PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Radiate Finance (DLPVault)

14 September 2023

paladinsec.co

info@paladinsec.co

Table of Contents

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocation for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

Paladin retains the right to re-use any and all knowledge and expertise gained during the audit process, including, but not limited to, vulnerabilities, bugs, or new attack vectors. Paladin is therefore allowed and expected to use this knowledge in subsequent audits and to inform any third party, who may or may not be our past or current clients, whose projects have similar vulnerabilities. Paladin is furthermore allowed to claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for Radiate Finance's DLPVault contract on the Arbitrum. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective.

1.1 Summary

Project Name	Radiate Finance
URL	TBC
Platform	Arbitrum
Language	Solidity
Preliminary Contracts	https://github.com/RadiateProtocol/dlp-vaults/tree/da3beeb7ef69fd18689f970c6150aa8e1ecba390/src
Resolution 1	https://github.com/RadiateProtocol/dlp-vaults/commit/fe39b435b2d2902a0ecf3efbcac96954b34cdc98
Resolution 2	https://github.com/RadiateProtocol/dlp-vaults/blob/05ca021b6d0c72ea9e89bb2c333cefeadc2bc80b/src/policies
Resolution 3	https://github.com/Radiate-Protocol/radiate-contacts/blob/update/audit_09_01/src/policies/DLPVault_Audit.sol

1.2 Contracts Assessed

Name	Contract	Live Code Match
`DLPVault`		

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	6	2	1	3
Medium	4	2	-	2
Low	13	7	-	6
Informational	2	1	-	1
Total	**25**	**12**	**1**	**12**

Classification of Issues

Severity	Description
High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
Medium	Bugs or issues with that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 DLPVault

ID	Severity	Summary	Status
01	HIGH	Malicious user can relock expired positions	✓ RESOLVED
02	HIGH	Kernel contract can prevent withdrawals	ACKNOWLEDGED
03	HIGH	Compromised admin can lock withdrawals	ACKNOWLEDGED
04	HIGH	Compromised admin can enable delegation to malicious leverager	✓ RESOLVED
05	HIGH	Compromised admin can steal tokens	ACKNOWLEDGED
06	HIGH	Malicious user can artificially cause thousands of queued withdrawals	PARTIAL
07	MEDIUM	Compound will return early for `totalSupply = 0`	✓ RESOLVED
08	MEDIUM	Malicious user can lock the vault through different paths	ACKNOWLEDGED
09	MEDIUM	`MFD.getAllRewards` might revert	✓ RESOLVED
10	MEDIUM	`_stakeTokens` might revert	ACKNOWLEDGED
11	LOW	AToken withdrawal might revert	✓ RESOLVED
12	LOW	Use of deprecated `safeApprove` method	✓ RESOLVED
13	LOW	The treasury will be charged a fee going back to itself	✓ RESOLVED
14	LOW	The swap threshold does not consider the reward token value	✓ RESOLVED
15	LOW	`executeOperation` might revert	ACKNOWLEDGED
16	LOW	Lack of `receiver` validation	✓ RESOLVED
17	LOW	Malicious admin can abuse token swaps	ACKNOWLEDGED
18	LOW	Admin has full control over reward tokens	ACKNOWLEDGED
19	LOW	Architectural risk: Large withdrawals might be block the queue for a long time	ACKNOWLEDGED
20	LOW	Inheritance of non-upgradeable contracts	ACKNOWLEDGED
21	LOW	Obtained Ether can never be withdrawn	✓ RESOLVED
22	LOW	Compounding can be sandwich-attacked	ACKNOWLEDGED
23	LOW	Checks-effects-interactions pattern is not adhered to	✓ RESOLVED
24	INFO	Users can artificially increase the overall queue sum	ACKNOWLEDGED
25	INFO	Unused import	✓ RESOLVED

2 Findings

2.1 DLPVault

`DLPVault` is a highly customized vault contract which allows users to stake their DLP tokens into Radiant's `MultiFeeDistributor` contract and receive a corresponding ERC20 receipt for their stake. These tokens are then locked for a specific time, which is determined by the contract owner and can be set to 1 month, 3 months, 6 months and 12 months, depending on the setting of `typeIndex`. After the locking period has passed, the unlocked positions will be withdrawn automatically and either relocked or paid out, depending on the active queue.

`DLPVault` will therefore accumulate a basket of reward tokens for the time the vault has a valid stake in the `MultiFeeDistributor` contract. These tokens are then claimed and swapped to WETH during each time the vault gets compounded. The received WETH will then be added as liquidity to the DLP token and restaked within the MFD contract.

Users will therefore have a compounding effect, which constantly increases the underlying value of their corresponding ERC20 shares. Users can queue a withdrawal at any time, which then gets executed as soon as the vault has sufficient funds. Due to the locking mechanism within the MFD contract, this might take some time until this payout can happen, depending on the existing queue and the remaining time for position unlocks.

It is important to note that as soon as users queue a withdrawal, they will not receive any additional rewards after this queue since the share redemption is executed at the time of the queue, and the withdrawal queue mechanism is based on the first-come-first-serve principle.

`DLPVault` therefore represents locked DLP tokens which can be used to receive additional radiant tokens for a lending / borrowing position in Radiant's main protocol. For this purpose, the Leverager contract will create a position on behalf of the `DLPVault` and then collects the accumulated Radiant token through the `DLPVault`.

Since the minimum DLP amount in order to receive Radiant token emissions is 5% of the total deposited value, the contract exposes a `boostDLP` function which allows the admin to stake additional DLP tokens on behalf of the `DLPVault` to increase the total locked value of DLP tokens.

**The scope of this audit does not include the `RolesConsumer` contract since the client mentioned that this has already been audited. Additionally, the audit wa conducted using the legitimate DLP token with the ERC20 implementation.*

This is the second audit that Paladin has done on this codebase. Since the first codebase had a large amount of high severity issues, the client accepted our recommendation of a full re-audit. However, since this codebase also includes a large amount of high severity issues, of which some are not easily fixable, we do not see this codebase as production ready and cannot advise our client to launch this codebase after a simple resolution round. To strengthen the security in the whole web3 landscape, we highly recommend that the fixed codebase undergo an unbiased audit by a third party, which has not already conducted two audits on this codebase.

2.1.1 Privileged Functions

- `setFee`
- `setDefaultLockIndex`
- `addRewardBaseTokens`
- `removeRewardBaseTokens`
- `setVaultCap`
- `enableCreditDelegation`
- `disableCreditDelegation`
- `withdrawTokens`
- `boostDLP`
- `unboostDLP`

2.1.2 Issues & Recommendations

Issue #01	Malicious user can relock expired positions
Severity	HIGH SEVERITY
Description	Due to the publicly callable `withdrawExpiredLocksForWithOptions` function, a user can simply relock positions as soon as they reach their unlock time, effectively locking funds within the MFD contract permanently.
Recommendation	Consider adding a function that allows `autoRelockDisabled` to be set to `true`.
Resolution	RESOLVED The auto relock is now disabled within the constructor.

Issue #02	Kernel contract can prevent withdrawals
Severity	HIGH SEVERITY

Description	The treasury address is fetched from the Kernel contract as follows:
	`function configureDependencies()`
	`external`
	`returns (Keycode[] memory dependencies)`
	`{`
	`dependencies = new Keycode[](2);`
	`dependencies[0] = toKeycode("ROLES");`
	`dependencies[1] = toKeycode("TRSRY");`
	`ROLES = ROLESv1(address(kernel.getModuleForKeycode(dependencies[0])));`
	`treasury = address(kernel.getModuleForKeycode(dependencies[1]));`
	`}`
	Moreover, the Kernel contract can be changed by itself:
	`function changeKernel(Kernel _kernel) external onlyKernel {`
	`kernel = _kernel;`
	`emit KernelChanged(address(_kernel));`
	`}`
	This could result in a state where treasury is being set to address(0), effectively preventing withdrawals due to revert in the fee transfer.

Recommendation	Consider implementing a simple safeguard which ensures that `treasury` can never be `address(0)`.
Resolution	ACKNOWLEDGED The client indicated that the `Kernel` contract will be configured properly so this case can never happen.

Issue #03	Compromised admin can lock withdrawals
Severity	HIGH SEVERITY
Description	There are multiple ways for a compromised admin to DoS the contract: A) The admin can add a non-compatible `RewardInfo` or too many RewardInfo's, resulting in a revert within the compound function. Additionally, if a non-compatible ERC20 token is being added, this might revert the compound function under certain circumstances like 0 value transfers. B) A compromised admin address can intentionally lock withdrawals for a certain time: a) Setting the default lock index to 3 (1 year) b) Depositing a large amount of DLP tokens via `boostDLP` c) Immediately calling `unboostDLP` with that large amount -> This will block project the queue progress until the said amount has been accumulated via: a) rewards b) withdrawal of unlocked positions Depending on the size of the withdrawal request, this can take a long time for the contract to accumulate the desired tokens, potentially as long as the initial position is locked, in the worst scenario. Whenever a reward token is removed, the accrued rewards from the last period will also be lost. *This issue is different from a user causing that state since no fee is applied on the `boostDLP` and `unboostDLP` calls.
Recommendation	Consider using a multi-signature contract with KYC-ed participants as admin.
Resolution	ACKNOWLEDGED The client stated that the admin will be a multi-signature wallet and a trusted entity.

Issue #04	Compromised admin can enable delegation to malicious leverager
Severity	HIGH SEVERITY

Description	`enableCreditDelegation` allows the admin to enable any address to borrow funds on behalf of the `DLPVault`, which means that the `DLPVault` will receive the debt.
	
	`/**`
	`  * @dev delegates borrowing power to a user on the specific debt token`
	`  * @param delegatee the address receiving the delegated borrowing power`
	`  * @param amount the maximum amount being delegated. Delegation will still`
	`  * respect the liquidation constraints (even if delegated, a delegatee cannot`
	`  * force a delegator HF to go below 1)`
	`**/`
	`function approveDelegation(address delegatee, uint256 amount) external override {`
	`  _borrowAllowances[_msgSender()][delegatee] = amount;`
	`  emit BorrowAllowanceDelegated(_msgSender(), delegatee, _getUnderlyingAssetAddress(), amount);`
	`}`
	`function mint(`
	`  address user,`
	`  address onBehalfOf,`
	`  uint256 amount,`
	`  uint256 rate`
	`) external override onlyLendingPool returns (bool) {`
	`  MintLocalVars memory vars;`
	
	`  if (user != onBehalfOf) {`
	`  _decreaseBorrowAllowance(onBehalfOf, user, amount);`
	`}`
	
	If this privilege is granted to a malicious address, user funds which are staked via the Leverager contract are at risk — the size of funds at risk depends on the `borrowRatio` since the health factor must be still considered as healthy after a borrow transaction.

Recommendation	Consider disallowing an arbitrary `_leverager` address to be passed as a parameter.

Resolution	RESOLVED A sufficient check was implemented.

Issue #05	**Compromised admin can steal tokens**
Severity	HIGH SEVERITY
Description	The admin can withdraw tokens via the `withdrawTokens` function. However, this is limited to the non-claimable balance within the contract. Under normal business logic, the token flow is as follow: a) Withdraw the expired locks b) Process the queue and increase claimableDLP c) Relock the remaining DLP which were not queued If in a) a large amount is withdrawn, but in b) this amount is not sufficient to process the queue, in c) only the `balance - queuedAmount` is being relocked. This will result in a state where the contract accumulates tokens until the balance is large enough to process the next index in the queue. However, a compromised admin address can potentially steal tokens during this accumulation phase. Moreover, the admin can simply call `withdrawExpiredLocksWithOptions` on the MFD contract, withdrawing locks without processing the queue in the same transaction, this would even allow the admin to steal tokens if the amount would be decreased by `queuedDLP` instead of `claimedDLP`. *This also applies to any `AToken` in the contract, the admin can therefore transfer out any `AToken`, including the collateral token, as long as the HF is still valid after the transfer.
Recommendation	Consider using a multi-signature contract with KYC-ed participants as admin.

Resolution	ACKNOWLEDGED The client stated that the admin will be a multi-signature wallet and a trusted entity.

Issue #06	**Malicious users can artificially cause thousands of queued withdrawals**
Severity	HIGH SEVERITY
Description	A malicious user can call the `redeem` function multiple times without any limitation while always passing 1 wei as share amount. This will then artificially increase the `withdrawalQueues` array, potentially pushing thousands of withdrawals into the queue. A user can therefore partially DoS the contract (the `compound` function would potentially need to be called hundreds of times to clear the queue, since it is limited with 30 queue clearances per call.)
Recommendation	A partial fix would be to limit the amount of queues which can be created per address, however, this will not prevent Sybil attacks. Another option would be to implement a *nominal* fee on each withdrawal on top of the dynamic fee and is paid in the native gas token to disincentivize users from abusing this logic.
Resolution	PARTIALLY RESOLVED Within each withdraw or redeem call, a check was implemented to ensure that a user cannot have more than 5 outstanding withdrawals in the queue. Within the withdrawal, `queueIndex` is pushed into a user corresponsive enum — this enum length is then used for the safety check. During the claim, the index is then forced to be removed from the enum, which will revert in the case it has already been claimed.

Issue #07	Compound will return early for `totalSupply = 0`
Severity	MEDIUM SEVERITY

Description	The compound function is necessary to process the withdrawal queue. Since it's possible for all users to have initiated their withdrawal and therefore no shares are left, the contract is in a state where it has to process the queue to honor all withdrawals.
	
	However, since `_processWithdrawalQueue` is internal, there is no way to process the queue because the compound function returns early:
	
	`function compound() public {`
	`  if (totalSupply() == 0) return;`
	
	`  // reward balance before`
	`  uint256 length = rewards.length;`
	`  uint256[] memory balanceBefore = new uint256[](length);`
	
	`  [...]`
	
	`  // process withdrawal queue`
	`  _processWithdrawalQueue();`
	
	`  // stake`
	`  _stakeDLP();`
	`}`
	
	*This issue was only rated as medium since a simple deposit will then increase the supply again, therefore, no assets will be locked in the contract.

Recommendation	Consider making the `_processWithdrawalQueue` function public.
Resolution	RESOLVED The function is now public.

Issue #08	Malicious user can lock the vault through different paths
Severity	MEDIUM SEVERITY

Description	Users can stake as many DLP tokens as they desire via the `deposit` function. For each `deposit` call, a new lock position within the MFD contract is created, assuming the deposit amount is larger than `minDLPBalance`.
	
	MFD code:
	`_insertLock(`
	`onBehalfOf,`
	`LockedBalance({`
	`amount: amount,`
	`unlockTime: block.timestamp.add(lockPeriod[typeIndex]),`
	`multiplier: rewardMultipliers[typeIndex],`
	`duration: lockPeriod[typeIndex]`
	`})`
	`);`
	
	A user can create multiple locks within the same block, resulting in `unlockTime` being exact same timestamp. To further understand the exploit mechanism, it is necessary to gain an understanding of how unlocked positions can be withdrawn — this is possible via the internal `_withdrawExpiredLocksFor` function which is triggered through various paths.
	
	However, the `DLPVault` has only two options to do so:
	
	a) Within the `compound` function
	
	b) Within the `stake` function
	
	Unfortunately, both paths only trigger `_withdrawExpiredLocksFor` with `limit=0` which forces a loop over **all** locks:

```
LockedBalance[] storage locks = userLocks[user];

if (locks.length != 0) {
    uint256 length = locks.length <= limit ? locks.length :
limit;
    for (uint256 i = 0; i < length; ) {
        if (locks[i].unlockTime <= block.timestamp) {
            lockAmount = lockAmount.add(locks[i].amount);
            lockAmountWithMultiplier =
lockAmountWithMultiplier.add(locks[i].amount.mul(locks[i].mu
ltiplier));
            locks[i] = locks[locks.length - 1];
            locks.pop();
            length = length.sub(1);
        } else {
            i = i + 1;
        }
    }
```

The clue here is that this does not force a large gas consumption as long as these positions have not exceeded their `unlockTime`.

However, since the attacker created these locks all in the same block, the contract will be sooner or later be in a state where this loop attempts to iterate over all created lock positions by this user, which will then ultimately fail due to too large of a gas consumption. This will then effectively lock the vault completely.

A simpler exploit would simply be calling `MFD.stake` with the vault as `onBehalfOf` and the shortest `typeIndex`. A user can therefore create an arbitrary amount of locks on behalf of the `DLPVault`, effectively locking the contract.

Fortunately, Radiant implemented a solution for this issue — `withdrawExpiredLocksForWithOptions` implements a limit parameter, which allows for paginated withdrawals.

This issue was downgraded to medium after it was determined that anyone can publicly call `withdrawExpiredLocksForWithOptions` to paginate unlock positions, any user or representative of the Radiate team can therefore *manually* solve the locked state.

Recommendation Consider implementing Radiant's safety mechanism, allowing the admin to withdraw unlocked positions partially.

Resolution	ACKNOWLEDGED For such scenarios, the Radiate team will manually unlock the positions if the contract is ever in such a state.

Issue #09	**`MFD.getAllRewards` might revert**
Severity	MEDIUM SEVERITY
Description	The MFD contract has two methods of claiming reward tokens: a) Calling `getAllRewards` which loops over all reward tokens and claims the corresponding share. b) Calling `getReward` with `rewardTokens` as input parameter, where the caller can decide which tokens will be withdrawn. If the Radiant team ever decides to add more reward tokens to the array, the first call will revert at some point when the function call exceeds Arbitrum's block size limit. In that scenario, the vault would be locked permanently since only scenario a) was implemented.
Recommendation	Consider changing the `getAllRewards` call to `getReward` and passing the desired reward tokens as input parameters, respectively just the array which is defined within the DLPVault contract as rewards.
Resolution	RESOLVED

Issue #10	`_stakeTokens` might revert
Severity	MEDIUM SEVERITY

Description	`_stakeTokens` checks if `amount` is larger than the return value of `bountyManager`. However, the MFD contract supports a state in which the `BountyManager` is not set and is therefore `address(0)`:
	
	`function _stake(uint256 amount, address onBehalfOf, uint256 typeIndex, bool isRelock) internal whenNotPaused {`
	`  if (amount == 0) return;`
	`  if (bountyManager != address(0)) {`
	`  require(amount >= IBountyManager(bountyManager).minDLPBalance(), "min stake amt not met");`
	`  }`
	`  require(typeIndex < lockPeriod.length, "invalid index");`
	
	In this state, the MFD contract works as expected but `DLPVault` can never stake any tokens since the return value is always expected to be a number, which results in a reverted transaction.
	
	This issue was rated only as Medium severity since the `bountyManager` cannot be set to `address(0)` after it has been set once:
	
	`function setBountyManager(address _bounty) external onlyOwner {`
	`  require(_bounty != address(0), "bounty is 0 address");`
	`  bountyManager = _bounty;`
	`  minters[_bounty] = true;`
	`}`
	
	However, theoretically, this issue could happen for future deployments / forked projects.

Recommendation	Consider implementing the same check as highlighted within the MFD code and only checking for the minimum balance whenever the `bountyManager` is set.

Resolution	ACKNOWLEDGED The Radiate team decided to not make any changes since the underlying MFD contract has a set bounty address which can never be set back to `address(0)` again. However, if there is ever a Radiant fork or a new MFD contract and this protocol is built on top of it, the recommendation should be implemented.

Issue #11	**AToken withdrawal might revert**
Severity	LOW SEVERITY
Description	The protocol can receive `ATokens` as a reward token, which then results in an attempt to withdraw these from the Lending protocol to receive the underlying token. However, this could revert under certain circumstances, such as if the lending protocol has no reserve tokens left. This issue was only rated as low severity since the contract owner can remove the blocking asset from the `rewardTokens` array.
Recommendation	Consider implementing a low-level call without forcing success.
Resolution	RESOLVED

Issue #12	Use of deprecated `safeApprove` method
Severity	LOW SEVERITY
Description	The `safeApprove` method of OpenZeppelin's `SafeERC20` library has been deprecated due to a problem with updating a user's allowance from a non-zero value to another non-zero value. Although there were no critical instances of this issue in the audited contracts, it is recommended not to use this method to prevent potential DoS scenarios.
Recommendation	Per OpenZeppelin's recommendation: whenever possible, use `safeIncreaseAllowance` and `safeDecreaseAllowance` instead.
Resolution	RESOLVED This issue was fixed in a third resolution round.

Issue #13	The treasury will be charged a fee going back to itself
Severity	LOW SEVERITY
Description	When users withdraw or redeem their dLP tokens, a withdrawal fee denominated in the vault's shares is charged. This fee is sent to the treasury address using the inherited ERC20 `_transfer` method. When the treasury wants to exchange the accumulated fees from shares to dLP tokens, it has to use the same `withdraw` or `redeem` function as users do. However, this would again apply fees, meaning that the treasury will not be able to redeem all its shares in a single withdrawal if the withdrawal fee is more than 0.
Recommendation	Consider not charging a withdrawal fee if the shares' owner is the treasury address.
Resolution	RESOLVED

Issue #14	The swap threshold does not consider the reward token value
Severity	LOW SEVERITY

Description	To prevent dust swaps, a swap threshold check is implemented to only execute a swap from a reward token to `WETH` if the pending reward tokens are above a given threshold. This threshold is hardcoded as 0.01 of a whole token of each reward token:
	
	`// Threshold`
	`if (`
	`  reward.pending <`
	`  (10 ** (IERC20Metadata(reward.token).decimals() - 2))`
	`) return;`
	
	A potential problem here is that the actual value of the pending reward tokens is not considered. For example, if the reward token is WBTC, 0.01 pending tokens could be $300 while for stablecoins it would be $0.01, which is a 30000x difference.

Recommendation	Consider whether this could become a problem for processing swaps.
Resolution	RESOLVED

Issue #15	`executeOperation` might revert
Severity	LOW SEVERITY
Description	`executeOperation` is called by AAVE during the flashloan callback and is responsible for the unlooping mechanism. It works as follows: a) Repay a specific amount which was computed before within the Leverager contract, this amount is the whole flashloan value. b) Withdraw the flashloan value + premium from the collateral position. c) Withdraw the desired user request, deducted the premium for the flashloan. However, if we take a look at steps a) & b), we quickly realize that it is more withdrawn than repaid, which should work fine under normal circumstances. However, if the LTV is already very high and the premium, which is applied on top, exceeds the LTV threshold, it will result in a revert due to an insufficient health factor. This issue was rated as low severity because the contract would be already in a liquidation state when the LTV reaches the necessary threshold to prevent the withdrawal call.
Recommendation	Consider refactoring the vault to make it possible to liquidate single user positions, this will then prevent this issue as well.
Resolution	ACKNOWLEDGED This issue will be fixed within `Leverager`.

Issue #16	Lack of receiver validation
Severity	LOW SEVERITY
Description	It is possible to pass a `receiver` parameter when queueing a withdrawal. This receiver will then receive the DLP tokens. Moreover, once a withdrawal is scheduled, it is not possible to cancel it. If the `receiver` is `address(0)`, the tokens will be permanently stuck in the contract.
Recommendation	Consider implementing a validation for `withdraw` and `redeem`, which ensures that the receiver is not `address(0)`.
Resolution	RESOLVED

Issue #17	Malicious admin can abuse token swaps
Severity	LOW SEVERITY
Description	The admin can set any UNIV3 pool for token swaps. Therefore, it is possible to create a pool which has not existed previously, using a new fee value. To this pool, the admin can now add liquidity to a bad range, resulting in profiting from the swaps within the compound functionality.
Recommendation	Consider using a multi-signature contract with KYC-ed participants as admin.
Resolution	ACKNOWLEDGED The client stated that the admin will be a multi-signature wallet and a trusted entity.

Issue #18	Admin has full control over reward tokens
Severity	LOW SEVERITY
Description	Via the `addRewardBaseTokens`, and respectively `removeRewardBaseTokens`, the contract owner can add or remove reward tokens. The owner can therefore simply remove all reward tokens, effectively preventing the vault from any yield. Moreover, when removing a reward token, the accumulated yield from the past period is lost.
Recommendation	Consider using a multi-signature contract with KYC-ed participants as admin.
Resolution	ACKNOWLEDGED The client stated that the admin will be a multi-signature wallet and a trusted entity.

Issue #19	**Architectural risk: Large withdrawals might be block the queue for a long time**
Severity	LOW SEVERITY

Description	Whenever a deposit above the minimum threshold of 5 DLP is executed, the excess amount is locked in the MFD contract (deducted the current queued amount).
	
	If a user now deposits a large amount of, for example, 100_000 DLP tokens and attempts to withdraw this large amount at a later time, this withdrawal attempt is being pushed into the `withdrawalQueue`.
	
	If the contract now reaches a state where index to be processed next is exactly the pushed one, all other withdrawals will be prevented from being processed until the contract has sufficient funds to cover this request, as the processing breaks the loop:
	
	`for (`
	`  uint256 i = 0;`
	`  i < MAX_QUEUE_PROCESS_LIMIT && withdrawalQueueIndex < length;`
	
	`  ) {`
	`  WithdrawalQueue memory queue = withdrawalQueues[`
	`  withdrawalQueueIndex`
	`  ];`
	
	`  if (balance < queue.assets) {`
	`  break;`
	`  }`
	
	This effectively makes the contract impractical to use as the queue can only continue once enough funds have been accumulated due to the withdrawal of unlocked positions.

Recommendation	This is an architectural decision taken by the Radiate team, a potential solution would be to not break the loop but continue with the subsequent queues. However, this would then potentially prevent the large withdrawal to never be paid out.
Resolution	ACKNOWLEDGED The Radiate team is aware of the implications of the Fi-Fo methodology for this purpose.

Issue #20	Inheritance of non-upgradeable contracts
Severity	LOW SEVERITY
Description	The contract inherits the `RolesConsumer` contract — this contract does not expose a gap variable and can therefore result in storage collisions whenever variables are being added in this contract.
Recommendation	Consider adding a gap variable in the `RolesConsumer` contract.
Resolution	ACKNOWLEDGED The Radiate team stated that there will be no new variables.

Issue #21	Obtained Ether can never be withdrawn
Severity	LOW SEVERITY
Description	The contract exposes a fallback function to obtain Ether, however, there is no function to withdraw Ether.
Recommendation	Consider removing this functionality.
Resolution	RESOLVED `withdrawTokens` now implements logic to transfer Ether out.

Issue #22	Compounding can be sandwich-attacked
Severity	LOW SEVERITY
Description	Within the `compound` function, the reward token is swapped to WETH using UniswapV3. `amountOutMin` and `sqrtPriceLimitX96` are both set to zero, which allows a slippage of 100%. This can be abused by a malicious third-party. The issue is only rated as low severity since the compound happens several times per day, minimizing the impact to a minimum. A similar issue also applies to the `joinPool` call since the composition will be changed and users could benefit from the increased radiant value.
Recommendation	A simple solution for this issue is to fetch an external oracle, however, since the probability is high that there is no oracle for some reward tokens, this might not work. Moreover, this could result in users DoS'ing the contract if a hardcoded slippage is used.
Resolution	ACKNOWLEDGED

Issue #23	Checks-effects-interactions pattern is not adhered to
Severity	LOW SEVERITY

Location	L360
	`function boostDLP(uint256 _amount) external onlyAdmin {`
	`    DLP.safeTransferFrom(msg.sender, address(this), _amount);`
	
	`    boostedDLP += _amount;`
	
	`    _stakeTokens(_amount);`
	`}`
	
	L402
	`function _sendCompoundFee(uint256 _index, uint256 _harvested) internal {`
	`    if (fee.compoundFee == 0) return;`
	
	`    RewardInfo storage reward = rewards[_index];`
	`    uint256 feeAmount = (_harvested * fee.compoundFee) / MULTIPLIER;`
	
	`    IERC20(reward.token).safeTransfer(treasury, feeAmount);`
	
	`    reward.pending -= feeAmount;`
	`}`

Description	There are multiple sections within the contract where external calls are executed before effects. This can result in reentrancy attacks and violates best practices for smart contract development.
Recommendation	Consider executing the external calls after the effects.
Resolution	RESOLVED

Issue #24	Users can artificially increase the overall queue sum
Severity	INFORMATIONAL
Description	Users can simply deposit and immediately withdraw again to create a large queue request. This can prevent other users from regularly withdrawing. This issue was only rated as informational since the contract implemented a fee for depositing/withdrawing.
Recommendation	Consider always setting a valid fee to prevent such scenarios.
Resolution	ACKNOWLEDGED

Issue #25	Unused import
Severity	INFORMATIONAL
Location	L14 `import {IMultiFeeDistribution, LockedBalance} from "../interfaces/radiant-interfaces/IMultiFeeDistribution.sol";`
Description	The `LockedBalance` import is unused.
Recommendation	Consider removing the unused import.
Resolution	RESOLVED


PALADIN
BLOCKCHAIN SECURITY