



Moonwell Finance

Smart Contract Audit Report

Prepared by: Halborn

Date of Engagement: January 24th, 2022 - February 1st, 2022

Visit: Halborn.com

DOCUMENT REVISION HISTORY	4
CONTACTS	4
1 EXECUTIVE OVERVIEW	5
1.1 INTRODUCTION	6
1.2 AUDIT SUMMARY	6
1.3 TEST APPROACH & METHODOLOGY	6
RISK METHODOLOGY	7
1.4 SCOPE	9
2 ASSESSMENT SUMMARY & FINDINGS OVERVIEW	10
3 FINDINGS & TECH DETAILS	11
3.1 (HAL-01) ALLOWING ERC777-KIND TOKENS ON PROTOCOL LEADS RE-ENTRANCY - MEDIUM	13
Description	13
Code Location	13
Risk Level	14
Recommendation	14
Remediation Plan	14
3.2 (HAL-02) USE OF DEPRECATED CHAINLINK API - MEDIUM	15
Description	15
Code Location	15
Risk Level	15
Recommendation	15
Remediation Plan	16
3.3 (HAL-03) ASSETS MAY LOCKED DOWN ON GOVERNORALPHA CONTRACT - MEDIUM	17
Description	17

Code Location	17
Proof Of Concept	17
Risk Level	18
Recommendation	18
Reference	19
Remediation Plan	19
3.4 (HAL-04) SHORT CIRCUIT IS NECESSARY FOR GAS OPTIMIZATION - LOW	20
Description	20
Code Location	21
Risk Level	21
Recommendation	22
Remediation Plan	22
3.5 (HAL-05) GOVERNORALPHA DOES NOT CONTROL QUEUED PROPOSALS ON CANCEL METHOD - LOW	23
Description	23
Code Location	23
Risk Level	24
Recommendation	24
Remediation Plan	24
3.6 (HAL-06) MISSING ZERO ADDRESS CHECKS - LOW	25
Description	25
Code Location	25
Risk Level	26
Recommendation	26
Remediation Plan	26
3.7 (HAL-07) MULTIPLE PRAGMA DEFINITION - INFORMATIONAL	27

	Description	27
	Code Location	27
	Risk Level	28
	Recommendation	28
	Remediation Plan	28
3.8	(HAL-08) UNUSED FUNCTION PARAMETERS - INFORMATIONAL	29
	Description	29
	Code Location	29
	Risk Level	29
	Recommendation	29
	Remediation Plan	30
4	AUTOMATED TESTING	31
4.1	STATIC ANALYSIS REPORT	32
	Description	32
	Results	32

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	02/01/2022	Ataberk Yavuzer
0.2	Document Edits	02/02/2022	Ataberk Yavuzer
0.3	Draft Review	02/02/2022	Gabi Urrutia
1.0	Remediation Plan	02/07/2022	Ataberk Yavuzer
1.1	Remediation Plan Review	02/07/2022	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Ataberk Yavuzer	Halborn	Ataberk.Yavuzer@halborn.com



EXECUTIVE OVERVIEW



1.1 INTRODUCTION

Moonwell Finance engaged Halborn to conduct a security audit on their smart contracts beginning on January 24th, 2022 and ending on February 1st, 2022. The security assessment was scoped to the smart contracts provided to the Halborn team.

1.2 AUDIT SUMMARY

The team at Halborn was provided one week for the engagement and assigned a full-time security engineer to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some security risks that were solved and addressed by the Moonwell Finance team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose.
- Smart Contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions([solgraph](#))
- Manual Assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Static Analysis of security for scoped contract, and imported functions.([Slither](#))
- Dynamic Analysis ([ganache-cli](#), [brownie](#), [hardhat](#))

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

- 5 - Almost certain an incident will occur.
- 4 - High probability of an incident occurring.
- 3 - Potential of a security incident in the long term.
- 2 - Low probability of an incident occurring.
- 1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

- 5 - May cause devastating and unrecoverable impact or loss.
- 4 - May cause a significant level of impact or loss.
- 3 - May cause a partial impact or loss to many.
- 2 - May cause temporary impact or loss.
- 1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
----------	------	--------	-----	---------------

10 - CRITICAL

9 - 8 - HIGH

7 - 6 - MEDIUM

5 - 4 - LOW

3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

1. Moonwell Finance Smart Contracts

(a) Repository: [Moonwell Finance - Moonwell Core](#)

(b) Commit ID: [70fb9c4a899ba0cd787855582ea3bd803317f51a](#)

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	3	3	2

LIKELIHOOD

IMPACT

		(HAL-02) (HAL-03)	(HAL-01)	
	(HAL-06)			
(HAL-07) (HAL-08)		(HAL-04) (HAL-05)		

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) ALLOWING ERC777-KIND TOKENS ON PROTOCOL LEADS RE-ENTRANCY	Medium	SOLVED - 02/04/2022
(HAL-02) USE OF DEPRECATED CHAINLINK API	Medium	SOLVED - 02/04/2022
(HAL-03) ASSETS MAY LOCKED DOWN ON GOVERNORALPHA CONTRACT	Medium	SOLVED - 02/04/2022
(HAL-04) SHORT CIRCUIT IS NECESSARY FOR GAS OPTIMIZATION	Low	NOT APPLICABLE
(HAL-05) GOVERNORALPHA DOES NOT CONTROL QUEUED PROPOSALS ON CANCEL METHOD	Low	NOT APPLICABLE
(HAL-06) MISSING ZERO ADDRESS CHECKS	Low	RISK ACCEPTED
(HAL-07) MULTIPLE PRAGMA VERSIONS	Informational	SOLVED - 02/04/2022
(HAL-08) UNUSED FUNCTION PARAMETERS	Informational	SOLVED - 02/04/2022



FINDINGS & TECH DETAILS



3.1 (HAL-01) ALLOWING ERC777-KIND TOKENS ON PROTOCOL LEADS RE-ENTRANCY - MEDIUM

Description:

The ERC777 standard allows the token contract to notify senders and recipients when ERC777 tokens are sent or received from their accounts with function hooks. These hooks are called as callbacks. If the recipient of the token is a smart contract, the smart contract may cause to re-entrancy by calling another transfer function.

During the tests, it was seen that the protocol could be affected by this vulnerability if ERC777-kind tokens are planned to be used. This may cause loss of funds.

Code Location:

Listing 1: MToken.sol (Lines 701,702,717)

```

696             //////////////////////////////////
697             // EFFECTS & INTERACTIONS
698             // (No safe failures beyond this point)
699
700             /* We write previously calculated values into storage */
701             totalSupply = vars.totalSupplyNew;
702             accountTokens[redeemer] = vars.accountTokensNew;
703
704             /* We emit a Transfer event, and a Redeem event */
705             emit Transfer(redeemer, address(this), vars.redeemTokens);
706             emit Redeem(redeemer, vars.redeemAmount, vars.redeemTokens
707             ↪ );
708
709             /* We call the defense hook */
710             comptroller.redeemVerify(address(this), redeemer, vars.
711             ↪ redeemAmount, vars.redeemTokens);
712
713             /*

```

```

712         * We invoke doTransferOut for the redeemer and the
       ↳ redeemAmount.
713         * Note: The mToken must handle variations between ERC-20
       ↳ and GLMR underlying.
714         * On success, the mToken has redeemAmount less of cash.
715         * doTransferOut reverts if anything goes wrong, since we
       ↳ can't be sure if side effects occurred.
716         */
717         doTransferOut(redeemer, vars.redeemAmount);
718
719         return uint(Error.NO_ERROR);

```

Risk Level:

Likelihood - 4

Impact - 3

Recommendation:

The supported tokens should be white-listed to ensure that no hijacking mechanism could be implemented, such as ERC777. Furthermore, check-effect-interactions should be controlled properly to avoid any re-entrancy issue.

Remediation Plan:

SOLVED: Moonwell Team solved this issue by implementing Reentrancy Guard and better check-effect-interaction design to `Comptroller.sol` contract.

Commit ID: `e23657c5fbeb12c7393fa49da6f350dc0bd5114e`

3.2 (HAL-02) USE OF DEPRECATED CHAINLINK API - MEDIUM

Description:

The `ChainlinkOracle` contract uses Chainlink's deprecated API `latestAnswer()`. Such functions might suddenly stop working if Chainlink stopped supporting deprecated APIs. This method will return the last value, but it is possible to check if the data is fresh.

Code Location:

Listing 2: ChainlinkOracle.sol (Lines 57,59)

```
52 function getChainlinkPrice(AggregatorV2V3Interface feed) internal
   ↳ view returns (uint) {
53     // Chainlink USD-denominated feeds store answers at 8
   ↳ decimals
54     uint decimalDelta = uint(18).sub(feed.decimals());
55     // Ensure that we don't multiply the result by 0
56     if (decimalDelta > 0) {
57         return uint(feed.latestAnswer()).mul(10**decimalDelta)
   ↳ ;
58     } else {
59         return uint(feed.latestAnswer());
60     }
61 }
```

Risk Level:

Likelihood - 3

Impact - 3

Recommendation:

It is recommended to use `latestRoundData()` method instead of `latestAnswer()`. This method allows executing some extra validations as shown as below:

Listing 3: Extra Validations (Lines 2,3,4)

```
1 (roundId, rawPrice, , updateTime, answeredInRound) =  
↳ AggregatorV3Interface(feed).latestRoundData();  
2     require(rawPrice > 0, "Chainlink price cannot be lower  
↳ than 0");  
3     require(updateTime != 0, "Round is in incompleted state");  
4     require(answeredInRound >= roundId, "Stale price");
```

Remediation Plan:

SOLVED: This issue was solved by implementing better ChainLink Oracle API call (`latestRoundData()`).

Commit ID: `e23657c5fbeb12c7393fa49da6f350dc0bd5114e`

3.3 (HAL-03) ASSETS MAY LOCKED DOWN ON GOVERNORALPHA CONTRACT – MEDIUM

Description:

Eth sent to Timelock will be locked in current implementation.

Code Location:

Listing 4: GovernorAlpha.sol (Line 205)

```
200 function execute(uint proposalId) external {
201     require(state(proposalId) == ProposalState.Queued, "
    ↳ GovernorAlpha::execute: proposal can only be executed if it is
    ↳ queued");
202     Proposal storage proposal = proposals[proposalId];
203     proposal.executed = true;
204     for (uint i = 0; i < proposal.targets.length; i++) {
205         timelock.executeTransaction(proposal.targets[i],
    ↳ proposal.values[i], proposal.signatures[i], proposal.calldatas[i],
    ↳ proposal.eta);
206     }
207     emit ProposalExecuted(proposalId);
208 }
```

Proof Of Concept:

- Set up the governance contracts (GovernanceAlpha, Timelock).
- Send eth to timelock contract.
- Set up a proposal to send 0.1 eth out. Code snippet in ether.js below. proxy refers to GovernorAlpha.

Listing 5

```
1     await proxy.propose(
2         [signers[3].address],
3         [ethers.utils.parseEther("0.1")],
4         [""];
```

```

5      [ethers.BigNumber.from(0)],
6      "Send funds to 3rd signer"
7  );

```

Listing 6

```

1  await proxy.execute(2); // this fails
2  await proxy.execute(2, {value: ethers.utils.parseEther("0.1")})
↳ // this would work

```

Risk Level:

Likelihood - 3

Impact - 3

Recommendation:

Consider applying the following changes.

Listing 7

```

1  function execute(uint proposalId) external {
2      require(state(proposalId) == ProposalState.Queued, "
↳ GovernorAlpha::execute: proposal can only be executed if it is
↳ queued");
3      Proposal storage proposal = proposals[proposalId];
4      proposal.executed = true;
5      for (uint i = 0; i < proposal.targets.length; i++) {
6          timelock.executeTransaction(proposal.targets[i],
↳ proposal.values[i], proposal.signatures[i], proposal.calldatas[i],
↳ proposal.eta);

```

```
7      }  
8      emit ProposalExecuted(proposalId);  
9  }
```

Reference:

<https://github.com/compound-finance/compound-protocol/pull/177/files>

Remediation Plan:

SOLVED: This issue was solved by removing `payable` keyword and `call.value()` method from the `execute()` function on `GovernorAlpha.sol` contract.

Commit ID: `e23657c5fbeb12c7393fa49da6f350dc0bd5114e`

3.4 (HAL-04) SHORT CIRCUIT IS NECESSARY FOR GAS OPTIMIZATION - LOW

Description:

If `votes` variable is equal to zero on `GovernorAlpha.sol:_castVote()` method contract should short circuit itself to consume less gas. The following lines will be executed even if `votes` amount is zero.

Listing 8: GovernorAlpha.sol

```
280 if (support) {
281     proposal.forVotes = add256(proposal.forVotes, votes);
282 } else {
283     proposal.againstVotes = add256(proposal.againstVotes,
    ↪ votes);
284 }
285
286     receipt.hasVoted = true;
287     receipt.support = support;
288     receipt.votes = votes;
289
290     emit VoteCast(voter, proposalId, support, votes);
291 }
```

Basically, the contract will call more functions such as `add256()` even it is not necessary.

The same issue also exists on `Well.sol:transferFrom()` function. If `rawAmount` parameter is equal to zero, the contract should short-circuit itself to prevent gas consume. The following lines will be executed even if `rawAmount` is equal to zero.

Listing 9: Well.sol

```

162 if (spender != src && spenderAllowance != uint96(-1)) {
163     uint96 newAllowance = sub96(spenderAllowance, amount,
    ↳ "Well::transferFrom: transfer amount exceeds spender allowance");
164     allowances[src][spender] = newAllowance;
165
166     emit Approval(src, spender, newAllowance);
167 }
168
169     _transferTokens(src, dst, amount);
170     return true;

```

Code Location:

Listing 10: Vulnerable Functions

```

1 GovernorAlpha.sol:_castVote(address voter, uint proposalId, bool
    ↳ support)
2 Well.sol:transferFrom(address src, address dst, uint rawAmount)

```

Risk Level:

Likelihood - 3

Impact - 1

Recommendation:

It is suggested to apply the following implementations for functions above.

Listing 11: GovernorAlpha.sol

```

278 uint96 votes = well.getPriorVotes(voter, proposal.startBlock);
279 if (votes == 0) {
280     return;
281 }
282
283 . . .

```

Listing 12: Well.sol

```

165 uint96 amount = safe96(rawAmount, "Well::approve: amount exceeds
    ↳ 96 bits");
166 if (amount == 0) {
167     emit Transfer(src, dst, 0); //emitting event is still
    ↳ necessary for following up the transfer standart.
168     return true;
169 }
170
171 . . .

```

Remediation Plan:

NOT APPLICABLE: This issue was marked as **NOT APPLICABLE** since the recommendation does not fit to intended behavior of Compound Protocol. Furthermore, **Moonwell Team** stay as close to the original contracts as possible, even if they are not completely optimal concerning gas efficiency, so that improvements to the original contracts may be adopted without significant refactoring, and the community can have better certainty that they function similarly to other contracts with the same code.

3.5 (HAL-05) GOVERNORALPHA DOES NOT CONTROL QUEUED PROPOSALS ON CANCEL METHOD - LOW

Description:

The `cancel(uint proposalId)` The cancel function is used to cancel the proposals. There is a check on the contract to do not cancel executed proposals. If the state of a proposal is not `QUEUED` yet, the contract will revert to cancel function. However, it will consume gas to achieve this. There is a missing control on the contract to only cancel `QUEUED` proposals.

Code Location:

Listing 13: GovernorAlpha.sol

```

210 function cancel(uint proposalId) public {
211     ProposalState state = state(proposalId);
212     require(state != ProposalState.Executed, "GovernorAlpha::
    ↳ cancel: cannot cancel executed proposal");
213
214     Proposal storage proposal = proposals[proposalId];
215     require(msg.sender == guardian || well.getPriorVotes(
    ↳ proposal.proposer, sub256(block.number, 1)) < proposalThreshold(),
    ↳ "GovernorAlpha::cancel: proposer above threshold");
216
217     proposal.canceled = true;
218     for (uint i = 0; i < proposal.targets.length; i++) {
219         timelock.cancelTransaction(proposal.targets[i],
    ↳ proposal.values[i], proposal.signatures[i], proposal.calldatas[i],
    ↳ proposal.eta);
220     }
221
222     emit ProposalCanceled(proposalId);
223 }
```


Risk Level:

Likelihood - 3

Impact - 1

Recommendation:

It is recommended to implement an additional check to control only **QUEUED** proposals are sent to this function.

Listing 14: GovernorAlpha.sol (Line 212)

```
210 function cancel(uint proposalId) public {
211     ProposalState state = state(proposalId);
212     require(state == ProposalState.Queued, "GovernorAlpha::
  ↳ cancel: Current proposal is not queued.");
```

Remediation Plan:

NOT APPLICABLE: This issue was marked as **NOT APPLICABLE** since the recommendation does not fit to intended behavior of Compound Protocol.

“A proposal is eligible to be cancelled at any time before its execution, including while queued in the Timelock, using this function.”

3.6 (HAL-06) MISSING ZERO ADDRESS CHECKS - LOW

Description:

Moonwell-Core contracts have multiple input fields on their both public and private functions. Some of these inputs are required as `address` variable.

During the test, it has seen all of these inputs are not protected against using the `address(0)` as the target address. It is not recommended to use zero address as target addresses on the contracts.

Code Location:

Listing 15: Missing Zero Address Checks

```

1 ChainlinkOracle.setAdmin(address).newAdmin (contracts/Chainlink/
↳ ChainlinkOracle.sol#88)
2 Comptroller._setBorrowCapGuardian(address).newBorrowCapGuardian (
↳ contracts/Comptroller.sol#968)
3 Comptroller._setPauseGuardian(address).newPauseGuardian (contracts
↳ /Comptroller.sol#986)
4 Comptroller.setWellAddress(address).newWellAddress (contracts/
↳ Comptroller.sol#1351)
5 MErc20.initialize(address,ComptrollerInterface,InterestRateModel,
↳ uint256,string,string,uint8).underlying_ (contracts/MErc20.sol#21)
6 MToken._setPendingAdmin(address).newPendingAdmin (contracts/MToken
↳ .sol#1144)
7 MErc20Delegator.constructor(address,ComptrollerInterface,
↳ InterestRateModel,uint256,string,string,uint8,address,address,
↳ bytes).admin_ (contracts/MErc20Delegator.sol#31)
8 MErc20Delegator._setImplementation(address,bool,bytes).
↳ implementation_ (contracts/MErc20Delegator.sol#60)
9 MErc20Immutable.constructor(address,ComptrollerInterface,
↳ InterestRateModel,uint256,string,string,uint8,address).admin_ (
↳ contracts/MErc20Immutable.sol#29)
10 MGlimmer.constructor(ComptrollerInterface,InterestRateModel,
↳ uint256,string,string,uint8,address).admin_ (contracts/MGlimmer.
↳ sol#27)

```

```

11 Reservoir.constructor(uint256,EIP20Interface,address).target_ (
    ↳ contracts/Reservoir.sol#32)
12 Timelock.constructor(address,uint256).admin_ (contracts/Timelock.
    ↳ sol#26)
13 Timelock.executeTransaction(address,uint256,string,bytes,uint256).
    ↳ target (contracts/Timelock.sol#81)
14 Unitroller._setPendingImplementation(address).
    ↳ newPendingImplementation (contracts/Unitroller.sol#38)
15 Unitroller._setPendingAdmin(address).newPendingAdmin (contracts/
    ↳ Unitroller.sol#85)

```

Risk Level:

Likelihood - 2

Impact - 2

Recommendation:

It is recommended to implement additional zero address checks to avoid usage of zero addresses contracts.

Remediation Plan:

RISK ACCEPTED: The Moonwell Team accepts the risk of this finding. Except the ChainlinkOracle contract, all the contracts mentioned require the new admin key to execute the `_acceptPendingAdmin` function, which protects against accidental attempts to set the admin or guardian to the zero address. It was decided not to make any changes.

3.7 (HAL-07) MULTIPLE PRAGMA DEFINITION – INFORMATIONAL

Description:

`Moonwell` contracts use different pragma versions. Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly.

Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma, for example, either an outdated pragma version that might introduce bugs that affect the contract system negatively or a recently released pragma version which has not been extensively tested. The latest pragma version (0.8.11) was released in December 2021. Many pragma versions have been lately released, going from version 0.7.x to the recently released version 0.8.x. in just few months.

Reference: [Solidity Releases](#)

In the Solidity GitHub repository, there is a JSON file with all bugs finding in the different compiler versions. It should be noted that pragma 0.6.12 and 0.7.6 are widely used by Solidity developers and have been extensively tested in many security audits.

Reference: [Solidity bugs by version](#)

Code Location:

Different pragma versions in use:

Listing 16

```
1 DAIInterestRateModel.sol - Pragma Version 0.5.16
2 Other Contracts - Pragma Version 0.5.17
```

Risk Level:**Likelihood - 1****Impact - 1****Recommendation:**

Consider locking and using a single pragma version without known bugs for the compiler version. If possible, consider using the latest stable pragma version that has been thoroughly tested to prevent potential undiscovered vulnerabilities, such as a pragma between 0.6.12 - 0.7.6, or the latest pragma 0.8.9 - 0.8.11. For example, after the Solidity v 0.8.x, arithmetic operations revert to underflow and overflow by default. By using this version, utility contracts like SafeMath.sol would not be needed.

Remediation Plan:

SOLVED: This issue was solved by the Moonwell Team. The DAIInterestRateModel.sol contract is not used by the Moonwell community, so this contract has been removed from the repository.

3.8 (HAL-08) UNUSED FUNCTION PARAMETERS – INFORMATIONAL

Description:

During the test, it was determined that a variable on the contract was not used for any purpose, although it was defined on the contract. This situation does not pose any risk in terms of security. But it is important for the readability and applicability of the code.

The `baseRatePerYear` parameter of `updateJumpRateModel` function on `DAIInterestRateModelV3.sol` contract is unused on that function.

Code Location:

Listing 17: DAIInterestRateModelV3.sol (Line 51)

```
51 function updateJumpRateModel(uint baseRatePerYear, uint gapPerYear
↳ , uint jumpMultiplierPerYear, uint kink_) external {
52     require(msg.sender == owner, "only the owner may call this
↳ function.");
53     gapPerTimestamp = gapPerYear / timestampsPerYear;
54     updateJumpRateModelInternal(0, 0, jumpMultiplierPerYear,
↳ kink_);
55     poke();
56 }
```

Risk Level:

Likelihood - 1

Impact - 1

Recommendation:

It is recommended to review `baseRatePerYear` variable, and delete it from the contract if this variable will be remained unused in the future.

Remediation Plan:

SOLVED: This issue has been solved by the [Moonwell Team](#). The [DAIInterestRateModel.sol](#) contract is not used by the [Moonwell](#) community, so this contract has been removed from the repository.



AUTOMATED TESTING



4.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance coverage of certain areas of the scoped contract. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their ABI and binary formats. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

Results:

```

Controller.grantRewardInternal(uint256,address,uint256) (contracts/Controller.sol#1289-1305) sends eth to arbitrary user
  Dangerous calls:
    - user.transfer(amount) (contracts/Controller.sol#1300)
MaxInMllion.repayBehalfExplicit(address,MGlimmer) (contracts/MaxInMllion.sol#37-46) sends eth to arbitrary user
  Dangerous calls:
    - MGlimmer._repayBorrowBehalf.value(borrows)(borrower) (contracts/MaxInMllion.sol#41)
Reference: https://github.com/cryptic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
GovernorAlpha.execute(uint256) (contracts/Governance/GovernorAlpha.sol#200-208) sends eth to arbitrary user
  Dangerous calls:
    - timelock.executeTransaction.value(proposal.values[1])(proposal.targets[1],proposal.values[1],proposal.signatures[1],proposal.calldatas[1],proposal.eta) (contracts/Governance/GovernorAlpha.sol#205)
)
Reference: https://github.com/cryptic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
Controller (contracts/Controller.sol#15-1361) contract sets array length with a user-controlled value:
  - accountAssets[borrower].push(mToken) (contracts/Controller.sol#156)
Controller (contracts/Controller.sol#15-1361) contract sets array length with a user-controlled value:
  - allMarkets.push(mToken(mToken)) (contracts/Controller.sol#94)
Reference: https://github.com/cryptic/slither/wiki/Detector-Documentation#array-length-assignment
MGlimmer.requireMD5Error(uint256,string) (contracts/MGlimmer.sol#156-175) uses a weak PRNG: "fullMessage[i + 3] = bytes1(uint8(48 + (errorCode % 10)))" (contracts/MGlimmer.sol#171)
Reference: https://github.com/cryptic/slither/wiki/Detector-Documentation#weak-PRNG
MERC20Delegator.delegateTo(address,bytes) (contracts/MERC20Delegator.sol#438-439) uses delegatecall to a input-controlled function id
  (success,returnData) = callee.delegatecall(data) (contracts/MERC20Delegator.sol#431)
MERC20Delegator.fallback() (contracts/MERC20Delegator.sol#471-485) uses delegatecall to a input-controlled function id
  (success) = Implementation.delegatecall(msg.data) (contracts/MERC20Delegator.sol#475)
Unittroller.fallback() (contracts/Unittroller.sol#135-147) uses delegatecall to a input-controlled function id
  (success) = controllerImplementation.delegatecall(msg.data) (contracts/Unittroller.sol#137)
PglStakingContractProxy.fallback() (contracts/staking/PglStakingContractProxy.sol#50-61) uses delegatecall to a input-controlled function id
  (success) = Implementation.delegatecall(msg.data) (contracts/staking/PglStakingContractProxy.sol#51)
Reference: https://github.com/cryptic/slither/wiki/Detector-Documentation#controlled-delegatecall
PotLike is re-used:
  - contracts/MoaiDelegate.sol#184-190
  - contracts/DAIInterestRateModelV3.sol#113-121
Reference: https://github.com/cryptic/slither/wiki/Detector-Documentation#name-reused
Controller.grantRewardInternal(uint256,address,uint256) (contracts/Controller.sol#1289-1305) ignores return value by well.transfer(user,amount) (contracts/Controller.sol#1294)
Reservoir.drip() (contracts/Reservoir.sol#46-87) ignores return value by token.transfer(target_,toBrip_) (contracts/Reservoir.sol#64)
PglStakingContract.deposit(uint256) (contracts/staking/PglStakingContract.sol#29-43) ignores return value by pglToken.transferFrom(msg.sender,address(this),pglAmount) (contracts/staking/PglStakingContract.sol#34)
PglStakingContract.redeem(uint256) (contracts/staking/PglStakingContract.sol#50-61) ignores return value by pglToken.transfer(msg.sender,pglAmount) (contracts/staking/PglStakingContract.sol#48)
PglStakingContract.claimMERC20(uint256,address,uint256) (contracts/staking/PglStakingContract.sol#220-228) ignores return value by token.transfer(recipient,amount) (contracts/staking/PglStakingContract.sol#227)
Reference: https://github.com/cryptic/slither/wiki/Detector-Documentation#unchecked-transfer
UnittrollerAdminStorage.controllerImplementation (contracts/ControllerStorage.sol#20) is never initialized. It is used in:
  - Controller.adminOrInitializing() (contracts/Controller.sol#1849-1851)
Reference: https://github.com/cryptic/slither/wiki/Detector-Documentation#uninitialized-state-variables

```

Comptroller.mintVerify(address,address,uint256,uint256) (contracts/Comptroller.sol#254-265) uses a Boolean constant improperly:
 -false (contracts/Comptroller.sol#262)
 Comptroller.borrowVerify(address,address,uint256) (contracts/Comptroller.sol#389-399) uses a Boolean constant improperly:
 -false (contracts/Comptroller.sol#396)
 Comptroller.repayBorrowVerify(address,address,address,uint256,uint256) (contracts/Comptroller.sol#437-454) uses a Boolean constant improperly:
 -false (contracts/Comptroller.sol#441)
 Comptroller.liquidateBorrowVerify(address,address,address,address,uint256,uint256) (contracts/Comptroller.sol#504-523) uses a Boolean constant improperly:
 -false (contracts/Comptroller.sol#520)
 Comptroller.setInterest(address,address,address,address,uint256) (contracts/Comptroller.sol#568-585) uses a Boolean constant improperly:
 -false (contracts/Comptroller.sol#582)
 Comptroller.transferVerify(address,address,address,uint256) (contracts/Comptroller.sol#620-631) uses a Boolean constant improperly:
 -false (contracts/Comptroller.sol#628)
 Merc20Delegate._becomeImplementation(bytes) (contracts/Merc20Delegate.sol#29-30) uses a Boolean constant improperly:
 -false (contracts/Merc20Delegate.sol#25)
 Merc20Delegate._resignImplementation() (contracts/Merc20Delegate.sol#35-42) uses a Boolean constant improperly:
 -false (contracts/Merc20Delegate.sol#37)
 Reference: <https://github.com/cryptic/slither/wiki/Detector-Documentation#misuse-of-a-boolean-constant>

BaseJumpRateModelV2.getSupplyRate(uint256,uint256,uint256,uint256) (contracts/BaseJumpRateModelV2.sol#115-120) performs a multiplication on the result of a division:
 -rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18) (contracts/BaseJumpRateModelV2.sol#118)
 -utilizationRate(cash,borrows,reserves).mul(rateToPool).div(1e18) (contracts/BaseJumpRateModelV2.sol#119)
 DAInterestRateModelV3.getPerfInterestRate() (contracts/DAInterestRateModelV3.sol#82-87) performs a multiplication on the result of a division:
 -pot_dsr().sub(1e27).div(1e9).mul(15) (contracts/DAInterestRateModelV3.sol#83-86)
 DAInterestRateModelV3.poke() (contracts/DAInterestRateModelV3.sol#92-107) performs a multiplication on the result of a division:
 -stabilityFeePerTimeStep = duty.add(jug.base()).sub(1e27).mul(1e18).div(1e27).mul(15) (contracts/DAInterestRateModelV3.sol#94)
 JumpRateModel.getSupplyRate(uint256,uint256,uint256,uint256) (contracts/JumpRateModel.sol#99-104) performs a multiplication on the result of a division:
 -rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18) (contracts/JumpRateModel.sol#102)
 -utilizationRate(cash,borrows,reserves).mul(rateToPool).div(1e18) (contracts/JumpRateModel.sol#103)
 WhitePaperInterestRateModel.getSupplyRate(uint256,uint256,uint256,uint256) (contracts/WhitePaperInterestRateModel.sol#79-84) performs a multiplication on the result of a division:
 -rateToPool = borrowRate.mul(oneMinusReserveFactor).div(1e18) (contracts/WhitePaperInterestRateModel.sol#82)
 -utilizationRate(cash,borrows,reserves).mul(rateToPool).div(1e18) (contracts/WhitePaperInterestRateModel.sol#83)
 Reference: <https://github.com/cryptic/slither/wiki/Detector-Documentation#divide-before-multiply>

EIP20NonStandardInterface (contracts/EIP20NonStandardInterface.sol#8-70) has incorrect ERC20 function interface:EIP20NonStandardInterface.transfer(address,uint256) (contracts/EIP20NonStandardInterface.sol#34)
 EIP20NonStandardInterface (contracts/EIP20NonStandardInterface.sol#8-70) has incorrect ERC20 function interface:EIP20NonStandardInterface.transferFrom(address,address,uint256) (contracts/EIP20NonStandardInterface.sol#48)
 ERC20MS (contracts/FaucetToken.sol#24-28) has incorrect ERC20 function interface:ERC20MS.transfer(address,uint256) (contracts/FaucetToken.sol#25)
 ERC20MS (contracts/FaucetToken.sol#24-28) has incorrect ERC20 function interface:ERC20MS.transferFrom(address,address,uint256) (contracts/FaucetToken.sol#27)
 GemLike (contracts/MdaDelegate.sol#192-196) has incorrect ERC20 function interface:GemLike.approve(address,uint256) (contracts/MdaDelegate.sol#193)
 Reference: <https://github.com/cryptic/slither/wiki/Detector-Documentation#incorrect-erc20-interface>

Comptroller.getHypotheticalAccountLiquidtyInternal(address,MToken,uint256,uint256) (contracts/Comptroller.sol#706-762) uses a dangerous strict equality:
 -vars.oraclePriceMantissa == 0 (contracts/Comptroller.sol#730)
 Comptroller.liquidateBorrowAllowed(address,address,address,uint256) (contracts/Comptroller.sol#464-494) uses a dangerous strict equality:
 -shortfall == 0 (contracts/Comptroller.sol#482)
 ExponentialNoError.mul(uint256,uint256,string) (contracts/ExponentialNoError.sol#150-157) uses a dangerous strict equality:
 -a == 0 || b == 0 (contracts/ExponentialNoError.sol#154)
 ExponentialNoError.mul(uint256,uint256,string) (contracts/ExponentialNoError.sol#150-157) uses a dangerous strict equality:
 -require(bool,string)(c / a == b,errorMessage) (contracts/ExponentialNoError.sol#155)

Reentrancy in Merc20Delegate._setImplementation(address,bool,bytes) (contracts/Merc20Delegate.sol#60-73):
 External calls:
 - delegateToImplementation(abi.encodeWithSignature("_resignImplementation()")) (contracts/Merc20Delegate.sol#64)
 - (success,returnData) = callee.delegatecall(data) (contracts/Merc20Delegate.sol#431)
 State variables written after the call(s):
 - implementation = implementation (contracts/Merc20Delegate.sol#68)
 Reentrancy in Merc20Delegate.constructor(address,ComptrollerInterface,InterestRateModel,uint256,string,string,uint8,address,address,bytes) (contracts/Merc20Delegate.sol#24-52):
 External calls:
 - delegateToImplementation(abi.encodeWithSignature("initialize(address,address,address,uint256,string,string,uint8),underlying_comptroller_,initialExchangeRateMantissa_,name_,symbol_,decimals_)") (contracts/Merc20Delegate.sol#38-49)
 - (success,returnData) = callee.delegatecall(data) (contracts/Merc20Delegate.sol#431)
 - _setImplementation(implementation_,false,becomeImplementationData) (contracts/Merc20Delegate.sol#48)
 - (success,returnData) = callee.delegatecall(data) (contracts/Merc20Delegate.sol#431)
 State variables written after the call(s):
 - admin = admin_ (contracts/Merc20Delegate.sol#51)
 Reentrancy in MToken.liquidateBorrowInternal(address,uint256,MTokenInterface) (contracts/MToken.sol#941-950):
 External calls:
 - error = MTokenCollateral accrueInterest() (contracts/MToken.sol#948)
 - liquidateBorrowFresh(msg.sender,borrower,repayAmount,MTokenCollateral) (contracts/MToken.sol#955)
 - allowed = comptroller.repayBorrowAllowed(address(this),payer,borrower,repayAmount) (contracts/MToken.sol#866)
 - allowed = comptroller.liquidateBorrowAllowed(address(this),address(MTokenCollateral),liquidator,borrower,repayAmount) (contracts/MToken.sol#969)
 - allowed = comptroller.setzeAllowed(address(this),setzerToken,liquidator,borrower,setzeTokens) (contracts/MToken.sol#1075)
 - setzeError = MTokenCollateral.setze(liquidator,borrower,setzeTokens) (contracts/MToken.sol#1022)
 State variables written after the call(s):
 - liquidateBorrowFresh(msg.sender,borrower,repayAmount,MTokenCollateral) (contracts/MToken.sol#955)
 - totalBorrows = vars.totalBorrowNew (contracts/MToken.sol#921)
 - liquidateBorrowFresh(msg.sender,borrower,repayAmount,MTokenCollateral) (contracts/MToken.sol#955)
 - totalReserves = vars.totalReservesNew (contracts/MToken.sol#1118)
 Reentrancy in MToken.redeemFresh(address,uint256,uint256) (contracts/MToken.sol#1014-720):
 External calls:
 - allowed = comptroller.redeemAllowed(address(this),redeemer,vars.redeemTokens) (contracts/MToken.sol#666)
 State variables written after the call(s):
 - accountTokens[redeemer] = vars.accountTokensNew (contracts/MToken.sol#710)
 - totalSupply = vars.totalSupplyNew (contracts/MToken.sol#709)
 Reference: <https://github.com/cryptic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1>

MToken._addReservesFresh(uint256).actualAddAmount (contracts/MToken.sol#1283) is a local variable never initialized
 Comptroller.borrowAllowed(address,address,uint256).err_scope_0 (contracts/Comptroller.sol#368) is a local variable never initialized
 MToken.redeemFresh(address,uint256).vars (contracts/MToken.sol#617) is a local variable never initialized
 MToken.borrowFresh(address,uint256).vars (contracts/MToken.sol#766) is a local variable never initialized
 MToken.mintFresh(address,uint256).vars (contracts/MToken.sol#500) is a local variable never initialized
 MToken.setzeInternal(address,address,uint256).vars (contracts/MToken.sol#1085) is a local variable never initialized
 MToken.repayBorrowFresh(address,address,uint256).vars (contracts/MToken.sol#876) is a local variable never initialized
 Reference: <https://github.com/cryptic/slither/wiki/Detector-Documentation#uninitialized-local-variables>

Comptroller.supportMarket(MToken) (contracts/Comptroller.sol#915-934) ignores return value by MToken.isMToken() (contracts/Comptroller.sol#924)
 MdaDelegate._becomeImplementation(address,address) (contracts/MdaDelegate.sol#42-67) ignores return value by pot.drip() (contracts/MdaDelegate.sol#63)
 MdaDelegate._resignImplementation() (contracts/MdaDelegate.sol#72-92) ignores return value by pot.drip() (contracts/MdaDelegate.sol#81)
 MdaDelegate accrueInterest() (contracts/MdaDelegate.sol#101-107) ignores return value by pot.ike(pot.daddr).drip() (contracts/MdaDelegate.sol#103)
 Merc20.initialize(address,ComptrollerInterface,InterestRateModel,uint256,string,string,uint8) (contracts/Merc20.sol#21-34) ignores return value by EIP20Interface(underlying).totalSupply() (contracts/Merc20.sol#33)
 Reference: <https://github.com/cryptic/slither/wiki/Detector-Documentation#unused-return>

```

Reentrancy in MDaiDelegate._becomeImplementation(address,address) (contracts/MDaiDelegate.sol#42-67):
  External calls:
  - dai = daiJoin.dai() (contracts/MDaiDelegate.sol#46)
  - vat = daiJoin.vat() (contracts/MDaiDelegate.sol#47)
  State variables written after the call(s):
  - daiJoinAddress = daiJoinAddress_ (contracts/MDaiDelegate.sol#51)
  - potAddress = potAddress_ (contracts/MDaiDelegate.sol#52)
  - vatAddress = address(vat) (contracts/MDaiDelegate.sol#53)
Reentrancy in MDaiDelegate.accrueInterest() (contracts/MDaiDelegate.sol#101-107):
  External calls:
  - PotLike(potAddress).drip() (contracts/MDaiDelegate.sol#103)
  State variables written after the call(s):
  - super.accrueInterest() (contracts/MDaiDelegate.sol#106)
    - accrualBlockTimestamp = currentBlockTimestamp (contracts/MToken.sol#453)
  - super.accrueInterest() (contracts/MDaiDelegate.sol#106)
    - borrowIndex = borrowIndexNew (contracts/MToken.sol#454)
  - super.accrueInterest() (contracts/MDaiDelegate.sol#106)
    - totalBorrows = totalBorrowsNew (contracts/MToken.sol#455)
  - super.accrueInterest() (contracts/MDaiDelegate.sol#106)
    - totalReserves = totalReservesNew (contracts/MToken.sol#456)
Reentrancy in MToken.borrowFresh(address,uint256) (contracts/MToken.sol#749-813):
  External calls:
  - allowed = comptroller.borrowAllowed(address(this),borrower,borrowAmount) (contracts/MToken.sol#751)
  State variables written after the call(s):
  - accountBorrows[borrower].principal = vars.accountBorrowsNew (contracts/MToken.sol#801)
  - accountBorrows[borrower].interestIndex = borrowIndex (contracts/MToken.sol#802)
  - totalBorrows = vars.totalBorrowsNew (contracts/MToken.sol#803)
Reentrancy in PglStakingContract.deposit(uint256) (contracts/staking/PglStakingContract.sol#29-43):
  External calls:
  - pglToken.transferFrom(msg.sender,address(this),pglAmount) (contracts/staking/PglStakingContract.sol#34)
  State variables written after the call(s):
  - distributeReward(msg.sender) (contracts/staking/PglStakingContract.sol#39)
    - accrualBlockTimestamp = block.timestamp (contracts/staking/PglStakingContract.sol#165)
  - distributeReward(msg.sender) (contracts/staking/PglStakingContract.sol#39)
    - accruedReward[recipient][i] = accruedReward[recipient][i].add(accruedAmount) (contracts/staking/PglStakingContract.sol#195)
  - distributeReward(msg.sender) (contracts/staking/PglStakingContract.sol#39)
    - rewardIndex[i] = rewardIndex[i].add(accruedPerPGL) (contracts/staking/PglStakingContract.sol#180)
  - distributeReward(msg.sender) (contracts/staking/PglStakingContract.sol#39)
    - supplierRewardIndex[recipient][i] = rewardIndex[i] (contracts/staking/PglStakingContract.sol#196)
  - supplyAmount[msg.sender] = supplyAmount[msg.sender].add(depositedAmount) (contracts/staking/PglStakingContract.sol#42)
  - totalSupplies = totalSupplies.add(depositedAmount) (contracts/staking/PglStakingContract.sol#41)
Reentrancy in MToken.mintFresh(address,uint256) (contracts/MToken.sol#497-562):
  External calls:
  - allowed = comptroller.mintAllowed(address(this),minter,mintAmount) (contracts/MToken.sol#499)
  State variables written after the call(s):
  - accountTokens[minter] = vars.accountTokensNew (contracts/MToken.sol#551)
  - totalSupply = vars.totalSupplyNew (contracts/MToken.sol#550)

```

As a result of the tests completed with the Slither tool, some results were obtained and these results were reviewed by [Halborn](#). In line with the reviewed results, it was decided that some vulnerabilities were false-positive and these results were not included in the report. The actual vulnerabilities found by Slither are already included in the findings on the report.



THANK YOU FOR CHOOSING

// HALBORN

