BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT

KernelDao

February 2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:

Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Kernel Dao
Website	kerneldao.com
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/Kelp-DAO/KelpDAO-contracts/tree/dcc956dcdfdb25d3b27e016d7c526eb8ff9ffb40/contracts/KERNEL
Resolution 1	https://github.com/Kelp-DAO/KelpDAO-contracts/commit/d377f3e08259f20806e7bba34e4fc79bbe652574
Resolution 2	https://github.com/Kelp-DAO/KelpDAO-contracts/tree/11f89990ed7e89a8e73781b402f71edb6830ad23/contracts/KERNEL

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed Resolution
High	1	1			
Medium	2	1	1		
Low	2	2			
Informational	5	4		1	
Governance	1			1	
Total	11	8	1	2	

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

KernelMerkleDistributor

The KernelMerkleDistributor contract is used by the protocol for distributing tokens to users based on a merkle root. Its practical implication is to airdrop KERNEL (unified governance token across Kernel's ecosystem) to a defined group of accounts.
KERNEL recipients can choose to either receive them directly or to re-route them to the KernelDepositPool contract to be staked and earn reward. The main interactions possible are:

- Claim: The caller provides the account address alongside the data required for the merkle proof verification and if successful, the respective amount of KERNEL is transferred to the account.
- ClaimAndStake: This flow builds up on the claiming one by staking the account tokens inside the KernelDepositPool on behalf of the account instead of sending them out.

Both functions are public, meaning that anyone can call them. However the assets would always be sent out to the account that was included in the proof, allowing gas prices for claiming to be paid by another party.

Privileged functions:

- setKernelDepositPool
- setProtocolTreasury
- setFeeInBPS
- setMerkleRoot
- pause / unpause

Appendix: Core Invariants

INV 1: claim() & claimAndStake() should only transfer rewards to accounts that have been included in the merkle root

INV 2: Each account should claim its reward only once per index

INV 3: The reward amount should only increase for an account with every new index

INV 4: claimAndStake() must always stake the tokens on behalf of the account that should receive them

INV 5: Merkle root updates should not break index distributions from previous versions of the root

INV 6: Upon claiming the claimable amount should always be the difference between the already claimed amounts and the total reserved to be distributed for that account

Issue_01	Governance Privileges
Severity	**Governance**
Description	The contract owner has full control over several variables that can impact the outcome of a transaction: - set the pool for depositing airdropped KERNEL tokens - set the protocol treasury - set the fees that det deducted upon claiming - update the merkle root - pause/unpause the protocol
Recommendations	Consider incorporating a Gnosis multi-signature contract as the owner and ensuring that the Gnosis participants are trusted entities
Comments / Resolution	Acknowledged.

Issue_02	Claims can be front-runned to prevent KERNEL tokens from being directly received by an account
Severity	**Low**
Description	The contract exposes two flows for claiming KERNEL tokens: - claim() - sends the tokens directly to the account - claimAndStake() - stakes the tokens in the pool Since the 2 functions are permissionless and share the same merkle tree verification logic, the following scenario is possible: 1. Bob is entitled to claim 5000 tokens and doesn't want to stake them, but rather wants to take them directly 2. Bob sends a claim() transaction 3. Alice frontruns claim(), by calling claimAndStake() using the arguments from Bobs transaction 4. As result the tokens are staked and Bob does not receive them directly Issue is low, since the only effect of it is that the account would have to additionally call the staking contract to withdraw the staked funds on its behalf
Recommendations	Since both functions can be called by anyone, this is rather an inherent issue of the design. The straightforward solution is to allow only the account receivers to claim their tokens. If the team wants the claim flow to be publicly callable (probably due to gas payments), then users must be warned of this drawback.

Comments / Resolution	Resolved by verifying msg.sender.

Issue_03	Frontrunning claimAndStake can be used to decrease one’s rewards
Severity	**Low**

Description	The claimAndStake function does not verify the msg.sender:
	
	function claimAndStake(
	        uint256 index,
	        address account,
	        uint256 cumulativeAmount,
	        bytes32[] calldata merkleProof
	    )
	        external
	        whenNotPaused
	        nonReentrant
	    {
	        uint256 amountToStake = _processClaim(index, account, cumulativeAmount, merkleProof);
	
	This will allow users to front-run the claimAndStake and call claim instead. As a result they will slow down the user’s stake, and deprive them from some reward. This could cause some loss for the victim if the rewardRate is high and the amountToStake is also a big amount. Some stakers could be incentivised to perform such attacks in order to slightly extend their profit.
	
	This issue is similar to the above mentioned but results in some opportunity cost.

Recommendations	Consider verifying the msg.sender

Comments / Resolution	Resolved by following recommendations.

Issue_04	Insufficient validation within claim and claimAndStake
Severity	**Informational**
Description	Inside the claim() & claimAndStake() functions check if the provided account argument by the caller is not address(0). It is unnecessary to execute logic with invalid data. Reverting early reduces gas consumption and sanitizes the function.
Recommendations	Consider implementing the above mentioned recommendation.
Comments / Resolution	Resolved.

Issue_05	Lack of reasonable limits for the fees upon claiming
Severity	**Informational**
Description	The setFeeInBPS() function is an admin restricted function that is used to define the percent of fees that will be deducted from the amount of tokens that an account should receive upon claiming. Currently the fees are not restricted in any way and can be set as high as 100%. Considering the governance nature of the protocol it makes sense to implement a transparent limit for the amount of fees that can be deducted from the received tokens
Recommendations	Consider defining a MAX_FEE constant variable and enforce it inside the function - this will prevent management from abusing it and setting it to an unreasonably high value - such as 100%, which does not make sense
Comments / Resolution	Resolved by following the recommendation.

KernelDepositPool

The KernelDepositPool contract is a derivative of the Synthetix staking rewards contract. It is being used to distribute rewards among the KERNEL holders that have staked their tokens in the contract.
For the most part the logic is a modified version of the Synthetix implementation, with the small addition that a new function called stakeFor() has been introduced. The function allows an account that is granted the STAKE_FOR_ROLE to stake KERNEL on behalf of another account. This newly added flow is used by the KernelMerkleDistributor contract to stake airdropped tokens on behalf of the accounts that receive them.The main flows exposed by the contract are:

- Stake: Allows a user to stake a specified amount of staking tokens
- StakeFor: Called by Kernel distributor to stake airdropped tokens
- Withdraw: Allows a user to withdraw staked tokens
- GetReward: Allows a user to claim their earned rewards

Privileged functions:
- setRewardsDuration
- notifyRewardAmount

Appendix: Reward distribution
The contract uses a synthetic-like approach to distribute rewards to the accounts that have staked KERNEL tokens inside the contract.

All rewards are accumulated in a global storage variable called rewardPerTokenStored, that accounts for the amount of rewards that have to be distributed for each token staked in the pool.

Upon staking the account is registered into a mapping where the current value of rewardPerTokenStored is snapshotted. Later when the account withdraws the difference between the current value and the one from the snapshot is what constitutes the accumulated rewards.

The global and per account storage variables that keep track of the accumulated rewards are always updated prior to any state changing interaction with the contract to ensure consistent and reliable accounting and distribution

Appendix: Core Invariants

INV 1: stake() & stakeFor() should increment the total and per account accumulator variables by the exact amount of tokens being transferred

INV 2: withdraw() must allow staked to get their assets at any time

INV 3: withdraw() must decrement the accumulator variables by the exact amount of tokens being transferred out from the contract

INV 4: getReward() always transfers the full amounts of accumulated rewards for an account
INV 5: reward duration can be updated only after the current distribute period has concluded

INV 6: commencing a new distribution period should always include the undistributed rewards from the current distribution

INV 7: rewards are accumulated only during the distribution period

INV 8: rewards are distributed only among tokens holders with staked tokens in the contract

Issue_06	rewardsBalance does not account new rewards correctly
Severity	**High**

Description	The notifyRewardAmount will try to add the new reward to the rewardsBalance:
	if (block.timestamp >= finishAt) {
	rewardRate = _amount / duration;
	rewardsBalance = _amount;
	} else {
	*uint256 remaining = (finishAt - block.timestamp) * rewardRate;*
	rewardRate = (_amount + remaining) / duration;
	rewardsBalance = _amount + remaining;
	}
	However instead of increasing the rewardsBalance, the function will actually erase the previous balance. As a result some users will be unable to withdraw due to underflow of rewardsBalance.

Recommendations	Consider calculating the reward based on the received reward amount, by sending it in the notifyRewardAmount function.
Comments / Resolution	Resolved.

Issue_07	Insufficient _amount validation in notifyRewardAmount
Severity	**Medium**
Description	The notifyRewardAmount function will perform the following verification to ensure the amount is enough to cover for the future rewards: *if (rewardRate * duration > rewardsToken.balanceOf(address(this))) revert RewardAmountGreaterThanBalance();* This however is not sufficient since the balance of the contract will still include unclaimed rewards and the staked amount (since the reward token will also be Kernel).
Recommendations	Consider calculating the reward based on the received reward amount.
Comments / Resolution	**Comment Team 2:** Failed resolution. The newly introduced rewardsBalance introduces the following issue: H-1 Follow the recommendation: calculate the reward based on the received reward amount, by sending it in the notifyRewardAmount function. **Comment Team 1:** The new calculation mechanism is more complex than it needs to and introduces serious issues: 1. Unclaimed tokens are still not properly considered - the newly introduced rewardsBalance() variable does not properly track the token balances that are distributed/unclaimed and leads to the following issue:

	- Kernel Team makes initial deposit of 1000 tokens into the pool and decides to distribute 200 of them, which causes the following state transition: + _amount = 200 + rewardsBalance = 0 + freeRewardTokens = 1000 - 0 = 1000 + _amount < freeRewardTokens => rewardsBalance = _amount => rewardsBalance = 200 - The pool had 2 stakers which get distributed the rewards evenly - 100 each - The distribution period ends, but the stakers have not claimed their rewards yet - NotifyRewardAmount() is called again to distribute 200 more tokens + _amount = 200 + rewardsBalance = 200 + freeRewardTokens = 1000 - 200 = 800 + _amount < freeRewardTokens => rewardsBalance = _amount => rewardsBalance = 200 - Distribution period end, distributing 100 tokens more to each account - total distributed 400 (2x200) - NotifyRewardAmount() is called a 3rd time for 700 tokens: + _amount = 700 + rewardsBalance = 200 + freeRewardTokens = 1000 - 200 = 800 + _amount < freeRewardTokens => rewardsBalance = _amount => rewardsBalance = 700 - As you can see 700 token would be allowed, although it is in reality 100 tokens more than what is left undistributed (1000-400) - Problem is that rewardsBalance only considers the last value of _amount and not actually the total accumulated 2. Another issue is that claiming rewards can be DOSed:

- The following line has been added to getReward():

if (rewardAmount > rewardsBalance) revert RewardAmountGreaterThanBalance();

- The accumulated rewards should not be greater than the current value of rewardsBalance. This leads to the following scenario:
+ Kernel team Initially distributes 800 tokens, which get split between 2 stakers (400 each)
+ After the current period ends, 200 more tokens are distributed, which updated the value of rewardsBalance to 200 from 800
+ When one of the 2 accounts decides to withdraw it will revert, since 400 > 200

Overall tracking the balances is a bit tricky, which is why Syntetix implementation is simpler, but also safer.

The above validation should be reduced in complexity and it is better to rely on actual amounts than on mathematical assumptions. This would reduce complexity and the attack surface:

- Regarding the amount validation, consider using safeTransfer() to deposit the assets into the pool, this would ensure that the input _amount is precise.
- Regarding the rewardRate validation - it would be much safer and reliable not to introduce new state variables and operations that can lead to more issues than benefits. The reward distribution mechanism of Syntetix is solid enough to handle proper distribution.
- Also remove the rewardBalance check in getReward()

Comment Supervisor:

I highly recommend simplifying the codebase, reverting the change

	and choosing the original suggested fix by simply executing a transferFrom, execute accounting before and after to fetch the balance and distribute the received balance. Every other approach is most likely to fail which has been historically proven.
	Resolution 2: This has been resolved.

Issue_08	Initial staking rewards are left unclaimed
Severity	**Medium**
Description	The DepositPool is a staking fork of Synthetix and contains a known vulnerability that leaves a portion of the rewards undistributed in the initial stages of staking.

It occurs due to the notifyRewardAmount function starting off the period with an end determined by the time at which the rewards were added and not when staking actually begins:

```
function notifyRewardAmount(uint256 _amount)
        external
        onlyRole(DEFAULT_ADMIN_ROLE)
       updateReward(address(0))  {
      if (_amount > rewardsToken.balanceOf(address(this))) revert
RewardAmountGreaterThanBalance();

      if (block.timestamp >= finishAt) {
          rewardRate = _amount / duration;
      } else {
          uint256 remaining = (finishAt - block.timestamp) * rewardRate;
          rewardRate = (_amount + remaining) / duration;
      }

      if (rewardRate == 0) revert RewardRateZero();
```

	*if (rewardRate * duration > rewardsToken.balanceOf(address(this))) revert RewardAmountGreaterThanBalance();*
	
	finishAt = block.timestamp + duration;
	updatedAt = block.timestamp;
	
	emit NotifyRewardAmount(_amount, finishAt);
	}
	
	This leads to the scenario where if we notify rewards at timestamp Y, but the first stake occurs at timestamp X, the rewards for the period X - Y will be left undistributed to anyone. These rewards will not be accounted for as remaining by the contract logic and would have to be manually recirculated in the next period.
	This will occur every time a complete withdrawal of stakes occurs. Also, the Merkle contract points to the possibility of having multiple deposit pools, for which this will occur every time a new pool is freshly created.
	
	While it is true that due to how notifyRewardAmount, these rewards could be “retrieved” it would first require the knowledge that these weren’t actually distributed (which may be hard b/c the contract always contains some unclaimed rewards) and at the same time it requires careful calculation of how much is “stuck”.

Recommendations	Consider implementing a governance function in order to retrieve the undistributed rewards. This can then be called. if the contract is not actively used anymore.
Comments / Resolution	**Comment Team 2:** Failed resolution. The stuck amount still cannot be claimed since they are a part of the rewardsBalance. **Comment Team 1:**

	Due to the problems of how rewardsBalance accounts for rewards (explained in the previous issue), the newly introduced recoverExtraRewards() function also does not track properly the non-distributed rewards. Currently it can withdraw non-claimed but properly distributed rewards of stakers. Strive for a simpler solution based on actual amounts. Create a new global accumulator called nonDistributedRewards and then when calling updateReward() calculate the total accumulated rewards for the period and in case totalKernelStaked = 0 then increment nonDistributedRewards with that amount. Use the value of nonDistributedRewards in recoverExtraRewards(). **Comment Supervisor:** Change in reward distribution must be reverted and simplified by the initial recommended fix. **Resolution 2:** Partially resolved. Still if there are periods during the distribution with 0 stakes, the rewards will be stuck in the contract.

Issue_09	Event are emitted even when no rewards have been claimed
Severity	**Informational**

Description	When getReward() is called the following check is executed:
	
	if (reward > 0) {
	        rewards[msg.sender] = 0;
	        rewardsToken.safeTransfer(msg.sender, reward);
	 }
	emit RewardsClaimed(msg.sender, reward);
	
	As one can see the event is emitted outside the *reward > 0* if clause, meaning that everytime the function is called an useless event would be emitted

Recommendations	Move the event emitting logic inside the if clause so that no events are emitted for 0 reward claims
Comments / Resolution	Resolved.

Issue_10	Insufficient validation
Severity	**Informational**
Description	Inside setRewardsDuration() functions check if the provided _duration parameter is not 0. Since such value would lead to division by 0 errors in notifyRewardAmount()
Recommendations	Consider implementing the above mentioned recommendation
Comments / Resolution	Resolved.

Issue_11	Transfer-tax incompatibility
Severity	**Informational**
Description	This contract is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting.
Recommendations	Consider not using these tokens.
Comments / Resolution	Acknowledged.