SMART CONTRACT AUDIT REPORT

for

OPYN

Prepared By: Shuxiao Wang

PeckShield

February 19, 2021

Document Properties

Client	Opyn
Title	Smart Contract Audit Report
Target	Gamma
Version	1.0
Author	Xudong Shao
Auditors	Xudong Shao, Xuxian Jiang
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 19, 2021	Xudong Shao	Final Release
1.0-rc	February 4, 2021	Xudong Shao	Release Candidate
0.3	February 1, 2021	Xudong Shao	Add More Findings #2
0.2	January 27, 2021	Xudong Shao	Add More Findings #1
0.1	January 20, 2021	Xudong Shao	Initial Draft

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **Gamma Protocol** design document and related smart contract source code, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Gamma Protocol

The Gamma protocol is a capital efficient option protocol that enables sellers to create spreads and other combinations, trade atomically, flash loan mint otokens, assign operators to roll over vaults, create perpetual instruments, and more. The Gamma protocol offers European, cash-settled options that auto-exercise upon expiry. Upon expiry, proceeds for long and short option holders are calculated and can be redeemed at any point after the proceeds have been finalized with a settlement price.

The basic information of the Gamma protocol is as follows:

Table 1.1: Basic Information of Gamma

Item	Description
Name	Opyn
Website	https://opyn.co/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 19, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/opynfinance/GammaProtocol (4eebbcb)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/opynfinance/GammaProtocol (6854a6b)

1.2 About PeckShield

PeckShield Inc. [7] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of the current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [6]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

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

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s), i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Gamma protocol design and implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	■
Low	3	■ ■ ■
Informational	3	■ ■ ■
Total	7	

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

2.2 Key Findings

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

Table 2.1: Key Gamma Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility With Deflationary Tokens in MarginPool::transferToPool()	Coding Practices	Confirmed
PVE-002	Informational	Inconsistent Expiry Check	Coding Practices	Fixed
PVE-003	Informational	Redundant Code in MarginCalculator::getExcessCollateral()	Coding Practices	Confirmed
PVE-004	Medium	Insufficient Collatoral in Controller::_settleVault()	Business Logic	Fixed
PVE-005	Informational	Uncovered Cases in Otoken::_getMonth()	Coding Practices	Confirmed
PVE-006	Low	Incompatibility With _redeem() in PayableProxyController::operate()	Coding Practices	Confirmed
PVE-007	Low	Allowance Increasement in PayableProxyController::operate()	Coding Practices	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Incompatibility With Deflationary Tokens in MarginPool::transferToPool()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MarginPool
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The Gamma protocol enables any user to create option tokens, that represent the right to buy or sell a certain asset in a predefined price (strike price) at expiry. MarginPool is the contract that moves and stores all the ERC20 tokens. Users only need to approve an asset to be used by MarginPool once, then the asset can be used to create multiple different options.

The transferToPool() function transfers an asset from a user to the pool and updates the assetBalance.

```
    function transferToPool(
        address _asset,
        address _user,
        uint256 _amount
    ) public onlyController {
        require(_amount > 0, "MarginPool: transferToPool amount is equal to 0");
        assetBalance[_asset] = assetBalance[_asset].add(_amount);

        // transfer _asset _amount from _user to pool
        ERC20Interface(_asset).safeTransferFrom(_user, address(this), _amount);
        emit TransferToPool(_asset, _user, _amount);
    }
```

Listing 3.1: MarginPool.sol

However, in the cases of deflationary tokens, as shown in the above code snippets, the input `_amount` may not be equal to the received amount due to the charged (and burned) transaction fee. As a result, the above operations may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts in the cases of deflationary tokens.

Recommendation Call `balanceOf` to update the `assetBalance` after token transferation.

Status This issue has been confirmed by the team. However, only the owner can add new tokens to whitelist, the dev team decides to leave it as it is.

3.2 Inconsistent Expiry Check

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Controller`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

As we introduced in Section 3.1, Gamma protocol enables sellers to create spreads and other combinations. The oTokens created by Gamma are cash settled European option which means that all options are automatically exercised at expiry.

The `hasExpired()` function checks if an oToken has expired. The `_getMarginRequired()` function calculates the amount of collateral needed for a vault.

```
    function hasExpired(address _otoken) external view returns (bool) {
        uint256 otokenExpiryTimestamp = OtokenInterface(_otoken).expiryTimestamp();

        return now >= otokenExpiryTimestamp;
    }
```

Listing 3.2: Controller.sol

```
    function _getMarginRequired(MarginVault.Vault memory _vault, VaultDetails memory
        _vaultDetails)
        internal
        view
        returns (FPI.FixedPointInt memory)
    {
        FPI.FixedPointInt memory shortAmount = _vaultDetails.hasShort
            ? FPI.fromScaledUint(_vault.shortAmounts[0], BASE)
            : ZERO;
        FPI.FixedPointInt memory longAmount = _vaultDetails.hasLong
            ? FPI.fromScaledUint(_vault.longAmounts[0], BASE)
```

```
                : ZERO;

        address otokenUnderlyingAsset = _vaultDetails.hasShort
            ? _vaultDetails.shortUnderlyingAsset
            : _vaultDetails.longUnderlyingAsset;
        address otokenCollateralAsset = _vaultDetails.hasShort
            ? _vaultDetails.shortCollateralAsset
            : _vaultDetails.longCollateralAsset;
        address otokenStrikeAsset = _vaultDetails.hasShort
            ? _vaultDetails.shortStrikeAsset
            : _vaultDetails.longStrikeAsset;
        uint256 otokenExpiry = _vaultDetails.hasShort
            ? _vaultDetails.shortExpiryTimestamp
            : _vaultDetails.longExpiryTimestamp;
        bool expired = now > otokenExpiry;
        ......
    }
```

Listing 3.3: MarginCalculator.sol

However, at expiry, the `hasExpired()` function will consider the oToken has expired while the `_getMarginRequired()` function only sets `expired` as `True` after expiry.

Recommendation The `_getMarginRequired()` function should sets `expired` as `True` at expiry.

```
    function _getMarginRequired(MarginVault.Vault memory _vault, VaultDetails memory
        _vaultDetails)
        internal
        view
        returns (FPI.FixedPointInt memory)
    {
        FPI.FixedPointInt memory shortAmount = _vaultDetails.hasShort
            ? FPI.fromScaledUint(_vault.shortAmounts[0], BASE)
            : ZERO;
        FPI.FixedPointInt memory longAmount = _vaultDetails.hasLong
            ? FPI.fromScaledUint(_vault.longAmounts[0], BASE)
            : ZERO;

        address otokenUnderlyingAsset = _vaultDetails.hasShort
            ? _vaultDetails.shortUnderlyingAsset
            : _vaultDetails.longUnderlyingAsset;
        address otokenCollateralAsset = _vaultDetails.hasShort
            ? _vaultDetails.shortCollateralAsset
            : _vaultDetails.longCollateralAsset;
        address otokenStrikeAsset = _vaultDetails.hasShort
    t
            ? _vaultDetails.shortStrikeAsset
            : _vaultDetails.longStrikeAsset;
        uint256 otokenExpiry = _vaultDetails.hasShort
            ? _vaultDetails.shortExpiryTimestamp
            : _vaultDetails.longExpiryTimestamp;
        bool expired = now >= otokenExpiry;
        ......
```

```
    }
```

Listing 3.4: AaveMarket.sol

Status The issue has been fixed in this commit: `8ba3d7b`.

3.3 Redundant Code in MarginCalculator::getExcessCollateral()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MarginCalculator`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

Gamma protocol allows users to mint oTokens as long as they deposit enough long oTokens and collatoral. The `getExcessCollateral()` function returns the amount of collateral that can be removed from an actual or a theoretical vault. It gets the `collateralDecimals` and truncates the tailing digits in `excessCollateralExternal` calculation. If the vault has long collatoral, the `collateralDecimals` is set to `longCollateralDecimals`. If not, it's set to `shortCollateralDecimals`.

```
    function getExcessCollateral(MarginVault.Vault memory _vault) public view returns (
        uint256, bool) {
        // get vault details
        VaultDetails memory vaultDetails = getVaultDetails(_vault);
        // include all the checks for to ensure the vault is valid
        _checkIsValidVault(_vault, vaultDetails);

        // if the vault contains no oTokens, return the amount of collateral
        if (!vaultDetails.hasShort && !vaultDetails.hasLong) {
            uint256 amount = vaultDetails.hasCollateral ? _vault.collateralAmounts[0] :
                0;
            return (amount, true);
        }

        FPI.FixedPointInt memory collateralAmount = ZERO;
        if (vaultDetails.hasCollateral) {
            collateralAmount = FPI.fromScaledUint(_vault.collateralAmounts[0],
                vaultDetails.collateralDecimals);
        }

        // get required margin, denominated in collateral
        FPI.FixedPointInt memory collateralRequired = _getMarginRequired(_vault,
            vaultDetails);
        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);
    }
```

Listing 3.5: MarginCalculator.sol

However, the function can use `vaultDetails.collateralDecimals` to calculate the `excessCollateralExternal` directly.

Recommendation Use `vaultDetails.collateralDecimals` to calculate the `excessCollateralExternal` .

Status This issue has been confirmed by the team. However, this is not a security issue, the dev team decides to leave it as it is.

3.4 Insufficient Collatoral in Controller::_settleVault()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: `Controller`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `Controller` contract is the entry point for all users, it manages all the opened vaults for all sellers, and also takes care of the redeem operation for buyers.

The `_settleVault()` function settles a vault after expiry and removes the net collateral after both long and short oToken payouts have been settled. It calls `getExcessCollateral()` to calculate the exact amount of collatoral returned to users. This function also returns a flag indicating whether there is excess margin in the vault.

```
    function _settleVault(Actions.SettleVaultArgs memory _args) internal onlyAuthorized(
        msg.sender, _args.owner) {
        require(_checkVaultId(_args.owner, _args.vaultId), "Controller: invalid vault id
            ");

```

```
        MarginVault.Vault memory vault = getVault(_args.owner, _args.vaultId);
        bool hasShort = _isNotEmpty(vault.shortOtokens);
        bool hasLong = _isNotEmpty(vault.longOtokens);

        require(hasShort  hasLong, "Controller: Can't settle vault with no otoken");

        OtokenInterface otoken = hasShort
            ? OtokenInterface(vault.shortOtokens[0])
            : OtokenInterface(vault.longOtokens[0]);

        address underlying = otoken.underlyingAsset();
        address strike = otoken.strikeAsset();
        address collateral = otoken.collateralAsset();
        uint256 expiry = otoken.expiryTimestamp();

        require(now >= expiry, "Controller: can not settle vault with un-expired otoken"
            );
        require(
            isSettlementAllowed(underlying, strike, collateral, expiry),
            "Controller: asset prices not finalized yet"
        );

        (uint256 payout, ) = calculator.getExcessCollateral(vault);

        if (hasLong) {
            OtokenInterface longOtoken = OtokenInterface(vault.longOtokens[0]);

            longOtoken.burnOtoken(address(pool), vault.longAmounts[0]);
        }

        delete vaults[_args.owner][_args.vaultId];

        pool.transferToUser(collateral, _args.to, payout);

        emit VaultSettled(_args.owner, _args.to, address(otoken), _args.vaultId, payout)
            ;
    }
```

Listing 3.6: Controller.sol

However, if the price of collatoral drops a lot, it may not be able to pay for the option. Fortunately, only whitelisted oTokens can be deposited as collatoral. The dev team will make sure the `otokenCollateralAsset` is the same with `otokenStrikeAsset`.

Recommendation Make sure the `otokenCollateralAsset` is the same with `otokenStrikeAsset` in `Controller::_depositCollateral()`.

Status The issue has been fixed in this commit: `6854a6b`.

3.5 Uncovered Cases in Otoken::_getMonth()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MPHIssuanceModel01`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

`Otoken` is the ERC20 compatible contract that each represents an option product. All the oTokens have 8 decimals, and the name and symbol of an oToken is determined by the underlying, strike, collateral, expiry and strike price.

In `Otoken` contract, the `_getMonth()` function returns the string representation of a month.

```
    function _getMonth(uint256 _month) internal pure returns (string memory shortString,
        string memory longString) {
        if (_month == 1) {
            return ("JAN", "January");
        } else if (_month == 2) {
            return ("FEB", "February");
        } else if (_month == 3) {
            return ("MAR", "March");
        } else if (_month == 4) {
            return ("APR", "April");
        } else if (_month == 5) {
            return ("MAY", "May");
        } else if (_month == 6) {
            return ("JUN", "June");
        } else if (_month == 7) {
            return ("JUL", "July");
        } else if (_month == 8) {
            return ("AUG", "August");
        } else if (_month == 9) {
            return ("SEP", "September");
        } else if (_month == 10) {
            return ("OCT", "October");
        } else if (_month == 11) {
            return ("NOV", "November");
        } else {
            return ("DEC", "December");
        }
    }
```

Listing 3.7: oToken.sol

However, if the input `_month` is 13, the returned string will be "DEC".

Recommendation Return zero if the input `_month` is out of scope.

Status This issue has been confirmed by the team. However, this is not a security issue, the dev team decides to leave it as it is.

3.6 Incompatibility With _redeem() in PayableProxyController::operate()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `PayableProxyController`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

The `PayableProxyController::operate()` function can be called by users for wrapping/unwrapping ETH before/after interacting with the `Gamma` Protocol. A number of actions can be executed through this function including `_redeem()`.

```
    function operate(Actions.ActionArgs[] memory _actions, address payable _sendEthTo) external payable nonReentrant {
        // create WETH from ETH
        if (msg.value != 0) {
            weth.deposit{value: msg.value}();
        }

        // verify sender
        for (uint256 i = 0; i < _actions.length; i++) {
            Actions.ActionArgs memory action = _actions[i];

            // check that msg.sender is an owner or operator
            if (action.owner != address(0)) {
                require(
                    (msg.sender == action.owner)  (controller.isOperator(action.owner, msg.sender)),
                    "PayableProxyController: cannot execute action "
                );
            }

            if (action.actionType == Actions.ActionType.Call) {
                // our PayableProxy could ends up approving amount > total eth received.
                ERC20Interface(address(weth)).safeIncreaseAllowance(action.secondAddress, msg.value);
            }
        }

        controller.operate(_actions);

```

```
        // return all remaining WETH to the sendEthTo address as ETH
        uint256 remainingWeth = weth.balanceOf(address(this));
        if (remainingWeth != 0) {
            require(_sendEthTo != address(0), "PayableProxyController: cannot send ETH to address zero");

            weth.withdraw(remainingWeth);
            _sendEthTo.sendValue(remainingWeth);
        }
    }
```

Listing 3.8: PayableProxyController.sol

```
    function _redeem(Actions.RedeemArgs memory _args) internal {
        OtokenInterface otoken = OtokenInterface(_args.otoken);

        require(whitelist.isWhitelistedOtoken(_args.otoken), "Controller: otoken is not whitelisted to be redeemed");

        address underlying = otoken.underlyingAsset();
        address strike = otoken.strikeAsset();
        address collateral = otoken.collateralAsset();
        uint256 expiry = otoken.expiryTimestamp();

        require(now >= expiry, "Controller: can not redeem un-expired otoken");
        require(
            isSettlementAllowed(underlying, strike, collateral, expiry),
            "Controller: asset prices not finalized yet"
        );

        uint256 payout = getPayout(_args.otoken, _args.amount);

        otoken.burnOtoken(msg.sender, _args.amount);

        pool.transferToUser(collateral, _args.receiver, payout);

        emit Redeem(_args.otoken, msg.sender, _args.receiver, collateral, _args.amount, payout);
    }
```

Listing 3.9: Controller.sol

However, the `_redeem()` function will burn the token of `msg.sender`. When the users call `Controller` through the `PayableProxyController`, the `msg.sender` will be the `PayableProxyController` contract and the operation will fail because the `PayableProxyController` doesn't have the OTokens in it.

Recommendation Revise `RedeemArgs` to make it compatible with the `PayableProxyController`.

Status This issue has been confirmed by the team. However, this is not a security issue, the dev team decides to leave it as it is.

3.7 Allowance Increasement in PayableProxyController::operate()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `PayableProxyController`
- Category: Coding Practices [3]
- CWE subcategory: CWE-1041 [1]

Description

As we introduced in Section 3.6, multiple actions can be executed through `PayableProxyController::operate()` to interact with the `Gamma` Protocol. This function will increase the weth allowance of `action.secondAddress` by `msg.value`.

```
    function operate(Actions.ActionArgs[] memory _actions, address payable _sendEthTo) external payable nonReentrant {
        // create WETH from ETH
        if (msg.value != 0) {
            weth.deposit{value: msg.value}();
        }

        // verify sender
        for (uint256 i = 0; i < _actions.length; i++) {
            Actions.ActionArgs memory action = _actions[i];

            // check that msg.sender is an owner or operator
            if (action.owner != address(0)) {
                require(
                    (msg.sender == action.owner)   (controller.isOperator(action.owner, msg.sender)),
                    "PayableProxyController: cannot execute action "
                );
            }

            if (action.actionType == Actions.ActionType.Call) {
                // our PayableProxy could ends up approving amount > total eth received.
                ERC20Interface(address(weth)).safeIncreaseAllowance(action.secondAddress, msg.value);
            }
        }

        controller.operate(_actions);

        // return all remaining WETH to the sendEthTo address as ETH
        uint256 remainingWeth = weth.balanceOf(address(this));
        if (remainingWeth != 0) {
```

```
            require(_sendEthTo != address(0), "PayableProxyController: cannot send ETH
                to address zero");

            weth.withdraw(remainingWeth);
            _sendEthTo.sendValue(remainingWeth);
        }
    }
```

Listing 3.10: PayableProxyController.sol

However, if multiple calls are included in the actions set, the operation will end up adding multiple `msg.values` in terms of allowance.

Recommendation Expand the `CallArgs` to include the ether amount that is intended to transfer to callee.

Status This issue has been confirmed by the team. However, this is not a security issue, the dev team decides to leave it as it is.

4 Conclusion

In this audit, we have analyzed the Gamma design and implementation. The system is a capital efficient option protocol that enables sellers to create spreads and other combinations, trade atomically, flash loan mint otokens, assign operators to roll over vaults, create perpetual instruments, and more. During the audit, we notice that the current code base is well structured and neatly organized, and those identified issues are promptly confirmed and fixed.

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

References

[1] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

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