SMART CONTRACT AUDIT REPORT

for

Augmented Finance

Prepared By: Yiqun Chen

PeckShield

August 28, 2021

Document Properties

Client	Augmented Finance
Title	Smart Contract Audit Report
Target	Augmented Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 28, 2021	Xuxian Jiang	Final Release
1.0-rc1	August 8, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 5

1.1 About Augmented Finance . . . 5

1.2 About PeckShield . . . 6

1.3 Methodology . . . 6

1.4 Disclaimer . . . 7

2 Findings 11

2.1 Summary . . . 11

2.2 Key Findings . . . 12

3 Detailed Results 14

3.1 Inconsistent Rate Scale Of ControlledRewardPool . . . 14

3.2 Potential Reentrancy Risk in BaseTokenLocker . . . 15

3.3 Redundant State/Code Removal . . . 16

3.4 Adjusted Authentication of setLockedAt() . . . 17

3.5 Improved BaseUniswapAdapter::_getAmountsInData() Logic . . . 18

3.6 Lack Of setRewardMinter() in RewardConfigurator . . . 19

3.7 Improved Sanity Checks For System Parameters . . . 20

3.8 Proper initializerRunAlways() Modifier . . . 21

3.9 Possible CooldownTimestamp Manipulation . . . 22

3.10 Incorrect ForwardingRewardPool::receiveBoostExcess() Logic . . . 23

3.11 Possible Fund Loss From (Permissive) Smart Wallets With Allowances to LendingPool 24

3.12 Incorrect isLiquidationEnabled() Logic . . . 26

3.13 Consistent Use of notFlashloaning . . . 27

3.14 Possible Costly StakeToken From Improper Pool Initialization . . . 28

3.15 Trust Issue Of Admin Keys . . . 29

3.16 Flashloan-Lowered StableBorrowRate For Mode-Switching Users . . . 31

3.17 Accommodation of Non-ERC20-Compliant Tokens . . . 33

3.18 Possible DoS On Forced Destruction Of LendingPool . . . 35

4 Conclusion **37**

References **38**

1 Introduction

Given the opportunity to review the `Augmented Finance` 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. `Augmented Finance` presents a unique, robust DeFi lending protocol that allows users to freely deposit virtual assets as collateral in order to borrow virtual assets. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed. Our results show that the given version of smart contracts was improved as several issues related to either security or performance were fixed and resolved. This document outlines our audit results.

1.1 About Augmented Finance

`Augmented Finance` is an autonomous non-custodial liquidity protocol for earning high interest on deposits and borrowing digital assets with low rates. It allows users to freely deposit virtual assets as collateral in order to borrow virtual assets and, the interest is automatically calculated by the smart contract protocol. Inspired by `Aave`, `Compound`, and `Curve`, `Augmented Finance` is designed as a commodity for the benefit of the DeFi ecosystem. It is designed with a protocol token `AGF` that can be used starting from governance voting, staking and yield boosting, to more utilities as `Augmented Finance` usage grows.

Table 1.1: Basic Information of Augmented Finance

Item	Description
Name	Augmented Finance
Website	https://augmented.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 28, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that `Qubit` assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://augmented-finance/augmented-finance-protocol.git (2a48d41)

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

- https://augmented-finance/augmented-finance-protocol.git (16b0ccf)

1.2 About PeckShield

PeckShield Inc. [18] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of 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	*High*	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	*Medium*	*Low*
			Likelihood	

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [17]:

- 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 checklist of 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 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) [16], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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) or blockchain software, 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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Augmented Finance` protocol. 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	8	■■■■■■■■
Low	6	■■■■■■
Informational	1	■
Undetermined	1	■
Total	18	

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. The implementation has been improved by resolving the identified issues (shown in Table 2.1), including 2 high-severity vulnerabilities, 8 medium-severity vulnerabilities, 6 low-severity vulnerabilities, 1 informational recommendation, and 1 undetermined issue.

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.

Table 2.1: Key Augmented Finance Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Inconsistent Rate Scale Of ControlledRewardPool	Numeric Errors	Fixed
PVE-002	Low	Potential Reentrancy Risk in BaseTokenLocker	Time and State	Fixed
PVE-003	Informational	Redundant State/Code Removal	Coding Practice	Fixed
PVE-004	Medium	Adjusted Authentication of setLockedAt()	Security Features	Fixed
PVE-005	Low	Improved BaseUniswapAdapter::_getAmountsInData() Logic	Coding Practice	Fixed
PVE-006	Low	Lack Of setRewardMinter() in RewardConfigurator	Business Logic	Fixed
PVE-007	Low	Improved Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-008	Medium	Proper initializerRunAlways() Modifier	Coding Practice	Fixed
PVE-009	High	Possible CooldownTimestamp Manipulation	Business Logic	Fixed
PVE-010	Low	Incorrect ForwardingRewardPool::receiveBoostExcess() Logic	Business Logic	Fixed
PVE-011	Low	Possible Fund Loss From (Permissive) Smart Wallets With Allowances to LendingPool	Business Logic	Resolved
PVE-012	Medium	Incorrect isLiquidationEnabled() Logic	Business Logic	Fixed
PVE-013	Undetermined	Consistent Use of notFlashloaning	Time and State	Fixed
PVE-014	Medium	Possible Costly StakeToken From Improper Pool Initialization	Time and State	Fixed
PVE-015	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-016	Medium	Flashloan-Lowered StableBorrowRate For Mode-Switching Users	Time and State	Resolved
PVE-017	Medium	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-018	High	Possible DoS On Forced Destruction Of LendingPool	Business Logic	Fixed

3 Detailed Results

3.1 Inconsistent Rate Scale Of ControlledRewardPool

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: ControlledRewardPool
- Category: Numeric Errors [15]
- CWE subcategory: CWE-190 [3]

Description

The Augmented Finance protocol is a DeFi lending protocol that is inspired from Aave with a variety of improvements and new features. During our analysis, we notice the current reward pools have the advanced mechanism to allow for flexible setting of reward rates.

To elaborate, we show below the _setRate() function. As the name indicates, it supports a new reward rate to be assigned. It comes to our attention that when the reward pool is paused, the internal storage state _pausedRated (line 88) is used to save the current reward rate, which is not scaled up to avoid possible precision loss. However, when it is later resumed, the effective reward rate is enforced via internalSetRate(_pausedRate) (line 138), which assumes the rate is already scaled up (similar to line 91). Note the constructor function assumes the initialRate is already scaled up with the multiplication of _rateScale.

```
86   function _setRate(uint256 rate) private {
87     if (isPaused()) {
88       _pausedRate = rate;
89       return;
90     }
91     internalSetRate(rate.rayMul(_rateScale));
92   }
```

Listing 3.1: ControlledRewardPool::_setRate()

```
132  function internalPause(bool paused) internal virtual {
133    if (paused) {
      _pausedRate = internalGetRate();
      internalSetRate(0);
      return;
    }
    internalSetRate(_pausedRate);
  }
```

Listing 3.2: `ControlledRewardPool::internalPause()`

Recommendation Be consistent in the internal `pausedRate` state to always have the same `_rateScale`.

Status The issue has been fixed by this commit: `9f359e5`.

3.2 Potential Reentrancy Risk in BaseTokenLocker

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `BaseTokenLocker`
- Category: Time and State [14]
- CWE subcategory: CWE-682 [7]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [20] exploit, and the recent `Uniswap/Lendf.Me` hack [19].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `BaseTokenLocker` as an example, the `lock()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (occurs in the internal `internalLock()` handler 106) starts before effecting the update on internal states, hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
97   function lock(
98     uint256 underlyingAmount,
    uint32 duration,
    uint256 referral
  ) external returns (uint256) {
    require(underlyingAmount > 0, 'ZERO_UNDERLYING');
    //    require(duration > 0, 'ZERO_DURATION');

    (uint256 stakeAmount, uint256 recoverableError) =
      internalLock(msg.sender, msg.sender, underlyingAmount, duration, referral, true);

    revertOnError(recoverableError);
    return stakeAmount;
  }
```

Listing 3.3: `PoolService::lock()`

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`. However, it is important to take precautions in making use of `nonReentrant` to block possible `re-entrancy`.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`. The affected functions include `lock()`, `lockExtend()`, and `lockAdd()`.

Status The issue has been fixed by the following pull request: `91`.

3.3 Redundant State/Code Removal

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A

- Target: `Multiple Contracts`
- Category: Coding Practices [12]
- CWE subcategory: CWE-563 [6]

Description

The `Augmented Finance` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the `BaseTokenLocker` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `BaseTokenLocker` contract, it has defined a public function `totalSupply()`, which still contains the debug-related calls (line 394). This debug-related calls are currently present in a number of contracts and they should be removed before deployment.

```
function totalSupply() public view override returns (uint256 totalSupply_) {
  (uint32 fromPoint, uint32 tillPoint, ) =
    getScanRange(uint32(block.timestamp / _pointPeriod), 0);

  totalSupply_ = _stakedTotal;

  console.log('totalSupply', fromPoint, tillPoint, totalSupply_);

  if (tillPoint == 0) {
    return totalSupply_;
  }
  ...
  }
```

Listing 3.4: `BaseTokenLocker::totalSupply()`

In addition, the contract `TeamRewardPool` contains an unused storage state `_accumRate` that can be removed as well.

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status The issue has been fixed by this commit: `5078a06`.

3.4 Adjusted Authentication of setLockedAt()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `TeamRewardPool`
- Category: Security Features [10]
- CWE subcategory: CWE-287 [4]

Description

The `Augmented Finance` protocol has a built-in incentive mechanism to reward protocol users upon a variety of protocol operations, such as `mint()`, `redeem()`, `borrow()`, and `repay()`. It also designs the necessary incentive mechanism for the team. In the following, we examine the `TeamRewardPool` contract.

To elaborate, we show below the `setUnlockedAt()` routine. It comes to our attention that it currently allows the team manager to adjust the reward lockup timestamp. This is inappropriate as only the controller should be able to adjust the reward lockup timestamp.

```
function setUnlockedAt(uint32 at) external onlyTeamManagerOrController {
  require(at > 0, 'unlockAt is required');
  // console.log('setUnlockedAt', _lockupTill, getCurrentTick(), at);
```

```
    require(_lockupTill == 0  _lockupTill >= getCurrentTick(), 'lockup is finished');
    _lockupTill = at;
  }
```

Listing 3.5: `TeamRewardPool::setUnlockedAt()`

Recommendation Revise the `onlyTeamManagerOrController` modifier of the above `setUnlockedAt()` routine to be `onlyController`.

Status The issue has been fixed by this commit: `2b216a6`.

3.5 Improved BaseUniswapAdapter::_getAmountsInData() Logic

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseUniswapAdapter`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned earlier, `Augmented` is inspired from `Aave` with shared components. In particular, there are a number of adapters that have been provided to facilitate the interaction with external DEX engines, including `Uniswap`. While examining the related adapter operations, we notice the current logic of `BaseUniswapAdapter` can be improved.

To elaborate, we show below the related `_getAmountsInData()`. The current implementation aims to compute the minimum input asset amount required to buy the given output asset amount. With the flashloan-related premium charges, it needs to adjust accordingly to compute the required `amountIn`. Currently, it is computed as `amountIn = amountOut.add(amountOut.mul(FLASHLOAN_PREMIUM_TOTAL).div(10000))`, which needs to be adjusted as the following: `amountIn = amountOut.mul(10000).div(10000.sub(FLASHLOAN_PREMIUM_TOTAL))`.

```
  function _getAmountsInData(
    address reserveIn,
    address reserveOut,
    uint256 amountOut
  ) internal view returns (AmountCalc memory) {
    if (reserveIn == reserveOut) {
      // Add flash loan fee
      uint256 amountIn = amountOut.add(amountOut.mul(FLASHLOAN_PREMIUM_TOTAL).div(10000)
          );
      uint256 reserveDecimals = _getDecimals(reserveIn);
```

```
442        address[] memory path = new address[](1);
443        path[0] = reserveIn;
444
445        return
446          AmountCalc(
447            amountIn,
448            amountOut.mul(10**18).div(amountIn),
449            _calcUsdValue(reserveIn, amountIn, reserveDecimals),
450            _calcUsdValue(reserveIn, amountOut, reserveDecimals),
451            path
452          );
453      }
454
455      (uint256[] memory amounts, address[] memory path) =
456        _getAmountsInAndPath(reserveIn, reserveOut, amountOut);
457
458      // Add flash loan fee
459      uint256 finalAmountIn = amounts[0].add(amounts[0].mul(FLASHLOAN_PREMIUM_TOTAL).div
          (10000));
460      ...
461  }
```

Listing 3.6: `BaseUniswapAdapter::_getAmountsInData()`

Note that the same adjustment is also applicable to the internal `finalAmountIn` computation (line 459).

Recommendation Proper take into account the flashloan-related premium fee in `_getAmountsInData()`.

Status The issue has been fixed by this commit: `cb7d4a6`.

3.6 Lack Of setRewardMinter() in RewardConfigurator

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `RewardConfigurator`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

Description

To effectively manage or configure different reward pools, the `Augmented Finance` protocol has a `RewardConfigurator` contract to dynamically adjust or configure current reward pools. While examining current configuration choices, we notice the current `RewardConfigurator` should be extended with new functionality.

To elaborate, we show below the `setRewardMinter()` function from the `BaseRewardController` contract. This function allows for the update of a new `reward minter`. However, this setting cannot be activated from the current `RewardConfigurator`.

```
function setRewardMinter(IRewardMinter minter) external override onlyConfigurator {
  _rewardMinter = minter;
}
```

Listing 3.7: `BaseRewardController::setRewardMinter()`

Recommendation Extend the current `RewardConfigurator` contract to add the support of dynamic update of reward minters, i.e., `setRewardMinter()`.

Status The issue has been fixed by this commit: `d7496f6`.

3.7 Improved Sanity Checks For System Parameters

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Augmented` protocol is no exception. Specifically, if we examine the `SlashableStakeTokenBase` contract, it has defined a number of protocol-wide risk parameters, such as `_cooldownPeriod` and `_unstakePeriod`. In the following, we show the corresponding routines that allow for their changes.

```
function setCooldown(uint32 cooldownPeriod, uint32 unstakePeriod)
  external
  override
  aclHas(AccessFlags.STAKE_ADMIN)
{
  _cooldownPeriod = cooldownPeriod;
  _unstakePeriod = unstakePeriod;
}
```

Listing 3.8: SlashableStakeTokenBase::setCooldown()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `_unstakePeriod` may enforce an unreasonably long lockup period, hence incurring cost to users or hurting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed by this commit: `92abc09`.

3.8 Proper initializerRunAlways() Modifier

- ID: PVE-008
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `VersionedInitializable`
- Category: Coding Practices [12]
- CWE subcategory: CWE-1041 [1]

Description

The `Augmented` protocol makes certain extensions on the initialization logic, including the new `initializerRunAlways` modifier. Note that unlike constructors, the `initializer` functions must be manually invoked. And this applies both to deploying an `Initializable` contract, as well as extending an `Initializable` contract via inheritance. It should be emphasized that when used with inheritance, parent initializers with `initializerRunAlways` modifier are not protected from multiple calls by another initializer.

To elaborate, we show below the related `initializerRunAlways` modifier. It comes to our attention that it contains the inclusion of body functions twice (lines 54 and 59). The two-time invocation may bring unexpected execution results from the included function body.

```
51  modifier initializerRunAlways(uint256 localRevision) {
52    uint256 topRevision = getRevision();
53    (bool initializing, bool skip) = _preInitializer(localRevision, topRevision);
54    _;
55
56    if (!skip) {
57      lastInitializingRevision = localRevision;
58    }
59    _;
60    if (!skip) {
61      lastInitializedRevision = localRevision;
62    }
63
64    if (!initializing) {
65      lastInitializedRevision = topRevision;
66      lastInitializingRevision = 0;
67    }
}
```

Listing 3.9: `VersionedInitializable::initializerRunAlways()`

Recommendation Revise the `initializerRunAlways` modifier to include the function body only once.

Status The issue has been fixed by this commit: `70df017`.

3.9 Possible CooldownTimestamp Manipulation

- ID: PVE-009
- Severity: High
- Likelihood: High
- Impact: High
- Target: `SlashableStakeTokenBase`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

Description

As mentioned in Section 3.1, the `Augmented Finance` protocol will reward participating users if they stake their tokens to receive pro-rata staking rewards. In order to prevent possible flashloan-assisted front-running attacks that may claim the majority of rewards, the staking logic is designed to have a cooldown period for staked assets. For each account, the associated cooldown period is recorded internally as `_stakersCooldowns`.

When there is a stake operation, the staking user's cooldown timestamp is properly updated. When the pool token is transferred, the receiver's cooldown timestamp will also be updated. The new cooldown timestamp is calculated in the following `getNextCooldown()` routine.

```
function getNextCooldown(
  uint32 fromCooldownPeriod,
  uint256 amountToReceive,
  address toAddress,
  uint256 toBalance
) public returns (uint32) {
  uint32 toCooldownPeriod = _stakersCooldowns[toAddress];
  if (toCooldownPeriod == 0) {
    return 0;
  }

  uint256 minimalValidCooldown = block.timestamp.sub(_cooldownPeriod).sub(
      _unstakePeriod);

  if (minimalValidCooldown > toCooldownPeriod) {
    toCooldownPeriod = 0;
  } else {
    if (minimalValidCooldown > fromCooldownPeriod) {
      fromCooldownPeriod = uint32(block.timestamp);
    }

    if (fromCooldownPeriod < toCooldownPeriod) {
      return toCooldownPeriod;
    } else {
      toCooldownPeriod = uint32(
        (amountToReceive.mul(fromCooldownPeriod).add(toBalance.mul(toCooldownPeriod)))
            .div(
          amountToReceive.add(toBalance)
        )
      );
    }
  }
  _stakersCooldowns[toAddress] = toCooldownPeriod;

  return toCooldownPeriod;
}
```

Listing 3.10: SlashableStakeTokenBase:getNextCooldown()

If a staking user has not passed the cooldown timestamp, the staked funds will be locked inside the staking contract. It comes to our attention that this above `getNextCooldown()` routine is public, which means any one is able to call it. Also, it surprisingly updates the given `toAddress`'s cooldown timestamp directly. In other words, a malicious actor may simply lock another victim's staking funds inside the contract.

Recommendation Restrict the `getNextCooldown()` call or make the function view-only.

Status The issue has been fixed by this commit: `b3669e9`.

3.10 Incorrect ForwardingRewardPool::receiveBoostExcess() Logic

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `ForwardingRewardPool`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

Description

The `Augmented Finance` protocol has designed a number of reward pools. Among existing reward pools, `ForwardingRewardPool` has a built-in reward provider. While examining its implementation, we notice its logic on the provided public function `receiveBoostExcess()` should be revised.

To elaborate, we show below the `receiveBoostExcess()` function. It simply delegates the call to the reward provider to receive the intended boost excess. However, the amount has been scaled up by `scaleRate(amount)` (line 102). Our analysis shows that the `scaleRate()` operation should not be applied. Otherwise, the boost excess becomes huge and likely messes up the internal accounting.

```
101   function receiveBoostExcess(uint256 amount, uint32 since) external override
          onlyController {
102     IBoostExcessReceiver(address(_provider)).receiveBoostExcess(scaleRate(amount), since
            );
103   }
```

Listing 3.11: `ForwardingRewardPool::receiveBoostExcess()`

Recommendation Remove the `scaleRate()` operation before passing the `receiveBoostExcess()` call to the internal reward provider.

Status The issue has been fixed as the related `ForwardingRewardPool` pool is removed in the new version.

3.11 Possible Fund Loss From (Permissive) Smart Wallets With Allowances to LendingPool

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LendingPool`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

Description

The `Augmented Finance` protocol inherits the core functionalities from `Aave`. Among all core functionalities, `flashloan` is a disruptive one that allows users to borrow from the reserves within a single transaction, as long as the user returns the borrowed amount plus additional premium. In this section, we report an issue related to the `flashloan` feature. The `flashloan` feature improves earlier versions by seamlessly integrating the borrow functionality to avoid returning back the flashloan within the same transaction.

```
function _flashLoan(
  FlashLoanLocalVars memory vars,
  address[] calldata assets,
  uint256[] calldata amounts,
  uint256[] calldata modes,
  bytes calldata params,
542      uint16 flashLoanPremium
543    ) private {
544      ValidationLogic.validateFlashloan(assets, amounts);
545
546      (address[] memory aTokenAddresses, uint256[] memory premiums) =
547        _flashLoanPre(address(vars.receiver), assets, amounts, flashLoanPremium);
548
549      require(
550        vars.receiver.executeOperation(assets, amounts, premiums, msg.sender, params),
551        Errors.LP_INVALID_FLASH_LOAN_EXECUTOR_RETURN
552      );
553
554      _flashLoanPost(vars, assets, amounts, modes, aTokenAddresses, premiums);
555    }
```

Listing 3.12: LendingPool::flashLoan()

To elaborate, we show above the code snippet of `flashLoan()` behind the feature. This particular routine implements the `flashloan` feature in a straightforward manner: It firstly transfers the funds to the specified receiver, then invokes the designated operation (`executeOperation` - line 550), next transfers back the funds from the receiver or creates an equivalent borrow.

However, our analysis shows that the above logic may be abused to cause fund loss of an innocent user if the user previously specified certain allowances to `LendingPool`. Specifically, if a flashloan is launched by specifying the innocent user an the `receiverAddress` argument, the `flashLoan()`) execution follows the logic by firstly transferring the loan amount to `receiverAddress`, invoking `executeOperation()` on the receiver, and then transferring the `amountPlusPremium` (no larger than the allowed spending amount) from the receiver back to the pool. Note that this flashloan is not initiated by the `receiverAddress`, who unfortunately pays the premium associated with the flashloan. We need to mention that the `executeOperation()` call will be invoked on the given `receiverAddress`. The compiler will place a sanity check in ensuring the `receiverAddress` is indeed a contract, hence restricting the attack vector only applicable to contract-based smart wallets. Also, current smart wallets may have a fallback routine, which unlikely returns `true` to allow the `executeOperation()` call to proceed without being reverted.[1]

Recommendation Revisit the design of affected routines in possibly avoiding initiating the `transferFrom()` call from the lending pool. Moreover, the revisited design may validate the `executeOperation()` call so that it is required to successfully transfer back the expected assets, if any.

Status The issue has been resolved as it is considered that a smart wallet may not return true to allow `executeOperation()` to proceed.

[1]An example is those smart wallets in `InstaDApp()`, a popular portal that simplifies the needs for DeFi users.

3.12 Incorrect isLiquidationEnabled() Logic

- ID: PVE-012
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `LendingPool`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

Description

The `Augmented Finance` protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., `mint()/redeem()` and `borrow()/repay()`. In the following, we examine one specific functionality, i.e., liquidation.

In particular, the `Augmented Finance` protocol has abstracted the liquidation and flashloan functionalities as standalone features that can be dynamically turned on or off. To elaborate, we show below the two view routines `isFlashLoanEnabled()/isLiquidationEnabled()` to query for the current configuration. It comes to our attention that the `isLiquidationEnabled()` function implements an incorrect logic in evaluating `!_flashloanDisabled` (line 1060), which should be `!_liquidationDisabled`.

```
1055  function isFlashLoanEnabled() external view returns (bool) {
1056    return !_flashloanDisabled;
1057  }

1059  function isLiquidationEnabled() external view returns (bool) {
1060    return !_flashloanDisabled;
1061  }
```

Listing 3.13: `LendingPool::isFlashLoanEnabled()/isLiquidationEnabled()`

Recommendation Correct the flawed logic of `isLiquidationEnabled()`.

Status The issue has been fixed by this commit: `d84dc3c`.

3.13 Consistent Use of notFlashloaning

- ID: PVE-013
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: LendingPool
- Category: Time and State [14]
- CWE subcategory: CWE-682 [7]

Description

As mentioned in Section 3.11, the Augmented Finance protocol allows the pooled assets for flashloans. With the concerns that the flashloan funds may be used against the protocol itself, the protocol provides an internal state nestedFlashLoanCalls to keep track of nested levels of flashloan calls.

To elaborate, we show below this notFlashloaning modifier, which essentially prevents the call when there is an active flashloan. However, our analysis shows that this modifier is only used in two basic operations, i.e., deposit() and borrow(), but not in other operations, including withdraw(), repay(), liquidationCall(), and rebalanceStableBorrowRate(). For consistency and extra precaution, we suggest to add the notFlashloaning modifier to those above functions.

```
modifier notFlashloaning() {
  require(_nestedFlashLoanCalls == 0, Errors.LP_FLASH_LOAN_RESTRICTED);
  _;
}
```

Listing 3.14: LendingPool::notFlashloaning()

Recommendation Apply notFlashloaning for all related LendingPool operations except the flashloan() call.

Status The issue has been fixed by this commit: 14f24a2.

3.14 Possible Costly StakeToken From Improper Pool Initialization

- ID: PVE-014
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `SlashableStakeTokenBase`
- Category: Time and State [11]
- CWE subcategory: CWE-362 [5]

Description

As mentioned earlier, the `Augmented Finance` protocol will reward participating users if they stake their tokens to receive pro-rata staking rewards. In order to prevent possible flashloan-assisted front-running attacks that may claim the majority of rewards, the staking logic is designed to have a cooldown period for staked assets.

To elaborate, we show below the `internalStake()` routine. This routine is used for liquidity providers to deposit desired liquidity and get respective pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
function internalStake(
  address from,
  address to,
  uint256 underlyingAmount,
  uint256 referral,
  bool transferFrom
) internal returns (uint256 stakeAmount) {
  require(underlyingAmount > 0, Errors.VL_INVALID_AMOUNT);
  uint256 oldReceiverBalance = balanceOf(to);
  stakeAmount = underlyingAmount.percentDiv(exchangeRate());

  _stakersCooldowns[to] = getNextCooldown(0, stakeAmount, to, oldReceiverBalance);
  ...
}
```

Listing 3.15: `SlashableStakeTokenBase::internalStake()`

```
function exchangeRate() public view override returns (uint256) {
  uint256 total = totalSupply();
  if (total == 0) {
    return PercentageMath.ONE; // 100%
  }
  return _stakedToken.balanceOf(address(this)).percentOf(total);
}
```

Listing 3.16: `SlashableStakeTokenBase::exchangeRate()`

Specifically, when the pool is being initialized, the share value directly takes the exchange rate of `PercentageMath.ONE` (line 231). As this is the first deposit, the current total supply equals the calculated `stakeAmount = underlyingAmount.percentDiv(exchangeRate())= 1WEI`. After that, the actor can further transfer a huge amount of `_stakedToken` with the goal of making the pool token extremely expensive.

An extremely expensive pool token can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to a precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `UniswapV2`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial stake provider, but this cost is expected to be low and acceptable. Another alternative requires a guarded launch to ensure the pool is always initialized properly.

Recommendation Revise current execution logic of `stake()` to defensively calculate the share amount when the pool is being initialized.

Status The issue has been fixed by the following pull request: `120`.

3.15 Trust Issue Of Admin Keys

- ID: PVE-015
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [10]
- CWE subcategory: CWE-287 [4]

Description

In the `Augmented Finance` protocol, there is a privileged account (with the `admin` role) that plays a critical role in governing and regulating the protocol-wide operations (e.g., performing sensitive operations and configuring system parameters). In the following, we show the representative functions potentially affected by the privileged accounts.

```
function setRedeemable(bool redeemable)
  external
  override
  aclHas(AccessFlags.LIQUIDITY_CONTROLLER)
{
  _redeemPaused = !redeemable;
}

function setPaused(bool paused) external override onlyEmergencyAdmin {
  _redeemPaused = paused;
}
```

Listing 3.17: A number of representative `setters` in `SlashableStakeTokenBase`

```
  function slashUnderlying(
    address destination,
    uint256 minAmount,
    uint256 maxAmount
  ) external override aclHas(AccessFlags.LIQUIDITY_CONTROLLER) returns (uint256 amount)
      {
    uint256 balance = _stakedToken.balanceOf(address(this));
    // console.log('balance: ', balance);
    uint256 maxSlashable = balance.percentMul(_maxSlashablePercentage);
    // console.log('max slashable: ', maxSlashable);

    if (maxAmount > maxSlashable) {
      amount = maxSlashable;
    } else {
      amount = maxAmount;
    }
    // console.log('amount: ', amount);
    if (amount < minAmount) {
      return 0;
    }
    // console.log('transferring to destination: ', destination);
    _stakedToken.safeTransfer(destination, amount);

    emit Slashed(msg.sender, destination, amount);
    return amount;
  }
```

Listing 3.18: `SlashableStakeTokenBase::slashUnderlying()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. Note that a compromised `owner` account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team confirmed that the privileged account will be later transferred to the DAO governance contract.

3.16 Flashloan-Lowered StableBorrowRate For Mode-Switching Users

- ID: PVE-016
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `LendingPool`
- Category: Business Logic [13]
- CWE subcategory: CWE-837 [8]

Description

Inherited from `Aave`, the `Augmented Finance` protocol supports both variable and stable borrow rates. The variable borrow rate follows closely the market dynamics and can be changed on each user interaction (either borrow, deposit, withdraw, repayment or liquidation). The stable borrow rate instead will be unaffected by these actions. However, implementing a fixed stable borrow rate model on top of a dynamic reserve pool is complicated and the protocol provides the rate-rebalancing support to work around dynamic changes in market conditions or increased cost of money within the pool.

In the following, we show the code snippet of `swapBorrowRateMode()` which allows users to swap between stable and variable borrow rate modes. It follows the same sequence of convention by firstly validating the inputs (Step I), secondly updating relevant reserve states (Step II), then switching the requested borrow rates (Step III), next calculating the latest interest rates (Step IV), and finally performing external interactions, if any (Section V).

```
307  function swapBorrowRateMode(address asset, uint256 rateMode) external override
       whenNotPaused {
308    DataTypes.ReserveData storage reserve = _reserves[asset];

310    (uint256 stableDebt, uint256 variableDebt) = Helpers.getUserCurrentDebt(msg.sender,
         reserve);

312    DataTypes.InterestRateMode interestRateMode = DataTypes.InterestRateMode(rateMode);

314    ValidationLogic.validateSwapRateMode(
315      reserve,
316      _usersConfig[msg.sender],
317      stableDebt,
318      variableDebt,
319      interestRateMode
320    );

322    reserve.updateState();

324    if (interestRateMode == DataTypes.InterestRateMode.STABLE) {
      IStableDebtToken(reserve.stableDebtTokenAddress).burn(msg.sender, stableDebt);
      IVariableDebtToken(reserve.variableDebtTokenAddress).mint(
        msg.sender,
        msg.sender,
        stableDebt,
        reserve.variableBorrowIndex
      );
    } else {
      IVariableDebtToken(reserve.variableDebtTokenAddress).burn(
        msg.sender,
        variableDebt,
        reserve.variableBorrowIndex
      );
      IStableDebtToken(reserve.stableDebtTokenAddress).mint(
        msg.sender,
        msg.sender,
        variableDebt,
        reserve.currentStableBorrowRate
      );
    }

    reserve.updateInterestRates(asset, reserve.aTokenAddress, 0, 0);

    emit Swap(asset, msg.sender, rateMode);
  }
```

Listing 3.19: LendingPool::swapBorrowRateMode()

Our analysis shows this `swapBorrowRateMode()` routine can be affected by a flashloan-assisted sandwiching attack such that the new stable borrow rate becomes the lowest possible. Note this attack is applicable when the borrow rate is switched from variable to stable rate. Specifically, to perform the attack, a malicious actor can first request a flashloan to deposit into the reserve pool so that the reserve's utilization rate is close to 0, then invoke `swapBorrowRateMode()` to perform the variable-to-borrow rate switch and enjoy the lowest `currentStableBorrowRate` (thanks to the nearly 0 utilization rate in current reserve), and finally withdraw to return the flashloan. A similar approach can also be applied to bypass `maxStableLoanPercent` enforcement in `validateBorrow()`.

Recommendation Revise current execution logic of `swapBorrowRateMode()` to defensively detect sudden changes to a reserve utilization and block malicious attempts.

Status This issue has been mitigated as the team confirms that the stable borrowing will not be available at protocol's launch and this feature is planned for further improvements.

3.17 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-017
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.20: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `transfer()` routine in the `Treasury` contract. If the `USDT` token is supported as `token`, the unsafe version of `IERC20(token).transfer(recipient, amount)` (line 35) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)! Also note that the related safe-version of `approve()` may be better instantiated twice: the first resets the allowance while the second sets the intended amount.

```
22   function approve(
23     address token,
24     address recipient,
25     uint256 amount
26   ) external aclHas(AccessFlags.TREASURY_ADMIN) {
27     IERC20(token).approve(recipient, amount);
28   }
29
30   function transfer(
31     address token,
32     address recipient,
33     uint256 amount
34   ) external aclHas(AccessFlags.TREASURY_ADMIN) {
35     IERC20(token).transfer(recipient, amount);
36   }
```

Listing 3.21: `Treasury::approve()/transfer()`

Note this issue is also applicable to other contracts, including `BaseUniswapAdapter`, `FlashLiquidationAdapter`, and `WETHGateway`.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status The issue has been fixed by this commit: `0e9b1a4`.

3.18 Possible DoS On Forced Destruction Of LendingPool

- ID: PVE-018
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: LendingPool
- Category: Business Logic [13]
- CWE subcategory: CWE-841 [9]

Description

The Augmented Finance protocol's main LendingPool contract takes a proxy-based implementation where the proxy contract is deployed at the front-end while the logic contract contains the actual business logic implementation. Specifically, it takes a delegatecall-based proxy pattern so that each component is split into two contracts: a back-end logic contract (that holds the implementation) and a front-end proxy (that contains the data and uses delegatecall to interact with the logic contract). From the user's perspective, they interact with the proxy while the code is executed on the logic contract. Moreover, to accommodate increased contract code size, the protocol splits the liquidation functionalities into another contracts LendingPoolCollateralManager. Note that both contracts can be queried from the MarketAccessController registry.

```
function initialize(IMarketAccessController provider) public initializer(POOL_REVISION
    ) {
  _addressesProvider = provider;
  _maxStableRateBorrowSizePct = 25 * PercentageMath.PCT;
  _flashLoanPremiumPct = 9 * PercentageMath.BP;
  _maxNumberOfReserves = 128;
}
```

Listing 3.22: LendingPool::initialize()

Our analysis shows that the current implementation may suffer from a denial-of-service (DoS) by forcing the LendingPool to self-destruct. Specifically, a malicious user Malice may call initialize() on the back-end logic contract of LendingPool, not the proxy. With that, the initialize() call successfully bypasses the validation from the modifier initializer(POOL_REVISION) and populates a malicious provider, which can be queries to return a controlled collateralManager. After that, Malice calls liquidationCall() to execute the code from collateralManager in the context of the logic contract of LendingPool. Since the collateralManager contract is controlled, it may simply execute self-destruct to destroy the LendingPool logic, which immediately corrupts the execution of the entire protocol.

Recommendation Ensure the LendingPool::initialize() call cannot be bypassed to thwart the above denial-of-service attack.

Status The issue has been fixed by the following pull request: `115`.

4 Conclusion

In this audit, we have analyzed the `Augmented Finance` design and implementation. The system presents a unique, robust DeFi lending protocol that allows users to freely deposit virtual assets as collateral in order to borrow virtual assets. It is designed with a protocol token `AGF` that can be used starting from governance voting, staking and yield boosting, to more utilities as `Augmented Finance` usage grows. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that `Solidity-based` 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[3] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[4] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[5] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[6] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[7] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[8] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

[10] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[11] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

[14] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[15] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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

[19] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[20] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.