PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Camelot

30 October 2022

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 full rights over all intellectual property (including expertise and new attack or exploit vectors) discovered during the audit process. Paladin is therefore allowed and expected to re-use this knowledge in subsequent audits and to inform existing projects that may have similar vulnerabilities. Paladin may, at its discretion, claim bug bounties from third-parties while doing so.

1 Overview

This report has been prepared for Camelot on the Arbitrum network. 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	Camelot
URL	https://app.camelot.exchange
Network	Arbitrum
Language	Solidity

1.2 Contracts Assessed

Name	Contract	Live Code Match
GrailTokenV2	0x3d9907F9a368ad0a51Be60f7Da3b97cf940982D8	✓ MATCH
XGrailToken	0x3CAaE25Ee616f2C8E13C74dA0813402eae3F496b	✓ MATCH
YieldBooster	0xD27c373950E7466C53e5Cd6eE3F70b240dC0B1B1	✓ MATCH
NFTPool	Deployed by NFTPoolFactory	✓ MATCH
NFTPoolFactory	0x6dB1EF0dF42e30acF139A70C1Ed0B7E6c51dBf6d	✓ MATCH
CamelotMaster	0x55401A4F396b3655f66bf6948A1A4DC61Dfc21f4	✓ MATCH
CamelotFactory	0x6EcCab422D763aC031210895C81787E87B43A652	✓ MATCH
CamelotPair	Deployed by CamelotFactory	✓ MATCH
UniswapV2ERC20	Dependency	✓ MATCH
Math	Dependency	✓ MATCH
SafeMath	Dependency	✓ MATCH
UQ112x112	Dependency	✓ MATCH
CamelotRouter	0xc873fEcbd354f5A56E00E710B90EF4201db2448d	✓ MATCH
UniswapV2Library	Dependency	✓ MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
🔴 **High**	5	5	-	-
🟠 **Medium**	7	7	-	-
🟡 **Low**	19	17	1	1
🟣 **Informational**	39	34	2	3
Total	**70**	**63**	**3**	**4**

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 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 GrailTokenV2

ID	Severity	Summary	Status
01	HIGH	`claimMasterRewards` function is fundamentally flawed	✓ RESOLVED
02	MEDIUM	`GrailTokenV2` distributes emissions retroactively if emissions are re-enabled after being set to zero	✓ RESOLVED
03	INFO	Governance risk: `maxSupply` can be changed to an infinitely large number	✓ RESOLVED
04	INFO	Lack of validation	✓ RESOLVED
05	INFO	Typographical errors	✓ RESOLVED

1.3.2 XGrailToken

ID	Severity	Summary	Status
06	HIGH	Redeem finalization is fundamentally flawed, allowing for an exploiter to finalize a redeem multiple times	✓ RESOLVED
07	LOW	`getGrailByVestingDuration()` can revert due to division by zero	✓ RESOLVED
08	INFO	[Frontend] Phishing vulnerability for `convertTo()`	✓ RESOLVED
09	INFO	Excess XGrailToken will get stuck in the contract	✓ RESOLVED
10	INFO	Allocations during redemption are not captured in the `usageAllocations` variable	✓ RESOLVED
11	INFO	`_allocate` and `_deallocate` lack usage whitelisting	✓ RESOLVED
12	INFO	Typographical errors and gas optimizations	✓ RESOLVED

1.3.3 YieldBooster

ID	Severity	Summary	Status
13	MEDIUM	Anyone can boost any other user's NFT without any form of approval, allowing users to DoS the burning of NFTs	✓ RESOLVED
14	LOW	YieldBooster does not validate pools in any way	PARTIAL
15	INFO	`forceDeallocate()` does not unboost the pool position and does not properly function if the YieldBooster somehow gets out-of-sync with the XGrailToken	✓ RESOLVED
16	INFO	Typographical errors	✓ RESOLVED

1.3.4 NFTPool

ID	Severity	Summary	Status
17	HIGH	`mergePositions` allows anyone to steal other users positions and harvests	✓ RESOLVED
18	MEDIUM	Various functions including NFT transfers and `updatePool` are missing reentrancy-guards	✓ RESOLVED
19	MEDIUM	The `mergePositions` function is flawed and may delete the entire positions	✓ RESOLVED
20	MEDIUM	The `destroyPosition` function de-allocates from `msg.sender` instead of whomever allocated the actual boost points	✓ RESOLVED
21	LOW	The transfer functions may have undesired side-effects with boosted tokens	✓ RESOLVED
22	LOW	Harvests break if xGrailRewardsShare is ever set to zero	✓ RESOLVED
23	LOW	`renewLockPosition` and `lockPosition` do not work if the lock is expired	✓ RESOLVED
24	LOW	`_checkOnNFTHarvest` is flawed for `harvestPositionTo`	✓ RESOLVED
25	LOW	Inconsistency: _harvestPosition does not update the boost multiplier if `isUnlocked` is enabled	✓ RESOLVED
26	LOW	The `createPosition` function may create position with 0 amount when using tokens with a fee on transfer	✓ RESOLVED
27	INFO	Typographical errors	✓ RESOLVED
28	INFO	Configurational issue: Parameters in initialize() function can be malicious	PARTIAL
29	INFO	`_requireOnlyOperatorOrOwnerOf` and `_requireOnlyApprovedOrOwnerOf` are doing the same	✓ RESOLVED

30	INFO	Inconsistency: `_safeRewardsTransfer` avoids failure if contract has insufficient but such checks are not present for the xGrail amount that is harvested	✓ RESOLVED
31	INFO	Lack of validation	✓ RESOLVED
32	INFO	`harvestPositionTo` exposes a frontend-phishing vulnerability	✓ RESOLVED
33	INFO	`harvestAllPositions`, `withdrawFromAllPositions` and `mergePositions` can run out of gas	ACKNOWLEDGED
34	INFO	`_destroyPosition` will revert in some edge-cases	✓ RESOLVED
35	INFO	`splitPosition` does not burn the position if its completely emptied	✓ RESOLVED
36	INFO	`mergePositions` is overprotective with the lock duration guards	✓ RESOLVED

1.3.5 NFTPoolFactory

ID	Severity	Summary	Status
37	LOW	`create2` not checked against a zero response	✓ RESOLVED
38	INFO	`_pools` is private	✓ RESOLVED

1.3.6 CamelotMaster

ID	Severity	Summary	Status
39	LOW	`massUpdatePool` only updates the active pools, causing potentially significant rewards to be distributed in hindsight if a pool is ever reactivated	✓ RESOLVED
40	LOW	Configurational risk: YieldBooster	ACKNOWLEDGED
41	LOW	Newly added pools can dilute rewards retroactively	✓ RESOLVED
42	LOW	`getPoolAddressByIndex` and `getActivePoolAddressByIndex` are uncallable due to a faulty guard clause	✓ RESOLVED
43	LOW	`startTime` is not aligned with `startTime` from GrailToken	✓ RESOLVED
44	INFO	Unused definition	✓ RESOLVED
45	INFO	`_grailToken` can be made immutable	✓ RESOLVED
46	INFO	`getPoolInfo` lacks a guard-clause	✓ RESOLVED
47	INFO	Lack of validation	ACKNOWLEDGED
48	INFO	Typographical errors	✓ RESOLVED
49	INFO	Lack of `safeTransfer` usage within `_safeRewardsTransfer`	✓ RESOLVED

1.3.7 CamelotFactory

ID	Severity	Summary	Status
50	MEDIUM	The owner fee can be used to block deposits and withdrawals	✓ RESOLVED
51	INFO	Lack of events in the constructor	✓ RESOLVED
52	INFO	`create2` success is unchecked (also present in Uniswap)	✓ RESOLVED
53	INFO	Typographical errors	✓ RESOLVED
54	INFO	Gas optimization	✓ RESOLVED

1.3.8 CamelotPair

ID	Severity	Summary	Status
55	HIGH	MEV bots can drain one asset of the pair	✓ RESOLVED
56	HIGH	Governance risk: Governance can drain the pairs	✓ RESOLVED
57	MEDIUM	`_k` lacks overflow protection	✓ RESOLVED
58	LOW	`initialize` function lacks an additional safeguard	✓ RESOLVED
59	LOW	Some `private` variables should be made `public`	✓ RESOLVED
60	LOW	Various functions are not guarded against reentrancy	✓ RESOLVED
61	INFO	`factory` can be made immutable	✓ RESOLVED
62	INFO	Gas optimizations	✓ RESOLVED
63	INFO	Typographical errors	✓ RESOLVED
64	INFO	External calls after K check are undesired	✓ RESOLVED

1.3.9 UniswapV2ERC20

ID	Severity	Summary	Status
65	INFO	`permit` can be frontrun to prevent someone from calling `removeLiquidityWithPermit` (also present in Uniswap)	✓ RESOLVED

1.3.10 Math, SafeMath and UQ112x112

No issues found.

1.3.11 CamelotRouter

ID	Severity	Summary	Status
66	LOW	The `quote` function returns erroneous values for the stableswap	✓ RESOLVED
67	LOW	`receive()` lacks a safeguard	✓ RESOLVED
68	INFO	Gas optimizations	✓ RESOLVED
69	INFO	The `addLiquidity` function does not properly support tokens with a fee on transfer (also present in Uniswap)	ACKNOWLEDGED
70	INFO	Phishing Issue: A malicious or hacked frontend could adjust routes, tokens or to parameters to steal tokens when users make swaps (also present in Uniswap)	PARTIAL

1.3.12 UniswapV2Library

No issues found.

2 Findings

2.1 Farm/GrailTokenV2

`GrailTokenV2` is the native token of the Camelot protocol and is used as a reward token within the Masterchef contract. Unlike regular farming tokens, `GrailTokenV2` handles the emission rate and emission distribution internally.

Once all necessary variables have been initialized, anyone can call `emitAllocations()` which calculates the current emissions and the correct allocation of emissions to various recipients based on the current emission rate. One share is assigned to the Masterchef, and the other share is assigned to the `treasuryAddress`. The treasury share is minted directly to the treasury.

The share for the Masterchef is minted to the `GrailTokenV2` contract itself and can then be distributed to the Masterchef via `claimMasterRewards`, which is callable only by the MasterChef itself.

`GrailTokenV2` uses a `maxSupply` variable which is defined within the constructor and sets an upper limit for the minting of the token. However, the owner of this contract has the ability to change the `maxSupply` with no upper-bound limit.

2.1.1 Token Overview

Address	TBC
Token Supply	TBC
Decimal Places	18
Transfer Max Size	None
Transfer Min Size	None
Transfer Fees	None
Pre-mints	TBC

2.1.2 Privileged Functions

- `claimMasterRewards [ onlyMaster ]`
- `initializeMasterAddress [ callable once ]`
- `initializeEmissionStart [ callable once ]`
- `updateAllocation`
- `updateEmissionRate`
- `updateMaxSupply`
- `updateTreasuryAddress`
- `transferOwnership`
- `renounceOwnership`

2.1.3 Issues & Recommendations

Issue #01	**claimMasterRewards function is fundamentally flawed**
Severity	● HIGH SEVERITY
Description	Currently, the claimMasterRewards function calls emitAllocations which emits the full outstanding allocation to this contract and increments the masterReserve by that amount. However, due to the general function flow, claimMasterRewards is called by CamelotMaster during _updatePool. While claimMasterRewards function claims the full amount, the _updatePool function only requests the amount which is allocated for one specific pool. Unfortunately, claimMasterRewards sets the masterReserve variable to zero, which makes it impossible for other pools to receive their fair share.
Recommendation	Consider simply deducting the effective amount from the masterReserve.
Resolution	✓ RESOLVED The effective amount is now deducted from the masterReserve.

Issue #02	`GrailTokenV2` distributes emissions retroactively if emissions are re-enabled after being set to zero
Severity	● MEDIUM SEVERITY

Location	Line 117-119
	`if (_maxSupply <= circulatingSupply || currentBlockTimestamp <= _lastEmissionTime || _lastEmissionTime == 0 || emissionRate == 0) {`
	`    return;`
	`}`

Description	`GrailTokenV2` allows for distributing emissions retroactively due to an erroneous early return in the emission update function. **Path to vulnerability** 1. Set emissions to zero 2. Wait a year 3. Set emissions to 1 token per second 4. 1 year of tokens will be emitted at 1 token per second This functionality is present because the allocation rate adjusting function will call `emitAllocations` in an attempt to bring the `lastEmissionTime` variable up to date. However, due to the aforementioned early return, this does not always happen. In these cases, the new rate will apply retroactively.
Recommendation	Consider adding a secondary if-statement that does adjust the last emission time. This issue also presents itself when the maximum cap is reached and later incremented. Consider what needs to be done in this case. The maximum cap could also return early upon the re-enabling of the emission rate (1. reach max-cap, 2. re-enable emissions 3. increment max cap, this path would still cause rewards to be enabled in hindsight). Therefore, we recommend moving this check to the updating section as well.

Code Recommendation	`if (currentBlockTimestamp <= _lastEmissionTime || _lastEmissionTime == 0) {`
	`    return;`
	`}`
	
	`if (_maxSupply <= circulatingSupply || emissionRate == 0) {`
	`    lastEmissionTime = currentBlockTimestamp;`
	`    return;`
	`}`

Resolution	✓ RESOLVED The recommended code was implemented.

Issue #03	**Governance risk: `maxSupply` can be changed to an infinitely large number**
Severity	● INFORMATIONAL
Description	The contract conveys the idea that the token has a fixed `maxSupply`. However, since the owner can change the `maxSupply` without any upper-bound limit, this is not the case.
Recommendation	Consider communicating this probability openly to the users or renaming the variable to a more transparent name.
Resolution	✓ RESOLVED A hard cap of 200,000 tokens has been added.

Issue #04	Lack of validation
Severity	● INFORMATIONAL
Description	During the contract creation, the `treasuryAddress_` parameter lacks a non-zero validation. Additionally, `initialSupply` should be validated to be smaller than `maxSupply_` in order to not mint more than the maximum supply.
Recommendation	Consider adding a non-zero validation within the constructor to align with the `updateTreasuryAddress` function.
Resolution	✓ RESOLVED

Issue #05	Typographical errors
Severity	● INFORMATIONAL

Description	We have consolidated the typographical errors into a single issue to keep the report brief and readable.
	
	Line 52
	`event EmitAllocations(uint256 masterShare, uint256 treasuryShare);`
	
	Starting an event name with `Emit` is rather redundant.
	
	Line 65
	`* @dev Throws error if called by any account other than the master or router`
	
	There is no functionality for the router within this contract.
	
	Line 95
	`return uint256(100).sub(masterAllocation());`
	
	100 is considered a magic value here: Consider using the more appropriate `ALLOCATION_PRECISION`.
	
	`masterEmissionRate` can be made `external`.

Recommendation	Consider fixing the typographical errors.
Resolution	✓ RESOLVED

2.2 Farm/XGrailToken

XGrailToken is a specialized token contract with several use-cases for its users.

Users can convert their Grail tokens to xGrail tokens at a 1:1 ratio at any time using the convert function. xGrail as no maximum supply (other than Grail's maximum supply). Whitelisted addresses have the privilege to transfer the xGrail tokens freely. For regular users, transfers of the xGrail token is limited to whitelisted addresses.

Once a user has converted their GrailToken to the XGrailToken, the user can then do two things with the XGrailToken:

1. The user can call the redeem() function to redeem xGrail back to Grail. This creates a vesting position for the user with a user-configured vesting time. If the user chooses to vest back over a longer duration, more Grail tokens are received back. Although the duration and ratio can be adjusted by the contract governance, a minimum duration of 15 days and a maximum duration of 90 days is configured initially. At 15 days, 50% is redeemed while at 90 days, 100% would be redeemed. The remainder is burned and any vesting duration between these two extremes would redeem a linear amount between 50% and 100%.

   During the whole vesting time, a portion of the vesting amount will be allocated to a governance configured dividendAddress to receive dividends. At the start, the percentage is configured at 50%. After the vesting time is over, finalization of the vest removes the allocation and pays back the beforehand calculated grailAmount which will never be above the original xGrail amount. The difference between the calculated GrailAmount and XGrailAmount then simply gets burned as the corresponding GrailToken.

2. The user can call the allocate() function with a specific usageAddress. This function then also withdraws the amount of XGrailToken which was provided

by the user from the user and allocates these `xGrail` tokens to the provided `usageAddress`. Unlike the previously mentioned function, this allows the user to provide a variable `usageAddress`. Once the user decides to remove their allocation, they can simply call `deallocate()` which removes the user's allocation on the `usageContract` and sends back the user's `xGrail` after a fee (configurable up to 2% per usage) has been deducted. The fee amount immediately gets burned in the contract as `Grail` tokens. This scenario can also get called from the `usageContract` directly with the `userAddress` as the first parameter. However, before the `usageContract` or the user can call this function, the user must approve the `userAddress` with the desired amount via `approveUsage`.

2.2.1 Privileged Functions

- `updateRedeemSettings`
- `updateDividendAddress`
- `updateDeallocationFee`
- `updateTransferWhitelist`
- `transferOwnership`
- `renounceOwnership`

2.2.2 Issues & Recommendations

Issue #06	**Redeem finalization is fundamentally flawed, allowing for an exploiter to finalize a redeem multiple times**
Severity	HIGH SEVERITY

Description	The finalization of redeems is fundamentally flawed. Users can request a redemption and finalize it after a certain delay. However, if multiple people request a redemption at the same time, the finalization will not actually delete the correct pending redemptions. Instead, it always deletes the most recent one, regardless of who finalized their redemption.
	
	This means that an exploiter can redeem the same pending redemption potentially many times.
	
	`RedeemInfo storage _redeem = userRedeems[msg.sender][redeemIndex];`
	
	`[...]`
	
	`_redeem = userRedeems[msg.sender][userRedeems[msg.sender].length - 1];`
	`userRedeems[msg.sender].pop();`
	
	The reason for this vulnerability being possible is because the developers incorrectly assumed that they can override the `redeemIndex` by re-assigning to the variable.
	
	However, when you re-assign a storage pointer, you simply point the Solidity pointer to a new location. Nothing is actually written to storage.

Recommendation	Consider writing to storage instead: `userRedeems[msg.sender][redeemIndex] = userRedeems[msg.sender][userRedeems[msg.sender].length - 1];` This should be done in all locations where this flow is utilized.
Resolution	✓ RESOLVED The recommended code snippet has been implemented in an internal function which is now always called.

Issue #07	`getGrailByVestingDuration()` can revert due to division by zero
Severity	● LOW SEVERITY
Description	Currently, the ratio is calculated as follows: `uint256 ratio = minRedeemRatio.add((duration.sub(minRedeemDuration)).mul(maxRedeemRatio.sub(minRedeemRatio)) .div(maxRedeemDuration.sub(minRedeemDuration))` If `maxRedeemDuration` and `minRedeemDuration` are equal, this results in a function revert due to a division by zero.
Recommendation	Consider keeping this edge-case in mind, we will also write out another recommendation to validate the `updateRedeemSettings` accordingly.
Resolution	✓ RESOLVED These two values are no longer required to be equal.

Issue #08	[Frontend] Phishing vulnerability for `convertTo()`
Severity	● INFORMATIONAL
Description	`convertTo()` exposes a to parameter which decides who will receive the minted `XGrailToken`. If the frontend is ever compromised, this can lead to stolen funds where the attacker sets the to parameter to his own wallet.
Recommendation	Our recommendation is to acknowledge this issue and keep the best security standards for the website. Alternatively, if this function is supposed to be used exclusively by contracts, the code could validate that the `msg.sender` is a contract to effectively lock out normal users.
Resolution	✓ RESOLVED The `convertTo` function is now exclusively callable by contracts, effectively negating any phishing concerns as users can no longer call it. We commend the client for this pragmatic approach.

Issue #09	Excess XGrailToken will get stuck in the contract
Severity	INFORMATIONAL
Description	The `_deallocate` function deducts a `deallocationFeeAmount` from the withdrawable `XGrailToken` balance: `uint256 deallocationFeeAmount = amount.mul(usagesDeallocationFee[usageAddress]).div(10000);` This `deallocationFeeAmount` then gets burned as the corresponding GrailToken, but the amount of XGrailToken will get stuck in the contract.
Recommendation	Consider also burning the excess amount of XGrailToken in the contract.
Resolution	✓ RESOLVED This amount is now burned as well.

Issue #10	Allocations during redemption are not captured in the `usageAllocations` variable
Severity	INFORMATIONAL
Description	The contract allows for a part of redemptions to get allocated to a dividends address. However, this part is not added to the `usageAllocations` accounting variable that keeps track of the total xGrail allocated to an address.
Recommendation	Consider whether this is desired. If so, consider incrementing and decrementing the accounting variable in all relevant functions (`redeem`, `finalizeRedeem`, `updateRedeemDividendsAddress` and `cancelRedeem`).
Resolution	✓ RESOLVED The client has indicated that this behavior is desired. No change was made.

Issue #11	`_allocate` and `_deallocate` lack usage whitelisting
Severity	INFORMATIONAL
Description	Presently the allocate and deallocate functions allow for users to allocate and deallocate their xGrail to any contract of their choosing. This might be considered a risk because it gives an exploiter more attack surface than is strictly necessary. For example, an exploiter might try abusing the the system by setting usageAddress to xGrail itself. Within the allocate function, the following line of code would then be called on xGrail. `function allocate(address usageAddress, uint256 amount, bytes calldata usageData) external nonReentrant {` Obviously, this is not supposed to be callable from xGrail itself, therefore a small privilege escalation would occur (this in fact would revert due to the `nonReentrant` guard). We did not find any direct exploits from this privilege escalation or from providing a malicious `usageAddress`, however, it is never a good idea to give exploiters a larger attack surface than is strictly necessary.
Recommendation	Consider adding a whitelist of valid usage addresses.
Resolution	✓ RESOLVED No changes have been made as the client hopes that third parties will start developing usages as well, and this should in fact be a permissionless process. We do remind the users that the attack surface vector is therefore still a little bit larger than we would want it to be, however, no attack methodology has been identified.

Issue #12	Typographical errors and gas optimizations
Severity	● INFORMATIONAL

Description	We have consolidated the typographical errors and the sections which can be further optimized for gas usage below.
	
	Line 40
	`IGrailTokenV2 public grailToken;`
	
	`grailToken` can be made immutable.
	
	Line 475
	`usageApprovals[userAddress][usageAddress] = approvedXGrail.sub(amount, "allocate: non authorized amount");`
	
	This error state seems unreachable — consider removing the error message as it wastes gas (it is always allocated in memory).
	
	The team can also consider whether it would make sense to also make `approveUsage` `nonReentrant`. This way, all external functions are nicely marked with `nonReentrant`.
	
	Finally, it might make sense to consider not requiring users to approve an allocation address in case they are going to allocate themselves. This is similar to how the ERC20 transfer function does not require approval, only `transferFrom`.
	
	Line 488
	`* @dev Allocates "amount" of available xGRAIL to "usageAddress" contract`
	
	The `_deallocate` comment should mention *deallocates* instead of allocates.

Recommendation	Consider fixing the typographical errors.
Resolution	✓ RESOLVED

2.3 Farm/YieldBooster

`YieldBooster` is responsible for the `XGrailToken` allocation. As previously mentioned in the `XGrailToken` description, each user has the ability to allocate their `XGrail` tokens to a specific `usageAddress` within the `XGrailToken` contract.

The `YieldBooster` contract represents this `usageContract` and allows users to boost their positions in the `NFTPool` contracts and keeps track of a user's total allocation, a pool contract's total allocation and a user's specific allocation for a pool contract and the specific `NFTId`.

2.3.1 Privileged Functions

- `updateForcedDeallocationStatus`
- `emergencyWithdraw`
- `transferOwnership`
- `renounceOwnership`

2.3.2 Issues & Recommendations

Issue #13	**Anyone can boost any other user's NFT without any form of approval, allowing users to DoS the burning of NFTs**
Severity	● MEDIUM SEVERITY
Description	`YieldBooster` does not validate that a user is the rightful owner, operator or approved wallet of an NFT they are boosting. This might go against the general practice within `NFTPool` where only `_requireOnlyOperatorOrOwnerOf` wallets can add to positions. This functionality seems to be especially problematic since `_destroyPosition` unallocates the full `position.boostPoint` from a single caller!
Recommendation	Consider whether this is an issue, and if so, consider validating that the user address adheres to `_requireOnlyOperatorOrOwnerOf` (it could be provided as a parameter to boost for example).
Resolution	✓ RESOLVED The client has indicated that this is in fact desired behavior. However, they did not realize that `_destroyPosition` often breaks in this case. The client has fixed this by only unstaking the position amount which was allocated by the actual NFT owner. The other stakers will need to unallocate themselves.

Issue #14	`YieldBooster` does not validate pools in any way
Severity	● LOW SEVERITY
Description	`YieldBooster` does not validate any pool addresses. This means that a malicious user, the "exploiter", will attempt to find ways to abuse the contract by sending a malicious pool address to it. Paladin did not find any vector (such as reentrancy) to abuse this contract through a malicious contract but is a proponent of strictly limiting the freedom which is given to exploiters. In this case, this freedom can be and should be trivially reduced.
Recommendation	Consider adding a `_validatePool()` function or similar which calls the `NFTPoolFactory` and confirms that the address was deployed by the pool factory. This should also be done with `allocate`, `deallocate` and `deallocateFromPool`. We strongly urge the team to implement this recommendation even though we could not find any direct exploit methods for this.
Resolution	● PARTIALLY RESOLVED The client has indicated they desire these pools to not be validated in case they want to adjust in the future. No changes have been made. Users should understand that the attack surface vector for exploiters remains rather large, even though we did not find a way to exploit this.

Issue #15	**`forceDeallocate()` does not unboost the pool position and does not properly function if the `YieldBooster` somehow gets out-of-sync with the `XGrailToken`**
Severity	INFORMATIONAL
Description	Currently the `forceDeallocate()` function only allows the user to receive his remaining share of `XGrailToken` without unwinding the boost position within the assigned NFTPool. However, since we expect this behavior is as desired, this issue is only marked as informational. Secondly, the balance which is unallocated to the `XGrailToken` is presently based on the local `YieldBooster` value, this means that there are certain call-paths where the `YieldBooster` will request to remove too many tokens (if `forceDeallocate` is called twice).
Recommendation	Consider using this function only in case of emergency. The secondary issue can be resolved simply by revoking the current allocation as returned by the `XGrailToken`. This makes the function even more available as it will be guaranteed to always remove the full allocation from the user to the YieldBooster. `uint256 amount = xGrailToken.usageAllocations(msg.sender, address(this));`
Resolution	✓ RESOLVED The recommendation has been implemented.

Issue #16	Typographical errors
Severity	INFORMATIONAL

Description	We have consolidated the typographical errors into a single issue to keep the report brief and readable.
	
	Line 46
	`event EmergencyWithdraw(address caller, address token, uint256 amount);`
	
	`token` can be provided as `IERC20` to avoid casting the value later on.
	
	Lines 70, 80
	`* returns multiplier * 1e2`
	
	These rates seem to be denominated in basis points (1e4) for now.

Recommendation	Consider fixing the typographical errors.
Resolution	✓ RESOLVED

2.4 Farm/NFTPool

`NFTPool` is a unique staking contract which mints an NFT to the staker for each staking position. Users then receive Grail and xGrail rewards for staking their tokens for a certain lockup duration. The amount of rewards the user receives increases with both the boost allocation the user has given to the NFT and the duration the user has locked it for. `spNFT` represents the position, its amount and its lock duration. The owner of it has full control over the position.

Each `NFTPool` has its own staking token and its own `NFT`. The `NFTPool` is deployed through the `NFTFactory` contract with the following parameters, of which only the first can be provided as a parameter:

- `lpToken`
- `MasterChef`
- `GrailToken`
- `XGrailToken`

Each `NFTPool` serves as a pool within `CamelotMaster` and needs to be added with the correct `allocationPoints` in order to receive rewards.

Each staking position can get boosted by up to a configurable 250% via two mechanisms:

- Using the boost point mechanism via `xGrail` allocation within the `XGrailToken` contract to achieve an additional `boostMultiplier`. This allows users to temporarily assign their `xGrail` tokens to the NFT to further boost it.

- Locking up the staking position for a predetermined duration to achieve a high `lockMultiplier`. The longer the duration a user commits to, the higher their multiplier (the multiplier increases linearly with the committed duration).

Both multipliers, the `lockMultiplier` and the `boostMultiplier`, can individually become as high as 150%, but only up to 250% combined. This is because the contract sets individual caps and an aggregated cap.

Besides the usual features like topping up a position and withdrawing from a position, this contract has some additional features which allows their users to merge their staking position NFTs into one NFT and split one staking position in two positions as well as relock their positions to achieve a higher `lockMultiplier`.

It should be noted that withdrawals are only available once the lock duration expires, as is expected with a lock. However, users are free to sell their position NFT over-the-counter at a discount. This means that the positions remain semi-liquid, which is both interesting and innovative at the same time.

The well-known `EmergencyWithdraw` function can only be called if a lock has expired, the caller is a privileged address or the contract activated the `emergencyUnlock` feature.

2.4.1 Privileged Functions

- `setLockMultiplierSettings`
- `setBoostMultiplierSettings`
- `setXGrailRewardsShare`
- `setUnlockOperator`
- `setEmergencyUnlock`
- `setOperator`

2.4.2 Issues & Recommendations

Issue #17	**mergePositions allows anyone to steal other users positions and harvests**
Severity	● HIGH SEVERITY

Description	The `mergePositions` function has a fundamental flaw in it which allows a user to steal other users' staking positions and merge it with their own staking position as well as steal other users harvest: it does not check if the caller is the owner of other `tokenIds` than the first one provided in the array.
	
	`for (uint256 i = 1; i < length; ++i) {`
	` uint256 tokenId = tokenIds[i];`
	`_requireOnlyOwnerOf(dstTokenId);`
	
	This should be `_requireOnlyOwnerOf(tokenId);`
	
	This means that a malicious user can add an array of `tokenIds` where he only needs to be the owner of the first `tokenId` in the array.
	
	A little blessing in disguise is the `_destroyPosition` call: If a staking position is boosted, the call will fail due to an underflow within the `_deallocate` function in the `YieldBooster` contract.
	
	However, if the malicious user only chooses non-boosted staking positions, the exploit will still work.

Recommendation	Consider validating the owner correctly.
Resolution	✓ RESOLVED The check has been updated to check the actual `tokenId`.

Issue #18	**Various functions including NFT transfers and `updatePool` are missing reentrancy-guards**
Severity	● MEDIUM SEVERITY
Description	The contract is presently extremely conservative by adding reentrancy guards to all unprivileged functions. This is defensive, but good and desired given that the contract is complex and contains various code locations which allow for reentrancy to occur. However, the developer forgot to add such guards to a couple of places: - The various NFT transfer functions - The `updatePool` function This means that whenever a reentrancy opportunity presents itself, the attacker can use it to transfer the NFT, potentially misleading the system. This has interesting consequences since the codebase caches the `nftOwner` in various locations, while this owner could have changed through reentrancy by the time it is used. We failed to create a valid proof-of-concept within the current codebase but one could conceptualize cases where a position is listed on an NFT and a reentrancy vector is used to 1. accept a bid, 2. transfer the NFT back to the old owner.
Recommendation	Consider adding reentrancy guards to all transfer functions and the `updatePool` function.
Resolution	✓ RESOLVED Reentrancy guards have been added.

Issue #19	**The `mergePositions` function is flawed and may delete the entire positions**
Severity	● MEDIUM SEVERITY
Description	Currently, `mergePositions` loops over all given `tokenIds` after `dstTokenId`. But it is possible to use duplicate `tokenIds` as parameters, which will simply revert for most cases because the position was already deleted and the NFT burned. However, if the last `tokenId` is the same as the first one, it will merge the two tokens together and delete it, resulting in a serious loss for users as all its positions would have been merged together and then at the end, deleting the entire position merged.
Recommendation	Consider adding a proper validation. The easiest way would be to use sorted token ids, and verifying that `lastTokenId < tokenIds[i]` or to merge all positions to a new NFT as it would revert on duplicated NFT id.
Resolution	✓ RESOLVED The client has only addressed the case where the first token is duplicated, which should be sufficient as other cases already revert: `require(tokenId != dstTokenId, "invalid token id");`

Issue #20	**The `destroyPosition` function de-allocates from `msg.sender` instead of whomever allocated the actual boost points**
Severity	● MEDIUM SEVERITY

Location	`Line 400`
	`IYieldBooster(yieldBooster()).deallocateFromPool(msg.sender, tokenId, boostPointsToDeallocate);`

Description	Withdrawals of NFTs can occur by various users and not just the owner of the NFT. However, if the NFT withdrawal withdraws the full amount, `_destroyPosition` is called. Yet, this function always tries to deallocate the boost points from `msg.sender`, who may not be whoever allocated points to the NFT and may definitely not be the NFT owner (eg. an approved wallet).
Recommendation	Consider what should happen with boosts if the token is transferred.
Resolution	✓ RESOLVED Similar logic to the `YieldBooster` has been implemented where all boost points of the `msg.sender` are deallocated and all other users who allocated to the NFT, including potentially the owner, must deallocate themselves.

Issue #21	The transfer functions may have undesired side-effects with boosted tokens
Severity	● LOW SEVERITY
Description	Presently, individuals boost `tokenIds`. However, when a token is transferred, it is unclear what should happen with this boost and various sections of the code may start reverting.
Recommendation	Consider what should happen with boosts if the token is transferred.
Resolution	✓ RESOLVED The client has clarified this: *We want users to be able to transfer a boosted position without unboosting it. A user can still get back his allocation once the position has been transferred.* *Given the further clarifications on how multiple people should be permitted to boost a single `tokenId`, we are comfortable with this behavior.*

Issue #22	**Harvests break if `xGrailRewardsShare` is ever set to zero**
Severity	● LOW SEVERITY
Description	The xGrail token does not permit conversions of zero tokens. However, `_harvest` tries to blindly convert the provided amount. Since the governance can configure the percentage of a harvest that is converted to xGrail, this means that they can never set the percentage to zero. if the conversion percentage is ever set to zero, the `convertTo` call will fail.
Recommendation	Consider wrapping the `convertTo` call in a non-zero check, or consider not failing on zero within the `convertTo` function.
Resolution	✓ RESOLVED `convertTo` is no longer called if the amount is zero.

Issue #23	**`renewLockPosition` and `lockPosition` do not work if the lock is expired**
Severity	● LOW SEVERITY
Description	These two functions are both calling the `internal` function `_lockPosition`. However, if the lock has expired, the math in the functions will fail due to an underflow.
Recommendation	Consider adding logic that handles the expired case.
Resolution	✓ RESOLVED Specific logic has been added.

Issue #24	`_checkOnNFTHarvest` is flawed for `harvestPositionTo`
Severity	● LOW SEVERITY
Description	`_checkOnNFTHarvest` currently checks if the owner of the NFT is able to handle harvests. However, the `harvestPositionTo` function exposes the to address as receiver.
Recommendation	Consider thinking about if this is desired, if not consider changing the logic to fit for `harvestPositionTo`.
Resolution	✓ RESOLVED The client has indicated that this is desired.

Issue #25	Inconsistency: `_harvestPosition` does not update the boost multiplier if `isUnlocked` is enabled
Severity	● LOW SEVERITY
Description	`_harvestPosition` does not update the boost multiplier if `isUnlocked` is enabled. However, throughout the codebase, when the `lockDuration` is adjusted, such an update occurs.
Recommendation	Consider whether this is desired, it might be desired to retain the original boost in this case as the user might not have been okay with this unlock. Alternatively, consider updating the boost multiplier similarly to how its done in the rest of the codebase.
Resolution	✓ RESOLVED This has been updated.

Issue #26	The `createPosition` function may create position with 0 amount when using tokens with a fee on transfer
Severity	● LOW SEVERITY
Location	L440 `amount = _transferSupportingFeeOnTransfer(_lpToken, msg.sender, amount);`
Description	`createPosition` currently checks that the amount parameter is greater than zero, but fails to check it after the `_transferSupportingFeeOnTransfer` is called. This may create positions with 0 amount.
Recommendation	Consider moving the amount check after the token has been transferred. This check should also be added to `addToPosition` in our opinion.
Resolution	✓ RESOLVED The recommendation has been implemented.

Issue #27	Typographical errors
Severity	● INFORMATIONAL

Description	We have consolidated the typographical errors into a single issue to keep the report brief and readable.
	
	Line 46
	`address public operator; // Used to delegate some pool features to project's owners`
	
	The operator is just a privileged address for the `setLockMultiplierSettings` function.
	
	Line 86
	`_grailToken.approve(address(_xGrailToken), uint256(-1));`
	
	We tend to prefer `type(uint256).max` instead of `uint256(-1)` to denote the maximum integer value. This is simply because the former properly communicates the fact that it's a maximum while the latter is an underflow trick.
	
	Line 153
	`* @dev Check if a userAddress is owner of a spNFT`
	
	This comment mentions a `userAddress` variable, however, the function simply uses `msg.sender` instead. It should be noted that this function presently does not check whether the NFT exists either, which we believe is implicit behavior which should be avoided.
	
	Line 209
	`* @dev Returns true if emergency unlocks are activated on this pool or on the master`
	
	This returns the latest minted NFT index. It should also be noted that as long as no NFTs are minted, both the function name and this description are inaccurate, renaming it to `getTokensMinted` might be more accurate.

Line 262

```
* @dev Returns expected multiplier for a "lockDuration"
duration lock (result is *1e2)
```

The result is in basis points (1e4) on our version.

Line 390

```
* @dev Set emergency unlock status for all pools
```

`emergencyUnlock` is only for all staking positions, not for all pools.

Line 558

```
* Can only be called by spNFT's owner
```

The `harvestPositionTo` function can be called by the owner but also by an approved address.

Line 941

```
* @dev If "to" is a contract, confirm whether it's able to
handle rewards harvesting
```

This should say "If `nftOwner` is a contract".

Line 953

```
* @dev If "to" is a contract, confirm whether it's able to
handle addToPosition
```

This should say "If `nftOwner` is a contract".

Line 964

```
* @dev If "to" is a contract, confirm whether it's able to
handle withdrawals
```

This should say "If `nftOwner` is a contract".

Recommendation	Consider fixing the above errors.
Resolution	✓ RESOLVED

Issue #28	Configurational issue: Parameters in `initialize()` function can be malicious
Severity	● INFORMATIONAL
Description	Within the `initialize` function, `_grailToken` and `_xGrailToken` can be freely set. If the deployer decides to set `_grailToken` as the same token as the staking token and `_xGrailToken` to a malicious contract which exposes a `transferFrom` feature, the deployer can steal all deposited staking tokens. Furthermore, it is important that all NFTPools solely get deployed through the correct `NFTPoolFactory`, otherwise it would be possible to call the `initialize` function twice and steal all tokens even if the configuration was correct in the first point.
Recommendation	Since the team is well established and this issue can be spotted upfront, our recommendation is to simply acknowledge this issue and pay attention to it during the contract configuration. A further safeguard could be an initialize parameter to ensure the initialize function can not be called twice.
Resolution	● PARTIALLY RESOLVED The client will be careful with setting the configuration values and has added the guard to prevent re-initialization.

Issue #29	**`_requireOnlyOperatorOrOwnerOf` and `_requireOnlyApprovedOrOwnerOf` are doing the same**
Severity	● INFORMATIONAL
Description	Both functions are checking for the same condition, this increases the contract size unnecessarily.
Recommendation	Consider removing one of these functions.
Resolution	✓ RESOLVED The client has indicated that there's a slight difference in approval flows with both of these contracts within the OpenZeppelin implementation and wants certain functions to be protected by one and others by another. No changes have been made.

Issue #30	**Inconsistency: `_safeRewardsTransfer` avoids failure if contract has insufficient but such checks are not present for the `xGrail` amount that is harvested**
Severity	● INFORMATIONAL
Description	The client has added a safety check to the Grail amount that is sent to the user on harvests: if the contract balance is insufficient, the contract balance is sent instead. This avoids failure on insufficient balances. This check is however not present on the `convertTo` call which means that harvests will still fail if the balance is insufficient.
Recommendation	Consider adding a `_safeConvertTo` function similar to `_safeRewardsTransfer`.
Resolution	✓ RESOLVED A `_safeConvertTo` function has been added.

Issue #31	Lack of validation
Severity	● INFORMATIONAL
Description	Currently, the variable `_maxLockDuration` lacks a proper upper validation. If this variable becomes too high, users would not be able to receive a proper `lockMultiplier`. `_requireOnlyOwnerOf` presently does not validate whether the NFT exists. `createPosition` presently does not validate that the `lockDuration` does not exceed the maximum either. If users provide a greater value, it currently does not seem to benefit them.
Recommendation	Consider setting a reasonable upper limit for `_maxLockDuration`.
Resolution	✓ RESOLVED The client has added the first check while keeping `createPosition` unchecked, the latter has been done to allow derivative contracts to lock for a longer duration. It is therefore desired.

Issue #32	`harvestPositionTo` exposes a frontend-phishing vulnerability
Severity	● INFORMATIONAL
Description	All functions that handle the transfer of funds to a variable to parameter are potential hacking vectors for phishing-attacks.
Recommendation	Consider focusing on best-practices and security mechanisms for the frontend.
Resolution	✓ RESOLVED The NFT owner must now be a smart contract which mitigates all phishing risk.

Issue #33	**`harvestAllPositions`, `withdrawFromAllPositions` and `mergePositions` can run out of gas**
Severity	● INFORMATIONAL
Description	The `harvestAllPositions` functions can potentially run out of gas if the user has too many staking positions. Since it is unlikely that a single user has so many staking positions that this function runs out of gas, we consider simply acknowledging this issue. In the rare case of this happening, the user can then simply use the `harvestPosition` function. The same applies for the `withdrawFromAllPositions` function. The `mergePositions` function will also run out of gas if the user adds too many `tokenIds`.
Recommendation	Consider recommending any affected user to use the alternative functions. Consider setting an upper limit for `tokenIds` within the `mergePositions` function.
Resolution	● ACKNOWLEDGED

Issue #34	`_destroyPosition` **will revert in some edge-cases**
Severity	● INFORMATIONAL
Description	Currently, the `_destroyPosition` function will revert under the following circumstances: 1. `stakingPosition` A was created and boosted by Bob 2. Bob transferred `stakingPosition` A to Alice 3. Alice tries to `mergePositions` including `stakingPosition` A, or Alice tries to withdraw her whole position. 4. Triggering `_destroyPosition` with `boostPoints` will result in an underflow within the `_deallocate` function in the `YieldBooster` contract.
Recommendation	Consider either acknowledging this issue and simply recommend calling the `emergencyWithdraw` function or adding logic that supports this behavior.
Resolution	✓ RESOLVED Only the tokens that are allocated by the sender will be unallocated automatically now.

Issue #35	`splitPosition` **does not burn the position if its completely emptied**
Severity	● INFORMATIONAL
Description	Currently, `splitPosition` does not burn the origin position if it is emptied. This will result in the user needing to call `emergencyWithdraw` to be able to burn the position properly.
Recommendation	Consider implementing logic for this case or simply not allowing the position to get completely emptied.
Resolution	✓ RESOLVED The splitAmount must now be strictly less than the `position.amount`, which ensures that the final two tokens both have value.

Issue #36	`mergePositions` is overprotective with the lock duration guards
Severity	● INFORMATIONAL
Description	Currently, the `mergePositions` function requires lock durations of the merged NFTs to be strictly extended. However, if these NFTs are nearly expired, they should theoretically only need to be extended by the remaining lock amount.
Recommendation	Consider whether it makes sense to require the unlock duration to strictly increase instead. We can also happily resolve this issue on the note that this is desired behavior or that the current behavior is sufficient and simpler.
Resolution	✓ RESOLVED The client would like to retain this logic on the following note: *The behavior is desired, as it avoids some edge cases while keeping it relatively simple.* We agree that it is indeed simple and elegant, and are happy to resolve the issue on this note. No changes have been made.

2.5 Farm/NFTPoolFactory

This contract is a factory for NFT pools. It can deploy new NFT pools for an underlying ERC20 `lpToken` with the `createPool` function. These pools are initialized with the configured `master`, `grailToken`, `xGrailToken` and `lpToken`.

Anyone can call the `createPool` function, however, it can only be called once per `lpToken` which means that each `lpToken` will have at most a single `NFTPool` deployed by the factory.

Pools are deployed using deterministic deployment, which means that theoretically contracts and off-chain tooling can know the addresses of pools beforehand and without explicitly querying the `NFTPoolFactory` instance.

2.5.1 Issues & Recommendations

Issue #37	`create2` not checked against a zero response
Severity	● LOW SEVERITY
Description	The `create2` opcode returns the zero address on failure (revert, already deployed, etc.). Theoretically this could have side-effects within the factory if this happens during a `create2` call within the `createPool` call.
Recommendation	Consider checking explicitly that the returned address is not `address(0)`. On more recent versions of Solidity, `create2` deployments can be made in a safe way using new `Contract{salt: salt}()`.
Resolution	✓ RESOLVED

Issue #38	`_pools` is private
Severity	● INFORMATIONAL
Description	Important variables that third-parties might want to inspect should be marked as public so that these third-parties can easily inspect them through the explorer, web3 and derivative contracts.
Recommendation	Consider marking the variable as public.
Resolution	✓ RESOLVED

2.6 Farm/CamelotMaster

CamelotMaster represents the pivotal point of the farm branch. The contract aggregates all valid NFTPools and is responsible for the correct reward distribution.

CamelotMaster also sets the YieldBooster contract which is responsible for calculating the boostPoint multiplier within each NFTPool, as well as the emergencyUnlock variable which is globally used within all NFTPools and allows withdrawals before the unlockTime has passed.

During the _updatePool of an NFTPool, the function claimRewards within CamelotMaster is called which then updates the specific NFTPool and claims the allocated rewards from the GrailTokenV2 via claimMasterRewards and then transfers the reward amount to the NFTPool.

2.6.1 Privileged Functions

- setYieldBooster
- setEmergencyUnlock
- add
- set
- transferOwnership
- renounceOwnership
- claimRewards [only pools added]

2.6.2 Issues & Recommendations

Issue #39	**`massUpdatePool` only updates the active pools, causing potentially significant rewards to be distributed in hindsight if a pool is ever reactivated**
Severity	● LOW SEVERITY
Description	Since the `massUpdatePools` function does only update the active pools, this will lead to an issue in the following edge-case: 1. Set pool X´s allocation to zero 2. Use the contract as usual 3. Set pool X’s allocation to 100 with withUpdate = true 4. This will then update all pools besides pool X 5. Pool X will distribute rewards in hindsight

Recommendation	Consider simply allow `massUpdatePools` to update all pools, or fixing it by changing the lines 282 to 286 to:
	`if (withUpdate) {`
	`      _massUpdatePools();`
	`      if (allocPoint > 0 && !`
	`_activePools.contains(poolAddress)) {`
	`      _updatePool(poolAddress);`
	`      }`
	`} else {`
	`      _updatePool(poolAddress);`
	`}`
	One could arguably opt for simplicity over efficiency here and move `_updatePool` outside of the if-else clause and remove the else branch completely. The slight gas inefficiency here probably does not outweigh the complexity a more complex section like the one we recommend above adds.

Resolution	✓ RESOLVED The client has opted for the simple and elegant solution of always updating the pool after the mass update occurred.

Issue #40	Configurational risk: `YieldBooster`
Severity	● LOW SEVERITY
Description	The owner has the privilege to change the `YieldBooster` to any address. If the `YieldBooster` is changed to a bad contract, this will break various functions within the NFTPool contract.
Recommendation	Consider acknowledging this issue and consider being careful with changing the `YieldBooster`. It might make more sense to make the `YieldBooster` upgradeable instead.
Resolution	● ACKNOWLEDGED

Issue #41	Newly added pools can dilute rewards retroactively
Severity	● LOW SEVERITY
Description	If a pool is newly added without `withUpdate` this will increase the `totalAllocPoint`, diluting other pools rewards retroactively. The same issue applies to the set function if the `totalAllocPoint` changes too much (this applies in both directions).
Recommendation	Consider always using `withUpdate` when adding new pools.
Resolution	✓ RESOLVED The client has indicated that they will use this consistently. No changes were made.

Issue #42	`getPoolAddressByIndex` and `getActivePoolAddressByIndex` are uncallable due to a faulty guard clause
Severity	● LOW SEVERITY

Description	Both functions have a flawed index sanity check:
	
	`if (index < _pools.length()) return address(0);`
	`if (index < _activePools.length()) return address(0);`
	
	These functions will always return the zero address.

Recommendation	Consider doing the check as follows: `if (index >= _activePools.length()) return address(0);`
Resolution	✓ RESOLVED

Issue #43	`startTime` is not aligned with `startTime` from `GrailToken`
Severity	● LOW SEVERITY
Description	Currently, the `startTime` lacks a validation to ensure it is >= `startTime` within `GrailToken`. If `CamelotMaster` has its `startTime` before the `startTime` within `GrailToken`, all pools that have been added receive less rewards than expected during the first harvest.
Recommendation	Consider validating the `startTime` parameter to be greater than the `startTime` of the `GrailToken`.
Resolution	✓ RESOLVED

Issue #44	Unused definition
Severity	INFORMATIONAL
Location	Line 23 `using SafeERC20 for IERC20;`
Description	SafeERC20 is not used within the contract, however, we recommend using `safeTransfer` instead of `transfer`.
Recommendation	Consider using `safeTransfer` or removing the unused definition.
Resolution	✓ RESOLVED This is now used.

Issue #45	`_grailToken` can be made immutable
Severity	INFORMATIONAL
Description	Variables that are only set in the constructor but never modified can be indicated as such with the immutable keyword. This is considered best practice since it makes the code more accessible for third-party reviewers and saves gas.
Recommendation	Consider marking the variable explicitly as immutable.
Resolution	✓ RESOLVED

Issue #46	`getPoolInfo` lacks a guard-clause
Severity	INFORMATIONAL
Description	Currently, `getPoolInfo` lacks a check that a `poolAddress` exists. In the case of a non-existent pool, this function will return incorrect values.
Recommendation	Consider adding a `validatePool` modifier to this function.
Resolution	✓ RESOLVED The client has indicated that they desire this function to return zero values in this instance.

Issue #47	Lack of validation
Severity	INFORMATIONAL

Location	`Line 196`
	`function add(INFTPool nftPool, uint256 allocPoint, bool withUpdate) external onlyOwner`

Description	The contract lacks validation within the above function. If the pool size becomes too large, `massUpdatePools` might run out of gas.
Recommendation	Consider either adding an explicit limit to the number of active `nftPools` or simply being careful when adding more pools.
Resolution	ACKNOWLEDGED

Issue #48	Typographical errors
Severity	● INFORMATIONAL

Description	We have consolidated the typographical errors into a single issue to keep the report brief and readable.
	
	`Line 344`
	`// claim expected rewards from master`
	It should say "claim expected rewards from the token".
	
	`Line 73`
	`event SetGrailToken(address grailToken);`
	This event is not used and should be removed or used in the constructor.
	
	`Line 182`
	`PoolInfo storage pool = _poolInfo[poolAddress_];`
	The pool should be `memory` to assert that this function will not affect storage.
	
	`Line 238`
	`require(!_pools.contains(poolAddress), "add: pool already exists");`
	The `validatePool` modifier should be used here instead of redefining the same requirement with a `require` statement.
	
	`Lines 341-342`
	`uint256 rewards = currentBlockTimestamp`
	`.sub(lastRewardTime) // nbSeconds`
	`      .mul(_grailToken.masterEmissionRate())`
	`      .mul(allocPoint)`
	`      .div(totalAllocPoint);`
	The `emissionRate` function should be used instead of redefining it here.

Recommendation	Consider fixing the typographical errors.

Resolution	✓ RESOLVED Most of these errors were fixed.

Issue #49	**Lack of `safeTransfer` usage within `_safeRewardsTransfer`**
Severity	● INFORMATIONAL
Description	In the `safeRewardsTransfer` function, the `transfer` method is used to transfer tokens. This will not work for tokens that return `false` on `transfer` (or malformed tokens that do not have a return value). Additionally, the `require` statement can then be removed.
Recommendation	Consider using `safeTransfer` instead of `transfer`.
Resolution	✓ RESOLVED

2.7 Core/CamelotFactory

The main responsibility of `CamelotFactory` is deploying the LP pair contracts. It also stores all necessary variables for the fee structure and privileged functions within the pair. The various privileged factory wallets can therefore adjust pair parameters like the fee amount, fee receiver and whether a pair is a stable pair.

2.7.1 Privileged Functions

- `setOwner`
- `setFeeAmountOwner`
- `setSetStableOwner`
- `setFeeTo`
- `setOwnerFeeShare`
- `setRefererFeeShare`

2.7.2 Issues & Recommendations

Issue #50	The owner fee can be used to block deposits and withdrawals
Severity	● MEDIUM SEVERITY
Location	Line 15 `uint public constant OWNER_FEE_SHARE_MAX = 100000;`
Description	`ownerFeeShare` is validated to be smaller or equal to 100%. However in the `_mintFee` function of the `CamelotPair` contract, line 134 would revert if `ownerFeeShare > 50_000` with a division by zero: `uint d = (FEE_DENOMINATOR / ownerFeeShare).sub(1);`
Recommendation	Consider ensuring that `ownerFeeShare <= 50_000` or add decimals and make sure that it cannot revert with a division by zero.
Resolution	✓ RESOLVED

Issue #51	Lack of events in the constructor
Severity	● INFORMATIONAL
Description	Functions that affect the status of sensitive variables should emit events as notifications, this should also be the case in the constructor.
Recommendation	Add events in the constructor.
Resolution	✓ RESOLVED A non-zero requirement has been added.

Issue #52	`create2` success is unchecked (also present in Uniswap)
Severity	● INFORMATIONAL
Description	If `create2` fails for any reason, it will return `address(0)` – a correct `create2` implementation should always require the return address to be non-zero. As there appears to presently be no vector to make `create2` fail, this is raised as an informational issue. However, this could theoretically change in a hard-fork. It should be noted that we expect the initialize to fail in most cases where no contract exists at the pair address.
Recommendation	Consider validating that the created address is non-zero.
Resolution	✓ RESOLVED

Issue #53	Typographical errors
Severity	● INFORMATIONAL

Description	We have consolidated the typographical errors into a single issue to keep the report brief and readable.
	
	Line 7
	`bytes32 public constant INIT_CODE_PAIR_HASH = keccak256(abi.encodePacked(type(CamelotPair).creationCode));`
	
	The `INIT_CODE_PAIR_HASH` is unused and should be removed.
	
	Line 25
	`event PairCreated(address indexed token0, address indexed token1, address pair, uint);`
	
	The last parameter can be length.
	
	Line 93
	`* @dev Updates the share of fees attributed to the owner (FeeManager)`
	
	There is no `FeeManager`.
	
	Line 110
	`function setRefererFeeShare(address referrer, uint referrerFeeShare)`
	
	The function should be renamed `setReferrerFeeShare`.

Recommendation	Consider fixing the typographical errors.
Resolution	✓ RESOLVED

Issue #54	Gas optimization
Severity	● INFORMATIONAL
Description	Currently the functions `setOwnerFeeShare` and `setRefererFeeShare` are storing the old variable into memory to be able to emit an appropriate event. However, as already done in the earlier functions, the event can simply emitted before the variable is changed: `emit OwnerFeeShareUpdated(ownerFeeShare, newOwnerFeeShare);` Though we generally like to emit events after the update, it makes sense to adjust these two functions to remain consistent with the methodology used (and save some gas).
Recommendation	Consider implementing the above suggestion.
Resolution	✓ RESOLVED

2.8 Core/CamelotPair

`CamelotPair` is involved in storing the asset pairs in a contract for use by the router to add and remove liquidity in equally-valued proportions and for swapping assets. It is a fork of the Uniswap version of this contract but extends it with the following features.

First of all, the pair has a configurational swap fee, which can be configured differently for the two pair tokens up to 2% each. This fee is split between:

a. the `referrerFee`, which is globally set in the factory, individually for each referrer and up to 20%.

b. the `ownerFeeShare` (only if `stableSwap` is `true`), which is set in the factory and can be up to 100% of the remaining fees after referral fee was taken.

c. the `mintFee` (only if `stableSwap` is false), which is calculated based on the `ownerFeeShare` within the `mintFee` function — this has the exact same purpose of `ownerFeeShare` but saves gas by not sending out the protocol swap fee percentage on each swap.

d. the remaining portion of the swap fee simply increases the LP pair value, as is common within Uniswap V2.

Due to the arbitrary fee amount and the `stableSwap` feature, there are a lot different fee results. Most notably for a `stableSwap` pair with maximized fees and a referrer, the pair will send out 1,6% to `feeTo` and 0.4% to the referrer leaving no fee in the pair for value accumulation. These fees will always be taken from the `inputToken`.

Additionally, `CamelotPair` implements a `stableSwap` feature, which can be turned on and off by the `setStableOwner` which is defined within the factory, a wallet managed by the Camelot team. All pairs can be deployed by users but they will always start off as simple Uniswap V2 pairs. It is up to the Camelot governance to decide whether they should become stable pairs. Once converted to a stable pair, the curve of the pair is adjusted to a x3y+y3x curve as is famous from Solidly. If the `stableSwap` feature is turned on, the governance swap fee logic is adjusted as the traditional gas efficient manner of collecting swap fees is not possible on this new curve. The pair will therefore not mint any liquidity tokens to the owner but instead send a small fee amount during each swap to the `feeTo` address.

As already implemented in Camelot V1, `sync()` can only be called if there is a valid ratio. This prevents an annoying exploit some owners have experienced where a malicious party sends one of the tokens to the pair before liquidity is added and calls `sync()`, thereby preventing the owner from adding liquidity through the normal user interface (which in fact crashes in this scenario due to a division by zero).

Another point to mention is that the Camelot factory owner's wallet can withdraw all tokens that do not belong to the pair, which does not do any harm.

It should be noted for anyone reviewing this pair that the stableswap curve is identical to the one used within Solidly. To validate the curve itself, third party reviewers could therefore compare the `_k` function with the one used within Solidly, though we do advise caution with such comparison as the actual swap function is quite different from Solidly.

2.8.1 Privileged Functions

- `setFeeAmount`
- `setStableSwap`

2.8.2 Issues & Recommendations

Issue #55	**MEV bots can drain one asset of the pair**
Severity	● HIGH SEVERITY
Description	When a stable pair is changed to xy=k pair, an MEV bot can drain the entire pair using flashbots (and a flashloan). PoC: Let's say there is pair with 1M USDC and 1M USD 1. The owner queries a transaction to set the pair to stable 2. The transaction is detected in the mempool and a MEV bot detects it and front runs it 3. The bot flashloans (from another protocol) and swaps 9M USDC to USDT, he will receive 998K USDT 4. The transaction to set the pair to a xy=k market is executed 5. The bot swaps the 998K USDT he received to USDC and receive 9M98 USDC 6. USDC is almost drained, 1M USDT and 20K USDC and pair exploited for half its TVL The more USDC the MEV bot swaps, the more USDC it will receive and the less USDC it will have at the end.
Recommendation	Consider not allowing a stable pair to be set to a xy=k curve again. A safety check could be added by checking that the reserves are in the same range (let's say +- 1%) of the parameters to change the curve of a pool. Finally, one could simply add reserve inputs to the function to simply require the function to be called from a helper contract which does more thorough rebalancing and safety checks. Consider adding an immutable pair feature which, when set by your team, prevents you from ever changing the pair type again on specific pairs. Pairs with this feature set will assuage investors as they know all vectors related to switching the pair type no longer apply.

Resolution	✓ RESOLVED The client has added strict validation to the function as was recommended. When swapping over to a different pair type, the pair must not have changed their reserves at all. This is perfect and completely prevents MEV bots from sandwiching the curve change. It will still be the careful responsibility of the team to make sure those reserves are balanced when they swap over, as unbalanced reserves still allow for back running. Secondly, to assuage investors, the team has added the recommended immutability feature and will be marking all core pairs as `immutable` once they are configured. We recommend investors to check that their specific pair is flagged as `immutable` to be sure that the pair is fully decentralized.

Issue #56	Governance risk: Governance can drain the pairs
Severity	● HIGH SEVERITY
Description	Currently, when switching a pair to the stable swap curve, `kLast` is being calculated using the `stableSwap` method. However, when switching back from `stableSwap` to `!stableSwap`, `kLast` is not being reset to zero which allows governance to drain the pair. PoC: 1. A pair is created as usual, with `stableSwap` turned off. 2. Basic operations are done with the pair to ensure `kLast != 0`. 3. Now `stableSwap` is activated, which decreases `kLast` significantly. 4. More basic operations are done (this step is not necessary), during this time, `kLast` does not change but the reserves increase. 5. `stableSwap` is being turned off which will trigger `_mintFee` in various functions 6. Trigger `_mintFee` by executing any liquidity event: `_mintFee` will then calculate an absurd amount to be minted to `feeTo` due to the heavy delta between `reserve0*reserve1` and `kLast`. 7. `feeTo` can simply break the pair which it received and then drain it. However, there is another way for the governance to drain the pair by changing an imbalanced pair, like BTC/USDC, to a stable pair as it would be very profitable to swap USDC to BTC. PoC: 1. Let's say there is pair with 2M USDC and 100 BTC, so BTC is $20K 2. Governance sets the BTC/USDC pair to a stable pair 3. Governance swaps 500K USDC and receive 49.8 BTC that are worth 996K 4. Governance stole 496K in a pool with a TVL of $4M.

	The fact that the governance can arbitrarily change the curve type is therefore a governance risk. The way this is addressed in Curve, a protocol with variable curve gradients, is by slowly adjusting the gradient over time, instead of all at once. This is of course difficult to accomplish with the current design.
Recommendation	Consider fixing the above issues: To fix the first issue, `kLast` should be set to zero when `stableSwap` is being turned off. This will ensure that `_mintFee` can not mint an absurd amount to `feeTo`. To fix the latter, one method could be to set an `_immutable` variable so that when this variable is set to true, no change of the curve can be done anymore. As most users will invest in the incentivized pools, we would strongly recommend to make the incentivized pools immutable, safeguarding most of the TVL against governance attacks and potential configurational issues.
Resolution	✓ RESOLVED The team has added the recommended immutability feature and will be marking all core pairs as immutable once they are configured. We recommend investors to check that their specific pair is flagged as `immutable` to be sure that the pair is fully decentralized. Secondly, the `k` adjustment has been fixed as was recommended.

Issue #57	_k lacks overflow protection
Severity	● MEDIUM SEVERITY
Description	_k lacks overflow protection. Especially in the case of stableswaps, this might cause the pairs for certain stablecoin pairs to misbehave. This would be especially problematic for stablecoins which are pegged to a much smaller denomination, eg. 1 wei (hypothetically). If an overflow were to occur in `_k`, the pair could completely malfunction and allow for unfavorable swaps to occur.
Recommendation	Consider using `SafeMath` consistently throughout the whole pair.
Resolution	✓ RESOLVED This section now uses SafeMath.

Issue #58	**`initialize` function lacks an additional safeguard**
Severity	● LOW SEVERITY
Description	Currently, the `initialize` function is safeguarded with `msg.sender = factory`. This is standard practice and ensures that it cannot be called twice due to the logic in the factory. However, if these contracts are ever forked and the factory is customized or made upgradeable, this will expose the risk of calling `initialize` twice, which might result in a rug.
Recommendation	Consider adding an `initialized = false` requirement and set it to true after the function is executed.
Resolution	✓ RESOLVED The `safeguard` variable has been added.

Issue #59	Some `private` variables should be made `public`
Severity	● LOW SEVERITY

Location	Line 30
	`uint private _decimals0;`
	
	Line 31
	`uint private _decimals1;`

Description	Important variables that third-parties might want to inspect should be marked as public so that these third-parties can easily inspect them through the explorer, web3 and derivative contracts.
Recommendation	Consider making the variables explicitly public.
Resolution	✓ RESOLVED

Issue #60	Various functions are not guarded against reentrancy
Severity	● LOW SEVERITY
Description	Though extremely minor as these functions are governance functions, `setFeeAmount`, `setStableSwap` and `drainWrongToken` all lack a lock modifier. A malicious governance might try to reenter in these functions during a swap to attempt to abuse logic within the swap contract through an ad-hoc variable change. We've not dived into specific vectors or whether it is feasible to attack the pair using privileged reentrancy, but as there is little cost to adding the modifier we see no downside as it will assuage investors.
Recommendation	Consider adding the lock modifier to all unlocked functions (presently only governance functions remain) to assuage investors.
Resolution	✓ RESOLVED All functions are now strictly guarded with reentrancy guards, including the governance functions.

Issue #61	`factory` can be made `immutable`
Severity	● INFORMATIONAL
Description	Variables that are only set in the constructor but never modified can be indicated as such with the immutable keyword. This is considered best practice since it makes the code more accessible for third-party reviewers and saves gas.
Recommendation	Consider making the factory `immutable`. Consider double checking that this still allows for the "partial match" to work in the explorer. We do understand if you prefer to keep it as an address to be consistent with Uniswap.
Resolution	✓ RESOLVED The client has retained this as a storage variable since `immutable` variables are not yet available with this version of Solidity.

Issue #62	Gas optimizations
Severity	INFORMATIONAL

Description	We have consolidated the sections which can be further optimized for gas usage below.
	
	<u>`Lines 97 and L117`</u>
	`97 emit FeeAmountUpdated(token0FeeAmount, token1FeeAmount);`
	`117 emit Sync(reserve0, reserve1);`
	
	The `FeeAmountUpdated` and `Sync` events should use the parameters instead of reloading values from storage to save gas. Consider using the following implementation instead:
	
	`97 emit FeeAmountUpdated(newToken0FeeAmount,`
	`newToken1FeeAmount);`
	
	`117 emit Sync(balance0, balance1);`

Recommendation	Consider implementing the gas optimizations mentioned above.
Resolution	✓ RESOLVED

Issue #63	Typographical errors
Severity	● INFORMATIONAL

Description	We have consolidated the typographical errors into a single issue to keep the report brief and readable.
	
	 <u>Line 6</u>
	 `import './libraries/UQ112x112.sol';`
	
	 <u>Line 13</u>
	 `using UQ112x112 for uint224;`
	
	 `UQ112x112` is not used within the contract therefore these two lines be removed.
	
	 `address public factory;`
	
	 factory can be cached to ICamelotFactory. We do understand if you prefer to keep it as an address for Uniswap compliancy.
	
	 <u>Line 19</u>
	 `address public token0;`
	
	 `token0` can have the type IERC20. We do understand if you prefer to keep it as an address for Uniswap compliancy.
	
	 <u>Line 20</u>
	 `address public token1;`
	
	 `token1` can have the type `IERC20`. We do understand if you prefer to keep it as an `address` to be consistent with Uniswap.
	
	 <u>Line 87</u>
	 `* @dev Updates the swap fees amount`
	
	 This function updates the swap fee percentages, not the raw amounts. A comment below it also indicates the factory owner can call it but only the `feeAmountOwner` of the factory can call it.

Lines 210 and 218

```
TokensData memory tokensData;
```

`tokensData` memory is allocated implicitly. From the compiler's perspective, this memory will temporarily point to a spot in memory which is supposed to be kept at zero. The only way this spot can ever be non-zero is through bad assembly code, which is not present here. However, in line with being extra careful we still recommend you to explicitly initialize the `tokensData` memory immediately by setting it to an explicit value right away:

```
TokensData memory tokensData  = TokensData({
    token0: token0,
    token1: token1,
    amount0Out: amount0Out,
             amount1Out: amount1Out,
    balance0: 0,
    balance1: 0,
})
```

The `skim` function presently lacks an event.

The decimals variables are in fact precision multipliers, not a decimal number.

Recommendation	Consider fixing the above typographical errors.
Resolution	✓ RESOLVED

Issue #64	External calls after K check are undesired
Severity	● INFORMATIONAL
Description	The pair makes several external calls after the K check occurs. This makes formal verification of the pair behavior impossible as theoretically these external calls could cause the invariant to be broken. This is raised as an informational issue as tokens which adjust their balances like this would likely not be suited for the dex anyways.
Recommendation	Consider, if you prefer idiomatic code, to do a secondary invariant increase check before the `_update` function is called, and always using fresh balances (right now they are not always fresh). Such a safeguard would further harden the safety of the pair and permit for more formal verification methods.
Resolution	✓ RESOLVED The `k` check has been refactored to the bottom of the `_swap`. We recommend extremely careful testing.

2.9 Core/UniswapV2ERC20

`UniswapV2Erc20` is an implementation of the ERC-20 Token Standard for denominating pool tokens. It is a fork of Uniswap's `UniswapV2ERC20` contract.

2.9.1 Issues & Recommendations

Issue #65	**`permit` can be frontrun to prevent someone from calling `removeLiquidityWithPermit` (also present in Uniswap)**
Severity	INFORMATIONAL
Description	If `permit` is executed twice, the second execution will revert. It is thus in theory possible for a bot to pick up `permit` transactions in the mempool and execute them before a contract can. The implications of this issue is that a bad actor could prevent a user from removing liquidity with a `permit` through the router. It is a denial of service attack which is present in all AMMs but which we have yet to witness being used since there is no profit from it.
Recommendation	Consider this issue if there are ever complaints by users that their `removeLiquidityWithPermit` transactions are failing. It could be the case that someone is using this vector against them. We do not recommend changing this behavior since it would cause a lot of extra work modifying the frontend to account for the new permit behavior. This issue is also present in Uniswap after all.
Resolution	✓ RESOLVED The client has indicated that they understand this issue and will mitigate it through the interface layer of the router once it starts presenting itself.

2.10 Core/Math, SafeMath and UQ112x112

`Math`, `SafeMath` and `UQ112x112` are various helper libraries which are each identical to the Uniswap implementation.

2.10.1 Issues & Recommendations

No issues found.

2.11 Periphery/CamelotRouter

`CamelotRouter` is a fork of the `UniswapV2Router` with slight modifications. It is the main interface for clients to interact with the Camelot DEX.

Compared to the `UniswapV2Router`, all normal swap functions have been removed:

- `swapExactTokensForTokens`
- `swapTokensForExactTokens`
- `swapExactETHForTokens`
- `swapTokensForExactETH`
- `swapExactTokensForETH`
- `swapETHForExactTokens`
- `_swap`

Therefore, the contract was simplified by only using the fee on transfer methods, which is not a problem as the functions would also work on normal tokens without such fees. The only limitation is that the user has no option to input an amount and receive an exact output amount as is possible with the traditional Uniswap router by using `swapETHForExactTokens` or `swapTokensForExactTokens`.

Finally, a referral mechanism was implemented within the contract: all swap methods have an additional referrer parameter which then gets passed to the pair's swap function and results in a referral fee which is paid to the referrer (if the fee is existent).

2.11.1 Issues & Recommendations

Issue #66	**The `quote` function returns erroneous values for the stableswap**
Severity	● LOW SEVERITY
Description	The `quote` function within the Uniswap router is supposed to give the swap result without any slippage or fees. However, presently the router uses traditional Uniswap V2 math to calculate this value, resulting in a wrongful output for stableswap pairs.
Recommendation	Consider adjusting this function to use the pair values or consider documenting this behavior. Be careful not to adjust the library for `quote` as the `addLiquidity` function uses `quote` to proportionally add liquidity (it should therefore remain unchanged within the library).
Resolution	✓ RESOLVED The client has updated the documentation of this function to better indicate their behavior.

Issue #67	**`receive()` lacks a safeguard**
Severity	● LOW SEVERITY
Description	Currently, the `receive()` fallback function is callable by everyone. However, it should only be callable by the WETH contract within the `withdraw` function. If it is called by anyone else, the Ether will just get stuck in the contract.
Recommendation	Consider safeguarding the `receive()` function appropriately.
Resolution	✓ RESOLVED

Issue #68	Gas optimizations
Severity	INFORMATIONAL
Description	Currently, the lock modifier is used for the following functions: - `swapExactTokensForTokensSupportingFeeOnTransferTokens` - `swapExactETHForTokensSupportingFeeOnTransferTokens` - `swapExactTokensForETHSupportingFeeOnTransferTokens` All of these functions call the swap function within the pair. Since this function is already safeguarded with a lock modifier, the present modifier can be removed from this contract.
Recommendation	Consider removing the lock modifier.
Resolution	✓ RESOLVED

Issue #69	The `addLiquidity` function does not properly support tokens with a fee on transfer (also present in Uniswap)
Severity	INFORMATIONAL
Description	`addLiquidity` always assumes tokens are not reflective. If liquidity is added where one of the tokens has a fee on transfer, tokens will be wasted to the pair. You can read more about this issue here: https://github.com/Uniswap/v2-periphery/issues/106
Recommendation	Consider implementing the approach described within issue 106 of v2-periphery or acknowledging this issue.
Resolution	● ACKNOWLEDGED

Issue #70	Phishing Issue: A malicious or hacked frontend could adjust routes, tokens or `to` parameters to steal tokens when users make swaps (also present in Uniswap)
Severity	INFORMATIONAL
Description	A malicious, e.g. compromised, frontend can easily mislead users into approving malicious transactions, even if the router matches the address described in this report. An trivial example of how this can be done is by changing the `to` parameter which indicates to whom tokens or liquidity has to be sent. Other ways to phish could include using malicious routes or tokens.
Recommendation	Consider carefully protecting the frontend and ideally having an unchangeable IPFS fallback implementation for it. Users should also verify that they are on the correct website when doing a swap.
Resolution	PARTIALLY RESOLVED The client has indicated they will carefully set up their frontend to minimize the risk of compromise.

2.12 Periphery/UniswapV2Library

`UniswapV2Library` is used to perform some common calculations like the amount received from a swap. The Camelot team has slightly modified it to work with the variable swap fee.

2.12.1 Issues & Recommendations

No issues found.

PALADIN
BLOCKCHAIN SECURITY