresponding to those updates.** TOB-OPYN-005

❑ **Make `callRestricted` `true` by default.** Alternatively, consider removing this mechanism altogether to prevent the above scenario, which can occur whenever it is `false`. TOB-OPYN-006

❑ **Consider making the `MarginCalculator` a library and changing the inner mapping of `vaults` into an array or linked list.** TOB-OPYN-007

❑ **Document the fact that short, long, collateral, and vault data structures may be sparse.** TOB-OPYN-008

❑ **Consider implementing a check of the invariants before `settleVault` is used.** TOB-OPYN-009

❑ **Consider having the protocol fetch the price feed for all assets.** This will ensure that the system is resilient to even stable price volatility. TOB-OPYN-010

❑ **Limit the use of hard-coded values tied to specific external contracts in favor of more general functionalities and assumptions.** TOB-OPYN-011

❑ **Change the semantics of `intToUint` such that it reverts if a negative number is passed in.** Also consider adding asserts to the code, perhaps asserts executed only during development or testing, to check that FPI values that are assumed to be non-negative are indeed non-negative. TOB-OPYN-012

❑ **Consider changing the semantics of `getExcessCollateral` and `getProceed` to return signed integers, with negative numbers representing undercollateralized vaults.** Alternatively, ensure that the outputs of `getExcessCollateral` are not discarded or used in logic that requires that function. TOB-OPYN-013

❑ **Consider introducing additional rounding direction options and multiplying by the inner denominator instead of using nested division.** Also consider adding asserts to the code, perhaps asserts executed only during development or testing, to check that FPI values are never fully nulled, as such values would probably cause subsequent calculations to be incorrect. TOB-OPYN-014

❑ **Make realistic assumptions about the possible values of the `MarginCalculator`'s configuration variables, and add a data validation mechanism to the setters to check that they are within those bounds.** Also remove the comments that indicate dust should be scaled by 1e27, when in fact it should be scaled by the collateral decimals. TOB-OPYN-015

❑ **Until you are certain that the underlying `yToken` contracts have been updated to resolve this issue, consider having the protocol request the `decimals` value as part of each Yearn pricer request.** TOB-OPYN-016

❑ **Consider using `yTokens` as collateral only for options that use those tokens, rather than a correlated asset, as the underlying asset.** TOB-OPYN-017

Long term

❑ **Ensure that all packages (not only contracts) receive upstream changes.** TOB-OPYN-001

❑ **Ensure that all proxy contracts have well-defined call conditions and means of verifying that those conditions are met.** TOB-OPYN-002

❑ **Leverage a static analysis tool such as Slither, which will catch this bug.** TOB-OPYN-003

❑ **Consider adding mechanisms that can detect adverse market conditions and take automated actions to minimize those conditions' impact on system stability.** TOB-OPYN-004

❑ **Consider expanding the test coverage to verify that key events are emitted during updates of important state variables.** TOB-OPYN-005

❑ **Ensure that the contracts are invulnerable to malicious behavior, in the presence of other network contracts, even after initial deployment.** TOB-OPYN-006

❑ **Invest time in thinking through the architecture of the contracts and the data flow to ensure readability, simplicity, and long-term maintainability.** TOB-OPYN-007

❑ **Consider swapping the deleted value and the last value and removing array elements with `.pop()`.** Using `.pop()` is the best way to delete the last element and decrease the length of an array. Alternatively, consider using a linked list, which would also solve this issue. TOB-OPYN-008

❑ **Consider adding more constraints to the `operate` function, such as invariant checks after every operation.** TOB-OPYN-009

❑ **Ensure that the system is resilient to all changes in market conditions.** TOB-OPYN-010

❑ **Ensure that all assumptions about contract input data are made explicit and are documented, and centralize the logic that validates these assumptions as much as possible to facilitate readability.** TOB-OPYN-011

❑ **Avoid patterns that use total functions that exhibit unintended behavior when handling unexpected input.** It is preferable for functions to revert when handling unexpected input (operating as partial functions). TOB-OPYN-012

❑ **Avoid the coercion of negative values into positive ones.** TOB-OPYN-013

❑ **Consider formally verifying the correctness of the `FixedPointInt256` library using a tool such as Manticore.** TOB-OPYN-014

❑ **Validate all data provided to the contracts, including by owners.** This will make the system resilient against mistakes made by owners and will limit the operation space of any attacker who gains control of an owner address. TOB-OPYN-015

❑ **Ensure that the properties assumed for external integrations with the Gamma Protocol are documented and verified.** If they are not, consider coding more defensively, using weaker sets of assumptions. TOB-OPYN-016

❑ **Consider how to handle collateral integrations that can have deflationary or inflationary effects on underlying assets; the Gamma Protocol code will likely need to be changed to ensure that those integrations function as intended, without violating collateralization invariants.** TOB-OPYN-017

Findings Summary

#	Title	Type	Severity
1	Contracts used as dependencies do not track upstream changes	Patching	Low
2	TradeCallee does not validate trade orders	Data Validation	High
3	Controller _call function lacks a return statement	Undefined Behavior	Informational
4	Adverse market conditions can eliminate liquidation incentives	Denial of Service	High
5	MarginCalculator defines events but never emits them	Auditing and Logging	Medium
6	callRestricted is disabled by default	Data Validation	Medium
7	Architecture can be simplified	Configuration	Informational
8	Short, long, collateral, and vault data structures may be sparse “arrays”	Data Validation	Informational
9	Error-prone operate function	Data Validation	Informational
10	Stablecoin value is assumed to be constant	Data Validation	High
11	Numerous internal and external assumptions	Data Validation	Informational
12	intToUint returns absolute values	Data Validation	High
13	getProceed returns absolute value of required collateral	Data Validation	Medium
14	Non-ideal handling of arithmetic	Data Validation	Informational
15	Unclear configuration values standards	Data Validation	High
16	Decimals set by Yearn pricer do not reflect changes to yToken decimals	Data Validation	Informational
17	yToken exchange rates are fully calculated only during withdrawals	Data Validation	High

1. Contracts used as dependencies do not track upstream changes

Severity: Low
Type: Patching
Target: `contracts/packages/**/*.sol`

Difficulty: High
Finding ID: TOB-OPYN-001

Description
Several third-party contracts have been copied and pasted into the Gamma Protocol repository, including the following:

- `BokkyPooBahsDateTimeLibrary.sol`
- The entire `packages/oz` directory
- `Spawn.sol`

Moreover, for the latter two, the code documentation does not specify the exact revision that was made or whether it was modified. As such, the contracts may not reliably reflect updates or security fixes implemented in their dependencies, as those changes must be manually integrated into the contracts.

Exploit Scenario
A third-party contract used in the Gamma Protocol receives an update with a critical fix for a vulnerability. An attacker detects the use of a vulnerable contract and exploits the vulnerability against the Gamma Protocol.

Recommendations
Short term, review the codebase and document the source and version of each dependency. Include third-party sources as submodules in your Git repository or as npm modules to maintain internal path consistency and ensure that dependencies are updated periodically.

Long term, ensure that all packages (not only contracts) receive upstream changes.

2. TradeCallee does not validate trade orders

Severity: High
Type: Data Validation
Target: `contracts/external/callees/TradeCallee.sol`

Difficulty: High
Finding ID: TOB-OPYN-002

Description
The `TradeCallee` contract provides an interface that enables a Gamma Protocol user to transact with the 0x decentralized exchange. Users send orders by calling the `batchFillLimitOrders` function, which passes in signed orders that 0x then validates and processes. The transaction origin (`tx.origin`) is used to specify the account of the trader, which transfers assets from the trader to the `TradeCallee` contract prior to the execution of the order.

```
        require(
            tx.origin == trader,
            "TradeCallee: funds can only be transferred in from the person sending the
transaction"
        );

        for (uint256 i = 0; i < orders.length; i++) {
            address takerAsset = orders[i].takerToken;
            ERC20Interface(takerAsset).safeTransferFrom(trader, address(this),
takerTokenFillAmounts[i]);
            ERC20Interface(takerAsset).safeIncreaseAllowance(address(exchange),
takerTokenFillAmounts[i]);
        }
```

Figure 2.1: The portion of `callFunction` that transfers assets from the `tx.origin` (without code comments).

Since there is no subsequent validation that the order submitted to this function was signed by `tx.origin`, the funds may be transferred from the user and leveraged to execute an order signed by another individual.

Exploit Scenario
Alice has preexisting approval to transfer assets to the `TradeCallee` contract to facilitate the purchase of assets using 0x. Eve, an attacker, creates a malicious proxy contract, which Alice then calls. The proxy contract calls the `Controller`, which calls `TradeCallee`. `TradeCallee` uses `tx.origin`'s funds (i.e., Alice's funds) to perform a trade signed by Eve.

Recommendations
Short term, consider implementing a signature verification step that occurs before orders are forwarded to 0x from `TradeCallee`, either by having `tx.origin` sign the data passed to

`callFunction` or by using the provided signatures for orders to validate that they were signed by `tx.origin`. Alternatively, use the `_sender` parameter instead of the `tx.origin` validation to prevent attackers from transferring users' funds.

Long term, ensure that all proxy contracts have well-defined call conditions and means of verifying that those conditions are met.

3. Controller _call function lacks a return statement

Severity: Informational
Type: Undefined Behavior
Target: `contracts/core/Controller.sol`

Difficulty: High
Finding ID: TOB-OPYN-003

Description
The `Controller`'s `_call` function includes a return value in its definition but does not return a value.

```
    function _call(Actions.CallArgs memory _args)
        internal
        notPartiallyPaused
        onlyWhitelistedCallee(_args.callee)
        returns (uint256)
    {
        CalleeInterface(_args.callee).callFunction(msg.sender, _args.data);

        emit CallExecuted(msg.sender, _args.callee, _args.data);
    }
```

Figure 3.1: The `_call` function's definition, with the return-related line highlighted in yellow.

The `operate` function, which invokes the `_call` function, does not use a return value.

Exploit Scenario
The `_call` function is used outside of the `operate` function and a function user expects a return value, leading to unexpected behavior.

Recommendations
Short term, ensure that all functions that should return a value do.

Long term, leverage a static analysis tool such as Slither, which will catch this bug.

4. Adverse market conditions can eliminate liquidation incentives

Severity: High | Difficulty: High
Type: Denial of Service | Finding ID: TOB-OPYN-004
Target: `contracts/core/Controller.sol, contracts/core/MarginCalculator.sol`

Description
A naked margin vault can be liquidated as market conditions change and the amount of collateral required for a vault with a short position increases. Liquidation processes are carried out via auctions, with a maximum offer price equal to the amount of collateral in the delinquent vault.

However, if the prevailing market price for a given option is equal to or in excess of the amount of collateral available at auction, other users may not be incentivized to liquidate the vault. The oToken `redeem` function will still allow the system-wide `MarginPool`, which holds all collateral assets, to fulfill the obligation, even though the obligation may no longer be secured by collateral.

Currently, the only mechanism for correcting this type of imbalance involves sending assets via a `donate` function. Instead, Opyn could act as a liquidator of last resort.

Exploit Scenario
Eve opens a partially collateralized vault holding an oToken that will be worth more than the current collateral amount when the token expires. Because of adverse market conditions such as a systemic price shock, extreme volatility spikes, or oToken illiquidity, Eve's vault avoids liquidation, and the oToken expires. This allows Eve to withdraw more funds from the Gamma Protocol `MarginPool` than she deposited into it.

Recommendations
Short term, consider requiring that margin users pay fees to subsidize the default risk incurred by the Gamma Protocol. With this requirement, the system would be similar to other DeFi systems that charge interest or stability fees when a user borrows funds.

Long term, consider adding mechanisms that can detect adverse market conditions and take automated actions to minimize those conditions' impact on system stability.

5. MarginCalculator defines events but never emits them

Severity: Medium
Difficulty: High
Type: Auditing and Logging
Finding ID: TOB-OPYN-005
Target: `contracts/core/MarginCalculator.sol`

Description
The `MarginCalculator` defines four types of events that correspond to updates of important variables used in the contract. However, the functions that modify these variables do not emit events, so changes are not logged appropriately.

```
    /// @notice emits an event when collateral dust is updated
    event CollateralDustUpdated(address indexed collateral, uint256 dust);
    /// @notice emits an event when new time to expiry is added for a specific product
    event TimeToExpiryAdded(bytes32 indexed productHash, uint256 timeToExpiry);
    /// @notice emits an event when new upper bound value is added for a specific time to
expiry timestamp
    event MaxPriceAdded(bytes32 indexed productHash, uint256 timeToExpiry, uint256 value);
    /// @notice emits an event when spot shock value is updated for a specific product
    event SpotShockUpdated(bytes32 indexed product, uint256 spotShock);
```

Figure 5.1: Events defined by the `MarginCalculator` that are never emitted.

As a result, changes made accidentally or maliciously could go unnoticed and ultimately affect the contract's operations.

Exploit Scenario
By setting the `spotShock` variable to a certain value, Eve enables the liquidation of margin vaults that should not be liquidated. Because of the lack of events, the `spotShock` change is not logged. As a result, owners of vaults that use naked margins may not be able to protect their collateral from being inappropriately liquidated.

Recommendations
Short term, ensure that all functions that modify key variables emit events corresponding to those updates.

Long term, consider expanding the test coverage to verify that key events are emitted during updates of important state variables.

6. callRestricted is disabled by default

Severity: Medium
Type: Data Validation
Target: `contracts/core/Controller.sol`

Difficulty: High
Finding ID: TOB-OPYN-006

Description
The `_call` function is used to call other contracts on behalf of the `Controller`. The function's `onlyWhitelistedCallee` modifier prevents it from calling contracts not meant to be called:

```
    function _call(Actions.CallArgs memory _args)
        internal
        notPartiallyPaused
        onlyWhitelistedCallee(_args.callee)
        returns (uint256)
    {
        CalleeInterface(_args.callee).callFunction(msg.sender, _args.data);

        emit CallExecuted(msg.sender, _args.callee, _args.data);
    }
```

Figure 6.1: The `_call` function has an `onlyWhitelistedCallee` modifier.

`onlyWhitelistedCallee` performs the following check:

```
    modifier onlyWhitelistedCallee(address _callee) {
        if (callRestricted) {
            require(_isCalleeWhitelisted(_callee), "CO3");
        }

        _;
    }
```

Figure 6.2: The `onlyWhitelistedCallee` modifier.

However, `callRestricted` is `false` by default and is not set to `true` in the constructor or `initialize` function.

```
    bool public callRestricted;
```

Figure 6.3: `callRestricted` is off by default.

This means that any user can call any other contract on the network by using the `Controller`'s address as the `msg.sender`.

Exploit Scenario
Rewards are added to the system. The reward tokens are owned by the `Controller` and distributed to users. The reward token has a fallback function, and when the `Controller` calls it, the `Controller` can execute arbitrary behavior. Eve invokes `_call`, with the token contract set as the destination. She transfers all reward tokens to her address and proceeds to sell them through an automated market maker or another system.

Recommendations
Short term, make `callRestricted` `true` by default. Alternatively, consider removing this mechanism altogether to prevent the above scenario, which can occur whenever it is `false`.

Long term, ensure that the contracts are invulnerable to malicious behavior, in the presence of other network contracts, even after initial deployment.

7. Architecture can be simplified

Severity: Informational
Type: Configuration
Target: `contracts/core/MarginCalculator|Controller.sol`

Difficulty: Undetermined
Finding ID: TOB-OPYN-007

Description
The system is composed of many parts, most of which are separate contracts. The `MarginPool`'s configuration as a discrete contract is not necessarily a liability, as it provides separation of concerns between the logic contract (the `Controller`) and the storage of funds (handled by the `MarginPool`).

However, other components, such as the `MarginCalculator`, could benefit from a simpler approach. While this component does contain storage variables, their general purpose is system-level configuration, and they are not expected to undergo frequent updates. As such, the `MarginCalculator` could be refactored into a Solidity library. The configuration options would then be passed to the calculator, either in a struct or individually. That would give the calculator many more `view` and `pure` functions, which would make testing the system with unit tests or Echidna much easier.

Whether the library should be an external one (deployed as an EVM contract) or an internal one (with its logic inserted into the contracts that call it) is a different question. Currently, the `MarginCalculator` is called only from the `Controller`. This suggests that using an internal library is the better approach, although the limitations of EIP-170 will need to be considered.

Another, albeit orthogonal, issue is the use of a mapping to store incrementing vaults and the use of `_checkVaultId` to check index bounds.

```
    /// @dev mapping between an owner address and the number of owner address vaults
    mapping(address => uint256) internal accountVaultCounter;
    /// @dev mapping between an owner address and a specific vault using a vault id
    mapping(address => mapping(uint256 => MarginVault.Vault)) internal vaults;
```

Figure 7.1: Data structures for vault storage.

Instead, vaults could be implemented as arrays or linked lists. Either implementation would be a more semantic approach. Furthermore, Solidity provides checks on array access by default.

Recommendations
Short term, consider making the `MarginCalculator` a library and changing the inner mapping of `vaults` into an array or linked list.

Long term, invest time in thinking through the architecture of the contracts and the data flow to ensure readability, simplicity, and long-term maintainability.

8. Short, long, collateral, and vault data structures may be sparse “arrays”

Severity: Informational
Type: Data Validation
Target: `contracts/core/Controller|MarginVault.sol`

Difficulty: Undetermined
Finding ID: TOB-OPYN-008

Description
The `MarginVault` contains arrays that store short and long oTokens and collateral.

```
    struct Vault {
        address[] shortOtokens;
        address[] longOtokens
        address[] collateralAssets;
        uint256[] shortAmounts;
        uint256[] longAmounts;
        uint256[] collateralAmounts;
    }
```

Figure 8.1: The vault definition (with comments removed).

When a new token or collateral is added, the array is pushed as shown below:

```
            _vault.shortOtokens.push(_shortOtoken);
            _vault.shortAmounts.push(_amount);
```

Figure 8.2: A new `shortOtoken` is added.

However, when one of these assets is removed, the slot that stored it is deleted:

```
            delete _vault.shortOtokens[_index];
```

Figure 8.3: A `shortOtoken` is removed.

This will lead to sparse arrays, which can be confusing to external parties, such as parties seeking to discern the length of these arrays.

Vaults are currently implemented as a mapping (TOB-OPYN-007) but are vulnerable to the same issue—that is, a vault may also be empty, even in the range of `[1, accountVaultCounter]`.

Recommendations
Short term, document the fact that short, long, collateral, and vault data structures may be sparse.

Long term, consider swapping the deleted value and the last value and removing array elements with `.pop()`. Using `.pop()` is the best way to delete the last element and decrease the length of an array. Alternatively, consider using a linked list, which would also solve this issue.

9. Error-prone operate function

Severity: Informational
Type: Data Validation
Target: `contracts/core/Controller.sol`

Difficulty: Undetermined
Finding ID: TOB-OPYN-009

Description
The `operate` function allows a user to compose actions through the `Controller`.

```
    function operate(Actions.ActionArgs[] memory _actions) external nonReentrant
notFullyPaused {
        (bool vaultUpdated, address vaultOwner, uint256 vaultId) = _runActions(_actions);
        if (vaultUpdated) {
            _verifyFinalState(vaultOwner, vaultId);
            vaultLatestUpdate[vaultOwner][vaultId] = now;
        }
    }
```

Figure 9.1: The `operate` function definition.

If a vault is updated, the code will verify its final state. However, the vault may be subject to multiple actions in the interim. The functions that carry out these interim actions will not impose final state invariants, nor will they update the timestamp.

Furthermore, several functions (particularly `settleVault`, but in theory most functions) may leave the vault in a post-state of emptiness, which could exacerbate this issue. The `settleVault` function checks the state before proceeding. However, if the code is refactored and this check is forgotten, it may be possible for a vault to end up in an unintended state and to then be settled (i.e., deleted) before any invariant checks are run.

Recommendations
Short term, consider implementing a check of the invariants before `settleVault` is used.

Long term, consider adding more constraints to the `operate` function, such as invariant checks after every operation.

10. Stablecoin value is assumed to be constant

Severity: High
Type: Data Validation
Target: `contracts/core/Oracle.sol`

Difficulty: High
Finding ID: TOB-OPYN-010

Description
The oracle contract provides a price feed to the system. However, when an asset is marked as stable (more precisely, when `setStablePrice` is called on it), the returned price will always be constant.

```
    function getPrice(address _asset) external view returns (uint256) {
        uint256 price = stablePrice[_asset];

        if (price == 0) {...}

        return price;
    }

[...]

    function getExpiryPrice(address _asset, uint256 _expiryTimestamp) external view returns
(uint256, bool) {
        uint256 price = stablePrice[_asset];
        bool isFinalized = true;

        if (price == 0) {...}

        return (price, isFinalized);
    }

[...]

    function getChainlinkRoundData(address _asset, uint80 _roundId) external view returns
(uint256, uint256) {
        uint256 price = stablePrice[_asset];
        uint256 timestamp = now;

        if (price == 0) {...}

        return (price, timestamp);
    }
```

Figure 10.1: The oracle functions that return `stablePrice`.

If the market price of a stablecoin drops, the system will assume that the peg still holds.

Setting an asset pricer for a stable asset would mitigate this issue. To do so, first set the stable price to 0:

```
    function setStablePrice(address _asset, uint256 _price) external onlyOwner {
        require(assetPricer[_asset] == address(0), "Oracle: could not set stable price for
an asset with pricer");

        stablePrice[_asset] = _price;

        emit StablePriceUpdated(_asset, _price);
    }
```

Figure 10.2: `Oracle.setStablePrice`.

Then set the asset pricer to enable oracle-backed pricing:

```
    function setAssetPricer(address _asset, address _pricer) external onlyOwner {
        require(_pricer != address(0), "Oracle: cannot set pricer to address(0)");
        require(stablePrice[_asset] == 0, "Oracle: could not set a pricer for stable
asset");

        assetPricer[_asset] = _pricer;

        emit PricerUpdated(_asset, _pricer);
    }
```

Figure 10.3: `Oracle.setAssetPricer`.

Exploit Scenario
The public learns that USDC is not sufficiently backed. As a result, the price of USDC plummets. When users redeem long put oTokens, the system will still assume the previously stable USDC price to be constant, even though market conditions dictate that USDC 1 buys only USD 0.80. A user entitled to a USD 100 profit on a long put option will receive the equivalent of USD 80 upon redemption. The drop in USDC also implies that the vault was undercollateralized at its expiration.

Recommendations
Short term, consider having the protocol fetch the price feed for all assets. This will ensure that the system is resilient to even stable price volatility.

Long term, ensure that the system is resilient to all changes in market conditions.

11. Numerous internal and external assumptions

Severity: Informational
Type: Data Validation
Target: `contracts/*`

Difficulty: Undetermined
Finding ID: TOB-OPYN-011

Description
The system makes many assumptions about both the semantics of its internal code and the data provided to it from outside sources. As a result, the system relies on hard-coded values and expects certain input, such as USDC tokens as the strike price-denominating asset, which may inhibit the expansion of token offerings. Additionally, the logic that verifies vault states is spread out over multiple contracts, which affects readability and automated invariant verification. We observed many of these checks and guards only after manually testing transactions.

If additional tokens were used to denominate the strike price of an asset, a number of USDC-specific references in the code would need to be made more general. For example, the `MarginCalculator` would need to be changed:

```
       // the exchangeRate was scaled by 1e8, if 1e8 otoken can take out 1 USDC, the
exchangeRate is currently 1e8
        // we want to return: how much USDC units can be taken out by 1 (1e8 units) oToken
        uint256 collateralDecimals = uint256(ERC20Interface(collateral).decimals());
        return cashValueInCollateral.toScaledUint(collateralDecimals, true);
```

Figure 11.1: References to USDC in the `MarginCalculator`.

Similarly, the `MarginVault` library exposes arrays that can hold many different long and short oTokens as well as different kinds of collateral. However, the logic that verifies that there is at most one asset of each type is located in the `MarginCalculator`. There are also a number of hard-coded zero-index references to `MarginVault` items, such as that in the `Controller` (shown below); these references would need to be made more general if multiple assets were stored in each array in the `MarginVault` (which the contract already implies is the case).

```
        // if vault is partially liquidated (amount of short otoken is still greater than
zero)
        // make sure remaining collateral amount is greater than dust amount
        if (vault.shortAmounts[0].sub(_args.amount) > 0) {
            require(vault.collateralAmounts[0].sub(collateralToSell) >= collateralDust,
"CO34");
        }

        // burn short otoken from liquidator address, index of short otoken hardcoded at 0
```

```
        // this should always work, if vault have no short otoken, it will not reach this
step
        OtokenInterface(vault.shortOtokens[0]).burnOtoken(msg.sender, _args.amount);
```

Figure 11.2: References to zero-index array values.

Another problematic assumption relates to the liveness of bots. When a vault is being settled, the expiry price of the underlying, strike, and collateral assets is fetched. However, if the expiry price is not set by the bot in the pricer, it will not be possible to settle the vault.

```
    function setExpiryPrice(
        address _asset,
        uint256 _expiryTimestamp,
        uint256 _price
    ) external {
        require(msg.sender == assetPricer[_asset], "Oracle: caller is not authorized to set
expiry price");
        require(isLockingPeriodOver(_asset, _expiryTimestamp), "Oracle: locking period is
not over yet");
        require(storedPrice[_asset][_expiryTimestamp].timestamp == 0, "Oracle: dispute
period started");

        storedPrice[_asset][_expiryTimestamp] = Price(_price, now);
        emit ExpiryPriceUpdated(_asset, _expiryTimestamp, _price, now);
    }
```

Figure 11.3: If this function is not called, it will not be possible to settle a vault.

Recommendations

Short term, limit the use of hard-coded values tied to specific external contracts in favor of more general functionalities and assumptions.

Long term, ensure that all assumptions about contract input data are made explicit and are documented, and centralize the logic that validates these assumptions as much as possible to facilitate readability.

12. intToUint returns absolute values

Severity: High | Difficulty: Undetermined
Type: Data Validation | Finding ID: TOB-OPYN-012
Target: `contracts/libs/FixedPointInt256|SignedConverter.sol`

Description
The system uses a custom library, `FixedPointInt256`, for fixed-point integer (FPI) calculations. One of the widely used functions is `toScaledUint`:

```
    function toScaledUint(
        FixedPointInt memory _a,
        uint256 _decimals,
        bool _roundDown
    ) internal pure returns (uint256) {
        uint256 scaledUint;

        if (_decimals == BASE_DECIMALS) {
            scaledUint = _a.value.intToUint();
        } else if (_decimals > BASE_DECIMALS) {
            uint256 exp = _decimals - BASE_DECIMALS;
            scaledUint = (_a.value).intToUint().mul(10**exp);
        } else {
            uint256 exp = BASE_DECIMALS - _decimals;
            uint256 tailing;
            if (!_roundDown) {
                uint256 remainer = (_a.value).intToUint().mod(10**exp);
                if (remainer > 0) tailing = 1;
            }
            scaledUint = (_a.value).intToUint().div(10**exp).add(tailing);
        }

        return scaledUint;
    }
```

Figure 12.1: `FixedPointInt256.toScaledUint`.

The function performs two tasks:

1. It converts the required number of decimals.
2. It turns the integer into an unsigned integer.

For the latter task, it uses `intToUint` from `SignedConverter`:

```
    function intToUint(int256 a) internal pure returns (uint256) {
        if (a < 0) {
            return uint256(-a);
```

```
        } else {
            return uint256(a);
        }
    }
```

Figure 12.2: `SignedConverter.intToUint`.

This function returns the absolute value of the integer. It is used only in the parent function (`toScaledUint`), so that function also returns the absolute value. `toScaledUint` is used on the following results:

1. The result of `_getNakedMarginRequired` in the external `getNakedMarginRequired` function
2. On the result of `_convertAmountOnExpiryPrice` in `getExpiredPayoutRate`
3. On the excess collateral amount in `getExcessCollateral`
4. On the auction price in `_getDebtPrice`

We were not able to demonstrate the presence of bugs in cases 1, 2, and 4 through fuzz testing; however, we believe that this pattern is error-prone and would generally advise against using it.

Exploit Scenario
Eve performs an unintended state transition, meaning that case 1, 2, or 4 leads to a negative result. Instead of causing a revert, the result's absolute value is used, which allows Eve to seize all tokens held by the `MarginPool`.

Recommendations
Short term, change the semantics of `intToUint` such that it reverts if a negative number is passed in. Also consider adding asserts to the code, perhaps asserts executed only during development or testing, to check that FPI values that are assumed to be non-negative are indeed non-negative.

Long term, avoid patterns that use total functions that exhibit unintended behavior when handling unexpected input. It is preferable for functions to revert when handling unexpected input (operating as partial functions).

13. getProceed returns absolute value of required collateral

Severity: Medium
Type: Data Validation
Target: `contracts/core/MarginCalculator|Controller.sol`

Difficulty: Low
Finding ID: TOB-OPYN-013

Description
As mentioned in TOB-OPYN-012, a function that returns the absolute value of a signed integer is used to return the amount of required collateral or excess collateral in `getExcessCollateral`.

```
        FPI.FixedPointInt memory excessCollateral =
collateralAmount.sub(collateralRequired);

        bool isExcess = excessCollateral.isGreaterThanOrEqual(ZERO);
        uint256 collateralDecimals = vaultDetails.hasLong
            ? vaultDetails.longCollateralDecimals
            : vaultDetails.shortCollateralDecimals;
        // if is excess, truncate the tailing digits in excessCollateralExternal calculation
        uint256 excessCollateralExternal = excessCollateral.toScaledUint(collateralDecimals,
isExcess);
        return (excessCollateralExternal, isExcess);
```

Figure 13.1: `MarginCalculator.getExcessCollateral`.

This function is used in the `Controller`'s `getProceed` function:

```
    function getProceed(address _owner, uint256 _vaultId) external view returns (uint256) {
        (MarginVault.Vault memory vault, uint256 typeVault, ) = getVault(_owner, _vaultId);

        (uint256 netValue, ) = calculator.getExcessCollateral(vault, typeVault);
        return netValue;
    }
```

Figure 13.2: `Controller.getProceed`.

It follows that `getProceed` will return a positive value even if a vault is undercollateralized, as it does not use a boolean that could indicate a negative underlying value.

Exploit Scenario
Eve creates a vault that becomes undercollateralized. An external system checks her vault and assumes that it is overcollateralized, leading to unintended consequences.

Recommendations

Short term, consider changing the semantics of `getExcessCollateral` and `getProceed` to return signed integers, with negative numbers representing undercollateralized vaults. Alternatively, ensure that the outputs of `getExcessCollateral` are not discarded or used in logic that requires that function.

Long term, avoid the coercion of negative values into positive ones.

14. Non-ideal handling of arithmetic

Severity: Informational
Type: Data Validation
Target: `contracts/*`

Difficulty: Undetermined
Finding ID: TOB-OPYN-014

Description
Improving certain arithmetic operations used in the contracts under review would increase readability and ensure consistency in the operations.

For example, the handling of rounding directions could be improved. There is some consideration of the rounding direction, such as when the `fromScaledUint` function is used. However, a function used in collateral requirement calculations will round down by default (as shown in the figure below), even though rounding up may be preferable.

```
    function _convertAmountOnLivePrice(
        FPI.FixedPointInt memory _amount,
        address _assetA,
        address _assetB
    ) internal view returns (FPI.FixedPointInt memory) {
        if (_assetA == _assetB) {
            return _amount;
        }
        uint256 priceA = oracle.getPrice(_assetA);
        uint256 priceB = oracle.getPrice(_assetB);
        // amount A * price A in USD = amount B * price B in USD
        // amount B = amount A * price A / price B
        return _amount.mul(FPI.fromScaledUint(priceA, BASE)).div(FPI.fromScaledUint(priceB,
BASE));
    }
```

Figure 14.1: The `_convertAmountOnLivePrice` function used in collateral calculations.

There is also code that uses nested, or repeat, division operations without validating intermediate values (e.g., without ensuring that intermediate results are not null). This could lead to unexpected results.

```
            a = FPI.min(one, _strikePrice.div(_underlyingPrice.div(spotShockValue)));
            b = FPI.max(one.sub(_strikePrice.div(_underlyingPrice.div(spotShockValue))),
ZERO);
```

Figure 14.2: Nested division in `_getNakedMarginRequired`.

Nested division in particular is problematic. Consider a case in which there are three integers, `a`, `b`, and `c`, such that `(a, b, c) = (1000, 3, 2)`. The mathematical result of `a / (b`

`/ c)` would be `a * c / b`, or `1000 * 2 / 3 = 666` $\frac{2}{3}$. The result of a nested division operation would be `floor(1000 / floor(3/ 2))`, or `1000` (an increase of 50% from the mathematical result). However, using `floor(1000 * 2 / 3)` (multiplying by the inner denominator), would result in `666`, a difference of 0.1%.

Recommendations
Short term, consider introducing additional rounding direction options and multiplying by the inner denominator instead of using nested division. Also consider adding asserts to the code, perhaps asserts executed only during development or testing, to check that FPI values are never fully nulled, as such values would probably cause subsequent calculations to be incorrect.

Long term, consider formally verifying the correctness of the `FixedPointInt256` library using a tool such as Manticore.

15. Unclear configuration values standards

Severity: High
Type: Data Validation
Target: `contracts/core/MarginCalculator.sol`

Difficulty: High
Finding ID: TOB-OPYN-015

Description
The `MarginCalculator` contains the following configuration variables that can be updated only by the owner:

```
    uint256 internal oracleDeviation;
    mapping(address => uint256) internal dust;
    mapping(bytes32 => uint256[]) internal timesToExpiryForProduct;
    mapping(bytes32 => mapping(uint256 => uint256)) internal maxPriceAtTimeToExpiry;
    mapping(bytes32 => uint256) internal spotShock;
    OracleInterface public oracle;
```

Figure 15.1: All of the `MarginCalculator`'s non-inherited storage variables.

The code expects the `spotShock`, `maxPriceAtTimeToExpiry`, and `oracleDeviation` variables to be expressed as FPI values, but the setter does not validate their values.

Similarly, the code comments (such as the notice field shown below) repeatedly indicate that `dust` should be set to an FPI value:

```
    /**
     * @notice set dust amount for collateral asset (1e27)
     * @dev can only be called by owner
     * @param _collateral collateral asset address
     * @param _dust dust amount
     */
    function setCollateralDust(address _collateral, uint256 _dust) external onlyOwner {
        require(_dust > 0, "MarginCalculator: dust amount should be greater than zero");

        dust[_collateral] = _dust;
    }
```

Figure 15.2: `MarginCalculator.setCollateralDust`.

In actuality, the code assumes that it is scaled by collateral decimals:

```
                FPI.FixedPointInt memory dustAmount = FPI.fromScaledUint(
                    dust[_vaultDetails.shortCollateralAsset],
                    _vaultDetails.collateralDecimals
                );
```

Figure 15.3: MarginCalculator.sol#L631-L634.

Exploit Scenario
Because of an inaccuracy in the natural specification and the lack of data validation in the setters, an owner sets one of the four abovementioned configuration variables incorrectly by 19 decimal places. As a result, Eve is able to seize the `MarginPool`'s funds.

Recommendations
Short term, make realistic assumptions about the possible values of the `MarginCalculator`'s configuration variables, and add a data validation mechanism to the setters to check that they are within those bounds. Also remove the comments that indicate dust should be scaled by 1e27, when in fact it should be scaled by the collateral decimals.

Long term, validate all data provided to the contracts, including by owners. This will make the system resilient against mistakes made by owners and will limit the operation space of any attacker who gains control of an owner address.

16. Decimals set by Yearn pricer do not reflect changes to yToken decimals

Severity: Informational
Type: Data Validation
Target: `contracts/core/pricers/YearnPricer.sol`

Difficulty: High
Finding ID: TOB-OPYN-016

Description
In the Yearn v2 vaults, the values that the `yToken` `name`, `symbol`, and `decimals` functions return for the current `_bestVault` can change. In particular, a change in the decimals value could affect the scaling of the Yearn pricer in the Gamma Protocol. The decimals are set during the instantiation of the pricer and cannot be updated.

```
    constructor(
        address _yToken,
        address _underlying,
        address _oracle
    ) public {
        require(_yToken != address(0), "YearnPricer: yToken address can not be 0");
        require(_underlying != address(0), "YearnPricer: underlying address can not be 0");
        require(_oracle != address(0), "YearnPricer: oracle address can not be 0");


        ERC20Interface underlyingToken = ERC20Interface(_underlying);
        underlyingDecimals = underlyingToken.decimals();
```

Figure 16.1: The Yearn pricer constructor.

PR 362 resolved this Yearn v2 issue, but the deployed contracts may not have been updated with this change.

Recommendations
Short term, until you are certain that the underlying `yToken` contracts have been updated to resolve this issue, consider having the protocol request the `decimals` value as part of each Yearn pricer request.

Long term, ensure that the properties assumed for external integrations with the Gamma Protocol are documented and verified. If they are not, consider coding more defensively, using weaker sets of assumptions.

17. yToken exchange rates are fully calculated only during withdrawals

Severity: High
Type: Data Validation
Target: `contracts/core/pricers/YearnPricer.sol`

Difficulty: High
Finding ID: TOB-OPYN-017

Description
In the Yearn v2 vaults, each `yToken` represents a share in a vault that invests in strategies, which can have gains and losses. The value of a `yToken` at any given moment can be estimated by using `pricePerShare`, which returns the value of that token in terms of the collateral one would receive when withdrawing it. However, `pricePerShare` takes into account only the current balance of the tokens in the vault as well as the strategies' outstanding debt on the loans issued to them to secure returns.

If a strategy is operating at a loss when the price of a `yToken` is requested, the `pricePerShare` will not reflect that loss until the strategy reports it to the vault, actually realizing the loss. However, if a user withdraws the `yToken` when the losing strategy is in the vault's withdrawal queue, that user may incur the loss before it is declared via a call to `report`.

If an oToken is collateralized by a `yToken`, it may result in inconsistent or incorrect payouts at the time of oToken settlements or redemptions. This is because it may not be possible to determine the true value of the underlying `yToken` without making a withdrawal from a `yToken` to the underlying asset.

Exploit Scenario
Eve notices that a Yearn vault strategy is going to report a significant loss within the next day and purchases an oToken that will expire before it does. When the redemption occurs, Eve is entitled to a fraction of the underlying asset's value (in proportion to the amount of the gain on the oToken), and the vault owner receives a proportionate share of the collateral in `yToken`. Opyn will need to settle the `yToken` collateral against the underlying asset; however, because of the imminent significant loss, Opyn will receive a lower-than-normal payout for the exchange, leaving the `MarginPool` short on that transaction. If the Gamma Protocol functionality requires that the `yToken` be converted to the underlying asset before the vault owner receives collateral, a significant enough loss in the Yearn vault could cause the position to be undercollateralized upon the expiry as well.

Recommendations
Short term, consider using `yTokens` as collateral only for options that use those tokens, rather than a correlated asset, as the underlying asset.

Long term, consider how to handle collateral integrations that can have deflationary or inflationary effects on underlying assets; the Gamma Protocol code will likely need to be

changed to ensure that those integrations function as intended, without violating collateralization invariants.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.
Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.

Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

C. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. An up-to-date version of the checklist can be found in crytic/building-secure-contracts.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, use the below output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of Echidna and
Manticore
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use slither-check-erc to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present. If they are present, make sure that they cannot change over the lifetime of a token.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use slither-prop to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's contract-summary printer to broadly review the code used in the contract.
- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., balances[token_address][msg.sender] may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

D. Fix Log

From June 4 to June 7, 2021, Trail of Bits reviewed Opyn's fixes for the issues identified in this report.

#	Title	Severity	Status
1	Contracts used as dependencies do not track upstream changes	Low	Partially Fixed: PR#422
2	TradeCallee does not validate trade orders	High	Fixed: PR#414
3	Controller _call function lacks a return statement	Informational	Partially Fixed: PR#415
4	Adverse market conditions can eliminate liquidation incentives	High	Not Fixed
5	MarginCalculator defines events but never emits them	Medium	Fixed: PR#416
6	callRestricted is disabled by default	Medium	Fixed: PR#417
7	Architecture can be simplified	Informational	Not Fixed
8	Short, long, collateral, and vault data structures may be sparse "arrays"	Informational	Not Fixed
9	Error-prone operate function	Informational	Not Fixed
10	Stablecoin value is assumed to be constant	High	Not Fixed
11	Numerous internal and external assumptions	Informational	Not Fixed
12	intToUint returns absolute values	High	Not Fixed
13	getProceed returns absolute value of required collateral	Medium	Fixed: PR#418
14	Non-ideal handling of arithmetic	Informational	Partially Fixed: PR#419
15	Unclear configuration values standards	High	Partially Fixed: PR#420

16	Decimals set by Yearn pricer do not reflect changes to yToken decimals	Informational	Fixed: PR#421
17	yToken exchange rates are fully calculated only during withdrawals	High	Not Fixed

Detailed Fix Log for Partially Fixed Issues

Finding 1. The recommendation here was to include dependencies as Git submodules and/or npm packages. This was not observed. However, the version of every external dependency has been added to the source code.

Finding 3. The recommendation here was to return the return value (`returndata`) of the inner call. This was not observed. However, the function signature of `_call` was changed to remove the return value, and this may prevent confusing users and/or developers.

Finding 14. The two cases of nested `divs` have been corrected. However, there are no additional tests (`asserts`), nor rounding options throughout the codebase.

Finding 15. Some natspec and code comments regarding the format of _dust have been corrected. However, there are at least two places where the error is still present: L236 and L643.