



**PALADIN**  
BLOCKCHAIN SECURITY

# Smart Contract Security Assessment

Final Report

For Camelot (Nitro & Presale)

14 November 2022



[paladinsec.co](https://paladinsec.co)



[info@paladinsec.co](mailto:info@paladinsec.co)

# Table of Contents

Table of Contents	2
Disclaimer	3
1 Overview	4
1.1 Summary	4
1.2 Contracts Assessed	4
1.3 Findings Summary	5
1.3.1 NitroPool	6
1.3.2 NitroPoolFactory	6
1.3.3 Presale	7
2 Findings	8
2.1 NitroPool	8
2.1.1 Privileged Functions	11
2.1.2 Issues & Recommendations	12
2.2 NitroPoolFactory	20
2.2.1 Privileged Functions	21
2.2.2 Issues & Recommendations	22
2.3 Presale	24
2.2.1 Privileged Functions	26
2.2.2 Issues & Recommendations	27

# 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 advocacy 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

<b>Project Name</b>	Camelot
<b>URL</b>	<a href="https://app.camelot.exchange">https://app.camelot.exchange</a>
<b>Network</b>	Arbitrum
<b>Language</b>	Solidity

## 1.2 Contracts Assessed

Name	Contract	Live Code Match
NitroPool	Deployed by NitroPoolFactory	✓ MATCH
NitroPoolFactory	0xe0a6b372Ac6AF4B37c7F3a989Fe5d5b194c24569	✓ MATCH
Presale	0x66eC1EE6c3AD04d7629Ce4a6d5d19ba99c365d29	✓ MATCH

## 1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
● High	1	1	-	-
● Medium	2	2	-	-
● Low	8	8	-	-
● Informational	8	6	-	2
<b>Total</b>	<b>19</b>	<b>17</b>	<b>-</b>	<b>2</b>

### 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 NitroPool

ID	Severity	Summary	Status
01	HIGH	_harvest is flawed in several ways if a user withdraws before harvesting is allowed	RESOLVED
02	MEDIUM	onNFTHarvest might start reverting as it blindly attempts to forward the full amounts	RESOLVED
03	MEDIUM	A malicious rewardsToken2 can be added after the pool is published, blocking rewards and forcing emergency withdrawals	RESOLVED
04	LOW	Lack of handling of ownership renunciation	RESOLVED
05	LOW	emergencyWithdraw can still revert in the edge case that the rewardDebt calculation overflows	RESOLVED
06	LOW	isValidNFTPool1 will break if it is used on any pool other than msg.sender	RESOLVED
07	LOW	addRewards remains callable at the endTime while these rewards may not be claimable	RESOLVED
08	INFO	Lack of reentrancy guards on various functions	RESOLVED
09	INFO	Contract does not support deposit fee tokens	RESOLVED
10	INFO	Typographical errors and gas optimizations	RESOLVED

## 1.3.2 NitroPoolFactory

ID	Severity	Summary	Status
11	LOW	Contract is not well suited for a create2 deployment pattern	RESOLVED
12	INFO	Typographical errors	RESOLVED
13	INFO	Gas optimization	RESOLVED

### 1.3.3 Presale

ID	Severity	Summary	Status
14	LOW	Any xGrail reapproval attempt will fail and permanently prevent claiming on the contract	✓ RESOLVED
15	LOW	Users will be unable to claim grail if their xGrail amount is zero	✓ RESOLVED
16	LOW	Contract might not be compatible with all USDC deployments	✓ RESOLVED
17	INFO	Lack of validation	✓ RESOLVED
18	INFO	Gas optimizations	ACKNOWLEDGED
19	INFO	Typographical errors	ACKNOWLEDGED



# 2 Findings

---

## 2.1 NitroPool

The NitroPool contracts are staking pools which emit extra rewards on top of the rewards from locking an LP position into an NFTPool. The idea of the NitroPool is that if your locked LP position meets certain requirements (such as lock duration), you can then stake the locked LP position into the NitroPool for additional rewards.

The NitroPool defines up to two additional reward tokens.

Deposits are made into the NitroPool simply by transferring your NFTPool NFT to it. This transfer will solely succeed if the necessary requirements are met for the NFT:

- It must be from the correct NFT pool
- The lock duration must exceed the configured minimum
- The lock expiry must be after the configured minimum
- The lock amount must exceed the minimum configured amount

The pool will continuously emit deposited tokens until the configured endTime of the pool to all stakers proportional to their staked amounts. Withdrawals can however be made at any time. The contract also contains a safer emergency withdrawal function which continues working even if various reward calculations for any reason would start reverting.

When a deposit is made, the users receive approval for the token. This means that they can still use their approval to freely operate on the token: harvest the underlying pool, add to the position, etc. However, even though the approval would typically allow them to transfer out the token, which would be undesirable, this is prevented since NFTPool implementations revert on such transfers as transfers from a contract can only be made if the operator is said contract.



The pool owner can at any point make the pool whitelisted, which means that any subsequent depositors can only deposit into the pool if they are whitelisted by the owner.

For deposits to be enabled, the pool must have been published. This is done by the pool owner when they call the `publish` function which can only be called once.

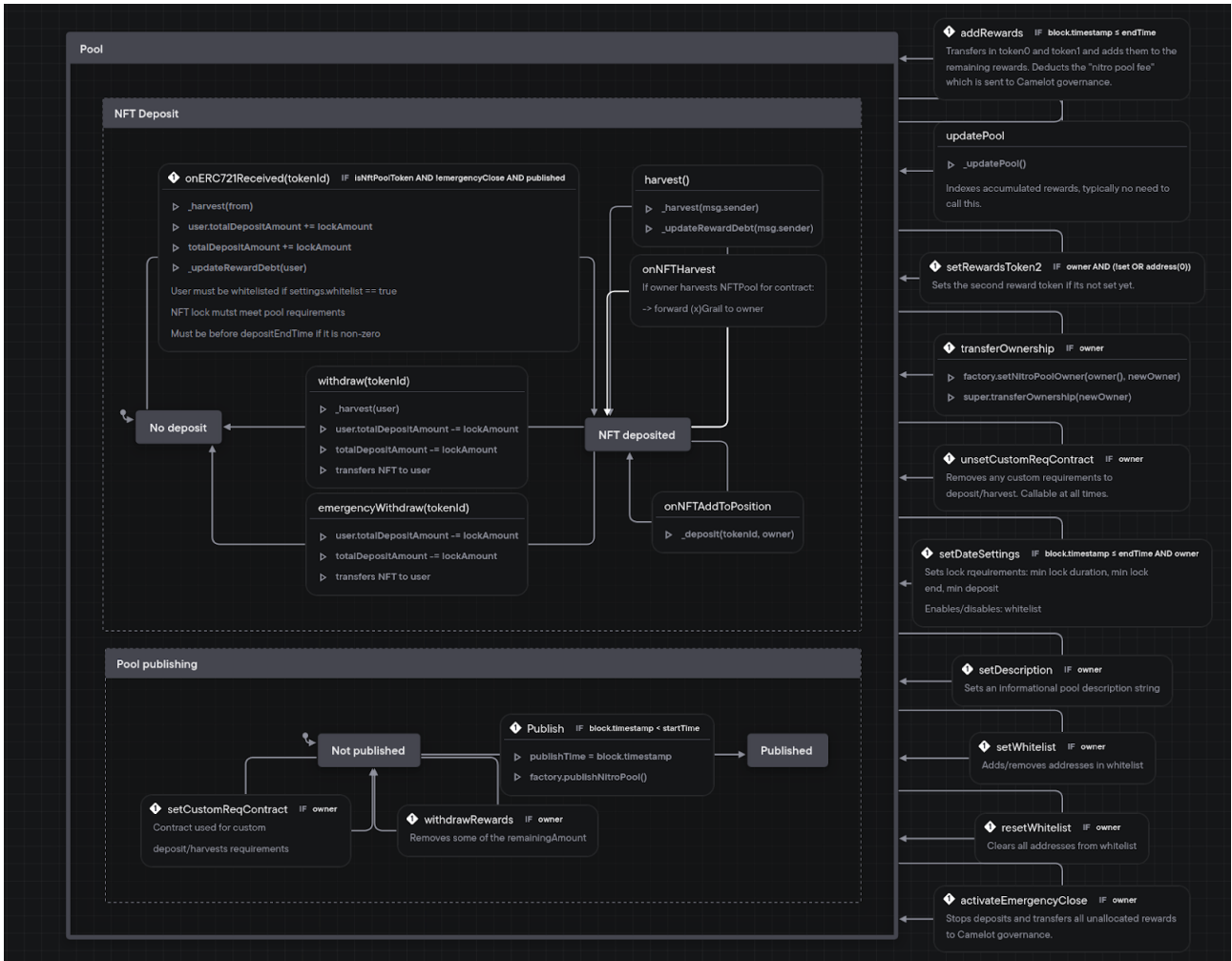
Various functions become more strict after a pool has been published:

- The custom requirements contract which adds additional safeguards on harvests and deposits can no longer be changed unless it's fully removed
- `withdrawRewards` can no longer be called, solely `activateEmergencyClose`
- Deposit requirements can only be adjusted to make the requirements more strict
- Most date settings can no longer be adjusted, only the `endTime` remains extendable

Finally, the owner can define a date settings which limit the user: They can define the last time where deposits are still allowed and they can define the earliest time when harvests become available.

The full `NitroPool` process and how its encoded within the smart contract can be seen in the following diagram:





## 2.1.1 Privileged Functions

- withdrawRewards
- setRewardsToken2
- setCustomReqContract
- setRequirements
- setDateSettings
- setDescription
- setWhitelist
- resetWhitelist
- publish
- activateEmergencyClose
- transferOwnership
- renounceOwnership



## 2.1.2 Issues & Recommendations

**Issue #01**      **\_harvest is flawed in several ways if a user withdraws before harvesting is allowed**

**Severity**

 HIGH SEVERITY

**Description**

\_harvest is flawed in multiple ways: First of all, crucial logic to payout cached harvests is not triggered if the current pending is zero. This means that if the user has cached a harvest and withdrawn their position afterwards, they will not be able to harvest their previously cached amount.

Secondly, a large error appears to occur on the following line:

Line 729

```
if (_currentBlockTimestamp() < settings.harvestStartTime &&  
canHarvest) {
```

This code is supposed to execute while no harvests are permitted, eg. we are before the harvest start time. However, the second parameter triggers when the customReqContract permits harvests. It should therefore say !canHarvest.

Lastly, the canHarvest check is missing in the second reward token.

**Recommendation**

Consider refactoring the \_harvest function to always payout pending rewards and to properly account for canHarvest.

Consider adding the canHarvest check to the second reward token.

**Resolution**

 RESOLVED

The recommended refactoring has been implemented.

**Issue #02****onNFTHarvest might start reverting as it blindly attempts to forward the full amounts****Severity** MEDIUM SEVERITY**Description**

The onNFTHarvest function blindly forwards the input amounts. However, these amounts are not always granted to users. This is because once the balance of the NFTPool runs too low, only the remainder will be sent as rewards.

**Recommendation**

Consider updating the NFTPool code to forward the actual amounts sent to onNFTHarvest. This makes more sense in our opinion and arguably should have been a recommendation if the NFTPool audit had considered this usage.

**Resolution** RESOLVED

This was fixed as recommended within the NFTPool, which is out of scope. This fix will be more formally attested within the live match of that audit.

**Issue #03****A malicious rewardsToken2 can be added after the pool is published, blocking rewards and forcing emergency withdrawals****Severity** MEDIUM SEVERITY**Description**

The NitroPool owner may add a second reward token at any time, as long as one was not set previously. This presents a griefing risk vector.

If the second reward token is malicious and reverts on transfer, users will be unable to claim any rewards for either reward token. At this point, the only way for a user to retrieve their NFT is via emergency withdraw, forgoing earned rewards.

**Recommendation**

Consider requiring the second reward token to be set prior to publishing. This will allow users and the protocol frontend to do appropriate due diligence on the pool's tokens.

**Resolution** RESOLVED

**Issue #04**      **Lack of handling of ownership renunciation**

**Severity**      ● LOW SEVERITY

**Location**      Line 391  
`function transferOwnership(address newOwner) public override  
onlyOwner {`

**Description**      The codebase overrides the `transferOwnership` function with custom logic which should always execute on ownership handover. However, another method can be used for ownership handover as well: `renounceOwnership`.


The latter is not overridden and the custom logic is hence not executed here.

**Recommendation**      Consider overriding the internal `_transferOwnership` instead.

**Resolution**      ✓ RESOLVED

`renounceOwnership` is now handled as well, the external methods are however overridden which is slightly less efficient.



**Issue #05****emergencyWithdraw can still revert in the edge case that the rewardDebt calculation overflows****Severity** LOW SEVERITY**Description**

The emergencyWithdraw function adjusts the rewardDebt by adjusting it to the current debt excluding the withdrawn tokens.

However, if a bad token that could be infinitely minted was added as a reward token, it is possible to increase the rewardDebt calculation so much that the multiplication portion of it overflows and reverts. This scenario would hence still block all withdrawals.

**Recommendation**

Consider using tryMul instead with the rewardDebt calculation. This way overflow can be handled explicitly within emergencyWithdraw to allow the withdrawal to still succeed.

**Resolution** RESOLVED

A tryMul has been implemented. However, we need to point out that the failure should still revert in deposit and withdraw as currently it is ignored there. This should only be a risk for tokens with a very large supply, in which case an exploiter could potentially claim excessive rewards. Small supply tokens, eg. almost all normal tokens in existence, should still not allow for exploitation.

**Issue #06** **isValidNFTPool will break if it is used on any pool other than msg.sender**

**Severity** LOW SEVERITY

**Location** Line 244-247  

```
modifier isValidNFTPool(address sender) {  
    require(msg.sender == address(nftPool), "invalid  
NFTPool");  
    -;  
}
```

**Description** The isValidNFTPool function takes in a sender function which is unused. If this function is ever used with anything but the msg.sender, this would cause severe malfunction.

Right now the sender is always msg.sender, so no user-facing concerns arise.

**Recommendation** Consider using the sender parameter:  

```
require(sender == address(nftPool), "invalid NFTPool");
```

**Resolution** RESOLVED

**Issue #07** **addRewards remains callable at the endTime while these rewards may not be claimable**

**Severity** LOW SEVERITY

**Location** Line 402  

```
require(_currentBlockTimestamp() <= settings.endTime, "pool  
has ended");
```

**Description** If a harvest occurs at the last second of the emissions, addRewards remains callable but these rewards will never be harvested.

**Recommendation** Consider making the check exclusive:  

```
require(_currentBlockTimestamp() < settings.endTime, "pool  
has ended");
```

**Resolution** RESOLVED



**Issue #08****Lack of reentrancy guards on various functions****Severity** INFORMATIONAL**Description**

The codebase makes a strong attempt at safeguarding itself against reentrancy. Even though no reentrancy exploits appear to be possible, we will include an issue in good faith to allow the client to harden their code further.

The following functions lack reentrancy guards:

- updatePool
- activateEmergencyClose
- setRewardsToken2 (optionally)
- setDateSettings (optionally)

**Recommendation**

Consider adding reentrancy guards to the above functions. activateEmergencyClose should also adhere to the checks-effects-interaction pattern.

**Resolution** RESOLVED**Issue #09****Contract does not support deposit fee tokens****Severity** INFORMATIONAL**Description**

The contract does not work with deposit fee reward tokens. This is because the addRewards function blindly assumes that the transferred tokens have been received.

**Recommendation**

Consider whether such tokens ever need to be supported. If so, consider adding a before-after pattern to addRewards. Alternatively, this issue will be resolved on the note that such tokens are not permitted within this contract.

**Resolution** RESOLVED

The addRewards function now supports tokens with a fee on transfer.

**Description**

We have consolidated the typographical errors and the sections which can be further optimized for gas usage below.

Line 95

```
IGrailTokenV2 grailToken_, IXGrailToken xGrailToken_,  
address owner_, INFTPool nftPool_, address rewardsToken1_,  
address rewardsToken2_, Settings calldata settings_
```

The tokens can be provided as IERC20 to avoid casting. Line 104 would then furthermore cast these addresses correctly.

Line 112

```
if (rewardsToken2_ != address(0)) {
```

This can in our opinion remain implicit which would save a few lines (by always setting the token).

Line 120

```
if (settings_.harvestStartTime == 0)  
settings.harvestStartTime = settings.startTime;
```

settings\_ can be used on the right hand side to save some gas.

Line 228

```
accRewardsToken1PerShare_ =  
rewardsToken1.accRewardsPerShare.add(rewardsAmount.mul(1e18)  
.div(totalDepositAmount));
```

Consider incrementing the memory variable instead of re-fetching the rewardsToken1.accRewardsPerShare from storage again. The same can be set for line 233.

Line 395

```
emit TransferOwnership(newOwner);
```

This event is already emitted at a lower level by Ownable.

---

Line 417

```
else delete customReqContract;
```

This can be simplified: The if statement appears to be completely redundant as this deletion zeroes it out as well.

Line 495

```
if(contractAddress != address(0)){
```

The UpdatePool event emits a timestamp which seems redundant.

---

**Recommendation** Consider fixing the typographical errors.

---

**Resolution**



---

## 2.2 NitroPoolFactory

NitroPoolFactory is responsible for deploying new NitroPool instances. The contract defines a `defaultFee` which is used as the default fee share percentage for NitroPool rewards. This fee can be zeroed out if the factory owner marks the Nitro pool or its owner as exempt.

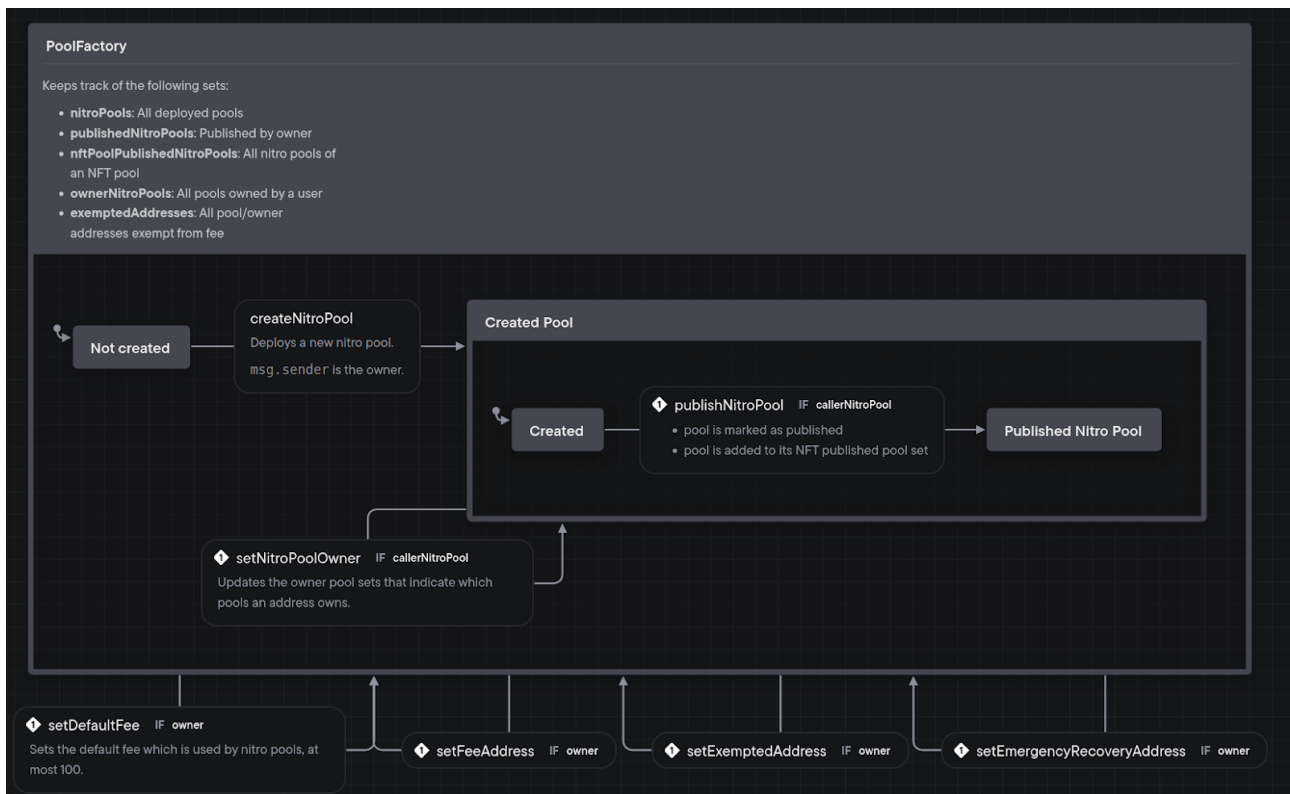
The factory defines various other utilities which are used by the pools themselves. It contains various enumerable sets which can be enumerated by the frontend. The pool owner can finally set various global variables:

- `defaultFee`: Documented above
- `feeAddress`: The global Camelot governance address which receives the fees
- `exemptedAddresses`: The pool and owner addresses without a governance fee
- `emergencyRecoveryAddress`: The global Camelot governance address which receives all reward tokens when `activateEmergencyClose()` is called by a pool owner

It should be noted that anyone can create a NitroPool and the relevant parameters to create such a pool are not validated. This means that malicious pools may exist with malicious tokens (NFT/ERC20). The frontend should only display trusted pools.

The full NitroPoolFactory process and how its encoded within the smart contract can be seen in the following diagram:







## 2.2.1 Privileged Functions

- `publishNitroPool [ nitroPool, callable once ]`
- `setNitroPoolOwner [ nitroPool ]`
- `setDefaultFee [ owner ]`
- `setFeeAddress [ owner ]`
- `setExemptedAddress [ owner ]`
- `setEmergencyRecoveryAddress [ owner ]`
- `transferOwnership`
- `renounceOwnership`

## 2.2.2 Issues & Recommendations

<b>Issue #11</b>	<b>Contract is not well suited for a create2 deployment pattern</b>
<b>Severity</b>	 LOW SEVERITY
<b>Location</b>	<u>Line 173</u> <pre>bytes32 salt = keccak256(abi.encodePacked(nftPoolAddress, rewardsToken1, rewardsToken2, _currentBlockTimestamp()));</pre>
<b>Description</b>	<p>The NitroPool is deployed using a unique salt which is the input to generate the relevant deployment address. However, as this salt is based on an ever-changing deployment timestamp, there is very little value to using a salt.</p> <p>Typically, a deterministic deployment salt is used so that other contracts can figure out the address of a pool without explicitly having to ask the factory. However, since the timestamp is always changing, this is simply impossible.</p>
<b>Recommendation</b>	Consider moving to a standard deployment pattern, and if so, consider moving to a constructor for the NitroPool.
<b>Resolution</b>	 RESOLVED A normal contract instantiation is now utilized.

**Issue #12**      **Typographical errors**

**Severity**      INFORMATIONAL

**Description**      We have consolidated the typographical errors into a single issue to keep the report brief and readable.

Line 6

```
import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
```

This import appears to be unused.

Line 143

```
* @dev Returns an exemptedAddress from its "index"
```

This comment is wrongly copied as the relevant function returns whether the address is exempted or not.

**Recommendation**      Consider fixing the typographical errors.

**Resolution**      RESOLVED

**Issue #13**      **Gas optimization**

**Severity**      INFORMATIONAL

**Location**      Line 210  

```
require(_ownerNitroPools[previousOwner].contains(msg.sender)  
, "invalid owner");
```

**Description**      It is sufficient to require the remove call as it returns a success boolean as well.

**Recommendation**      Consider implementing the gas optimization above.

**Resolution**      RESOLVED

---

## 2.3 Presale

The Presale contract is designed to sell a number of Grail tokens for a number of SALE\_TOKENs (presumably a stablecoin) as a public raise.

While the sale is active, users can call the buy function with the amount of SALE\_TOKEN (expected to be USDC) they wish to spend. If the user provides a referral, 3% of those tokens will go to the referral while the remainder is always immediately sent to the Camelot treasury. In exchange for these tokens, the user receives presale allocation which they will be allowed to exchange for both Grail and xGrail tokens after the sale has ended. Exactly 35% of the tokens eligible to a user will be given as xGrail.

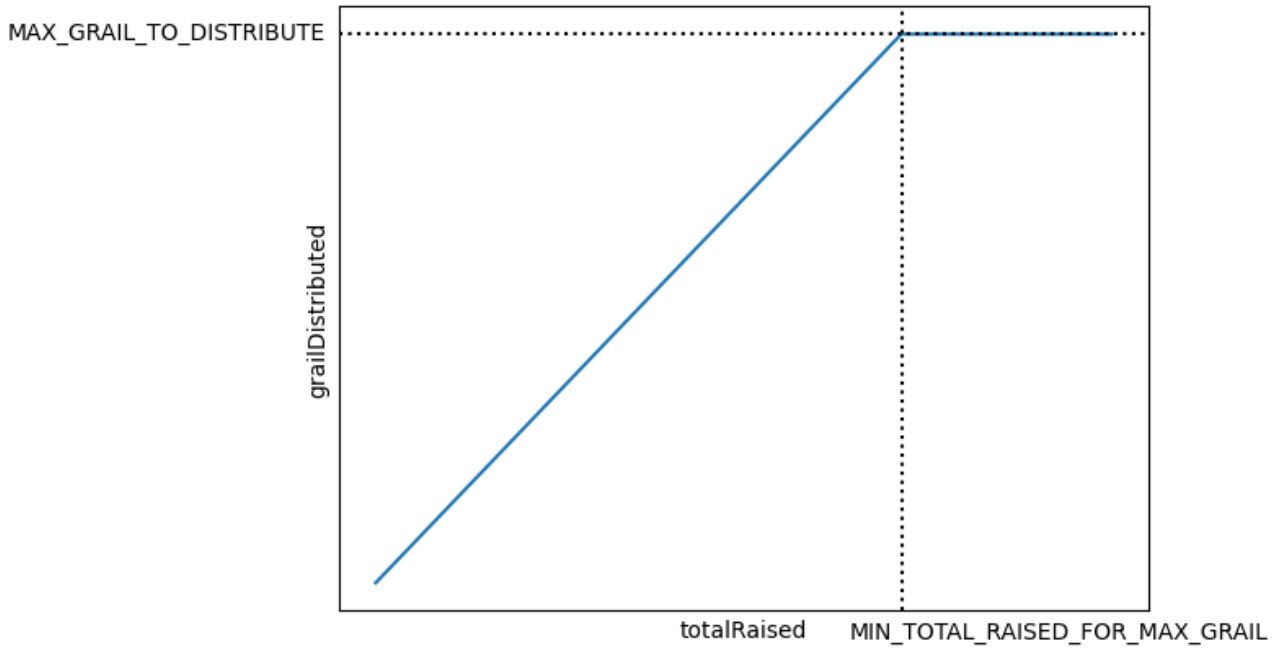
The presale is configured to distribute exactly 15,000 tokens in total, for a desired total of 300,000 USDC. Each token is therefore sold at a price of \$20. If more than 300,000 tokens have been sold, the price will be higher.

Finally, the team is free to assign a discount to specific users.

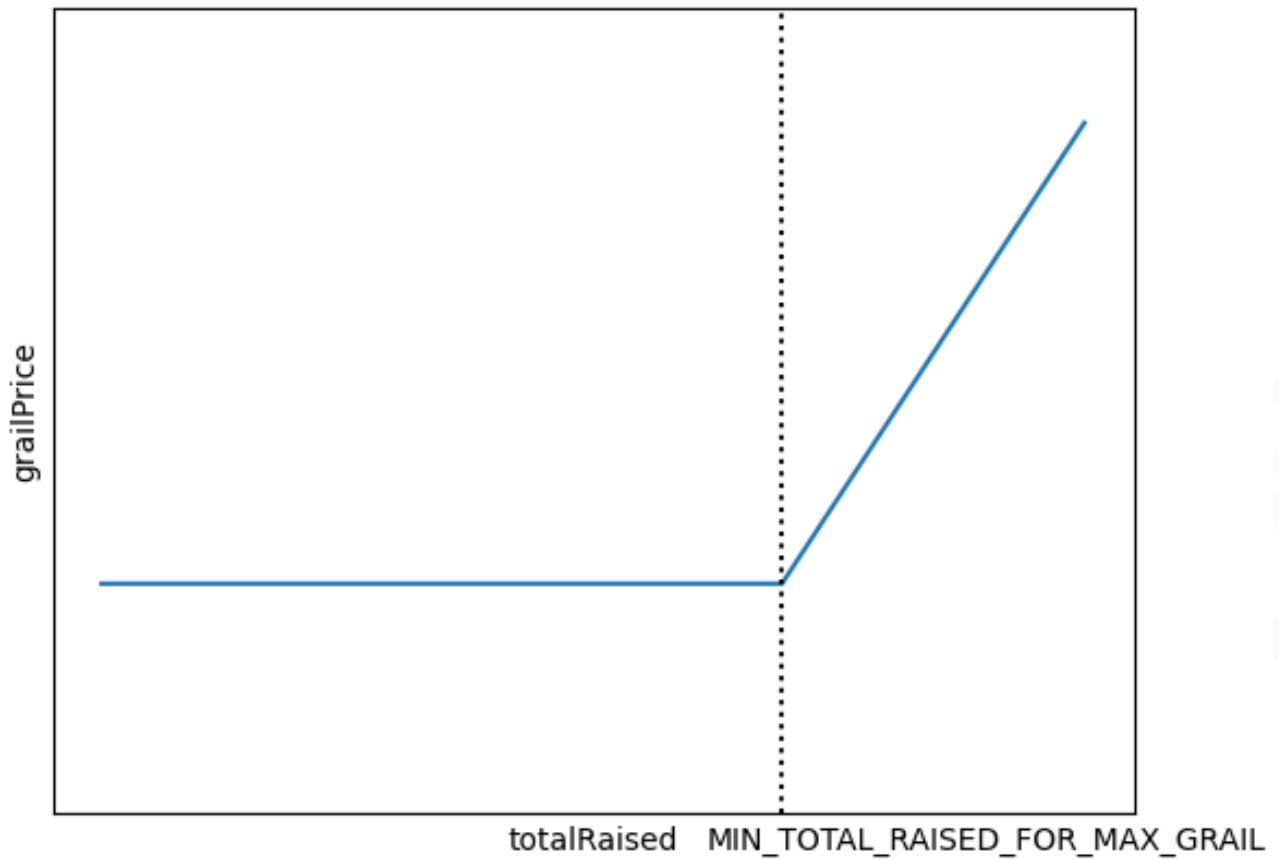
The number of Grail which will be distributed to users is capped via the MAX\_GRAIL\_TO\_DISTRIBUTE variable, which is reached once the MIN\_TOTAL\_RAISED\_FOR\_MAX\_GRAIL has been reached.



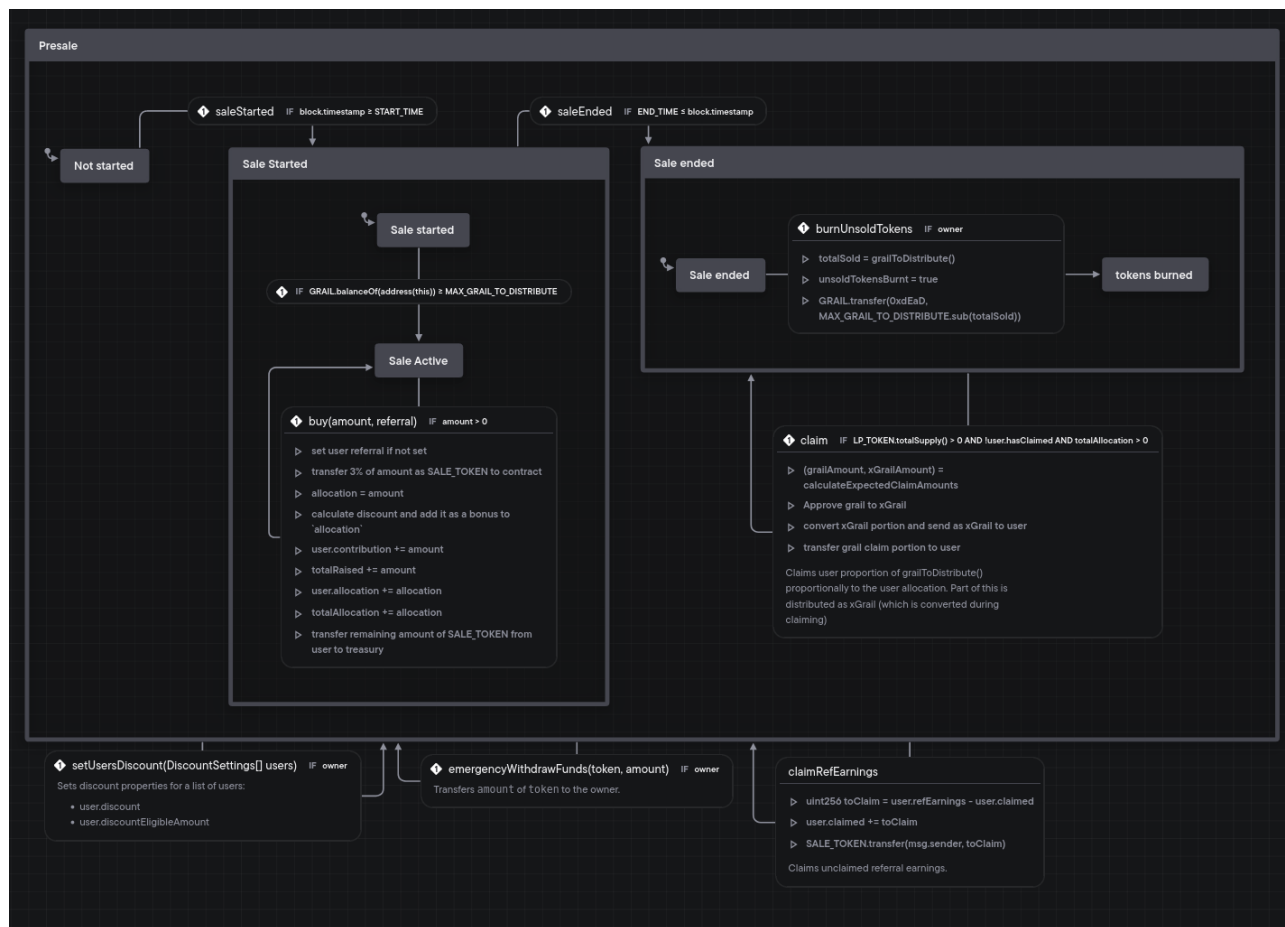




This means that as long as there is small to moderate interest in the presale, users will be paying a constant price. However, once more than the minimum for maximum Grail has been reached, all users will pay an increasingly expensive price as no further Grail will be emitted for new purchases:



The full presale process and how it is encoded within the smart contract can be seen in the following diagram:





## 2.2.1 Privileged Functions

- setUsersDiscount
- emergencyWithdrawFunds
- burnUnsoldTokens
- transferOwnership
- renounceOwnership

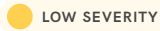
## 2.2.2 Issues & Recommendations

<b>Issue #14</b>	<b>Any xGrail reapproval attempt will fail and permanently prevent claiming on the contract</b>
<b>Severity</b>	 LOW SEVERITY
<b>Location</b>	<u>Line 241-243</u> <pre>if (GRAIL.allowance(address(this), address(XGRAIL)) &lt; xGrailAmount) {     GRAIL.safeApprove(address(XGRAIL), uint256(-1)); }</pre>
<b>Description</b>	<p>The safeApprove attempt will only work the first time it is called. Subsequent calls will almost always fail as safeApprove by OpenZeppelin is a notoriously annoying function:</p> <pre>require((value == 0)    (token.allowance(address(this), spender) == 0), "SafeERC20: approve from non-zero to non- zero allowance");</pre>
<b>Recommendation</b>	<p>Consider using a normal approval instead, or the recommended OZ pattern:</p> <pre>GRAIL.safeApprove(address(XGRAIL), 0); GRAIL.safeApprove(address(XGRAIL), type(uint256).max);</pre> <p>Note that we recommend using <code>type(uint256).max</code> as it is more readable than <code>uint256(-1)</code>.</p>
<b>Resolution</b>	 RESOLVED

**Issue #15**      **Users will be unable to claim `grail` if their `xGrail` amount is zero**

<b>Severity</b>	
<b>Location</b>	<u>Line 246</u> <code>XGRAIL.convertTo(xGrailAmount, msg.sender);</code>
<b>Description</b>	The <code>convertTo</code> function reverts on zero amount. If the user purchases an exceptionally small amount of tokens, the <code>convertTo</code> here would revert as well and users would not be able to claim the remaining Grail tokens.
<b>Recommendation</b>	Consider either accepting this as users will never purchase such a small amount, or consider adding a non-zero check in line with correctness.
<b>Resolution</b>	

**Issue #16**      **Contract might not be compatible with all USDC deployments**

<b>Severity</b>	
<b>Location</b>	<u>Line 45</u> <code>uint256 public constant MIN_TOTAL_RAISED_FOR_MAX_GRAIL = 300000000000;</code>
<b>Description</b>	The contract presently assumes a USDC token with 6 decimals. If the contract is ever used on a chain with a different number of decimals for the stablecoin, the contract will misbehave and tokens might all get sold for a fraction of the expected sale amount.
<b>Recommendation</b>	Consider either carefully documenting this or scaling the value by the decimals of the USDC to be safe.
<b>Resolution</b>	 <p>The contract now clearly documents it only works with a 6 decimal USDC. The client and developers who fork this contract should of course remain careful.</p>

**Issue #17****Lack of validation****Severity** INFORMATIONAL**Description**

The contract contains functions with parameters which are not properly validated. Having unvalidated parameters could allow the governance or users to provide variable values which are unexpected and incorrect. This could cause side-effects or worse exploits in other parts of the codebase.

Consider validating `setUsersDiscount`: The discount should be validated to be smaller or equal to 100 if this is how it is supposed to be used.

**Recommendation**

Consider validating the function parameter mentioned above.

**Resolution** RESOLVED

The discount is now capped to 35%.



**Issue #18****Gas optimizations****Severity** INFORMATIONAL**Description**

We have consolidated the sections which can be further optimized for gas usage below.

Line 24

```
address ref;
```

By moving this address to right above `hasClaimed`, it will be packed together with the `hasClaimed` boolean and save a storage slot, which can reduce gas costs significantly. Optionally, `discount` could also be packed in this slot as it would still fit in a `uint8`. This `discount` can also be packed within the `DiscountSettings` struct.


Line 133-138

```
function grailToDistribute() public view returns (uint256){  
    if (MIN_TOTAL_RAISED_FOR_MAX_GRAIL > totalRaised) {  
        return  
MAX_GRAIL_TO_DISTRIBUTE.mul(totalRaised).div(MIN_TOTAL_RAISE  
D_FOR_MAX_GRAIL);  
    }  
    return MAX_GRAIL_TO_DISTRIBUTE;  
}
```

`totalRaised` is read from storage twice, less of a big deal.

**Recommendation**

Consider implementing the gas optimizations mentioned above.

**Resolution** ACKNOWLEDGED

## Severity

INFORMATIONAL

## Description

We have consolidated the typographical errors into a single issue to keep the report brief and readable.

Line 140-142

```
/**  
* @dev Get user share times 1e5  
*/
```

Line 194

```
// Readjust user new allocation
```

The lines above should state *user's*.

`claimRefEarnings` and `setUsersDiscount` can be marked as `external`. Note that we also recommend marking `claimRefEarnings` as `non-reentrant` as well if there is a reentrancy threat on buy.

Line 242

```
GRAIL.safeApprove(address(XGRAIL), uint256(-1));
```

A cleaner way to get the max `uint256` is using `type(uint256).max`.

Line 283

```
function emergencyWithdrawFunds(address token, uint256  
amount) external onlyOwner
```

`token` can be provided as an `IERC20`.

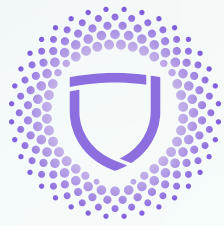
`burnUnsoldTokens` finally should emit an event (it should be noted that it might also make sense to use `safeTransfer` within this function).

## Recommendation

Consider fixing the typographical errors.

## Resolution

ACKNOWLEDGED



**PALADIN**  
BLOCKCHAIN SECURITY