PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Ghast Protocol

20 June 2023

paladinsec.co

info@paladinsec.co

Table of Contents

Disclaimer

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

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

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

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

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

1 Overview

This report has been prepared for Ghast Protocol's gstLend contract 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	Ghast Protocol
URL	TBC
Platform	Arbitrum
Language	Solidity
Preliminary Contracts	https://github.com/Jun1on/gstLend/tree/7222a62cf2b6dab031d8a1bd0b3c5ce4e7a4e6e4
Resolution 1	https://github.com/Jun1on/gstLend/blob/dadcee47f2e3735bd8d555bdfd4f9db681b31065/contracts/GHALend.sol
Resolution 2	https://github.com/Jun1on/gstLend/blob/9af3b921c047c3d14657e21783fcae9521d19047/contracts/GHALend.sol

1.2 Contracts Assessed

Name	Contract	Live Code Match
`gstLend`		

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Failed	Acknowledged (no change made)
High	7	4	1	1	1
Medium	3	2	-	-	1
Low	11	8	-	-	3
Informational	6	5	-	-	1
Total	**27**	**19**	**1**	**1**	**6**

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 gstLend

ID	Severity	Summary	Status
01	HIGH	Withdrawals send out USDC twice	✓ RESOLVED
02	HIGH	Users can profit from their own borrowing position	✓ RESOLVED
03	HIGH	`setRewards` does not set `updatedAt`	✓ RESOLVED
04	HIGH	Owner can liquidate all users	✓ RESOLVED
05	HIGH	Owner can steal users' USDC	FAILED
06	HIGH	Contract owner can DoS different functionalities	PARTIAL
07	MEDIUM	Collateral value can be manipulated	ACKNOWLEDGED
08	MEDIUM	Change of `esRatio` can result in DoS of getReward	✓ RESOLVED
09	MEDIUM	`xamount` lacks a zero check	✓ RESOLVED
10	LOW	Potential leftover amount when all is repaid/withdrawn	ACKNOWLEDGED
11	LOW	Excess borrowing is possible	✓ RESOLVED
12	LOW	Missing documentation for `borrowAPR`	ACKNOWLEDGED
13	LOW	Ultra low borrow rate can result in griefing/loss of funds	ACKNOWLEDGED
14	LOW	`DepositCap` is incorrectly validated	✓ RESOLVED
15	LOW	USDC conversion might round down	✓ RESOLVED
16	LOW	Users can borrow up to LTV	✓ RESOLVED
17	LOW	`borrowAPR` displays base for `totalBorrows = 0`	✓ RESOLVED
18	LOW	Lack of validation for address during liquidation	✓ RESOLVED
19	INFO	Checks-effects-interactions pattern is not adhered to	PARTIAL
20	INFO	Unnecessary logic for LTV = 0	✓ RESOLVED
21	INFO	Insufficient spec on interest rate calculation	✓ RESOLVED
22	INFO	Lack of `safeTransfer` usage	✓ RESOLVED
23	INFO	Typographical issues	✓ RESOLVED

1.3.2 Scope Extension

ID	Severity	Summary	Status
24	HIGH	Owner has full privileges over liquidation and funds	ACKNOWLEDGED
25	LOW	Users can liquidate their own positions	✓ RESOLVED
26	LOW	Fee change should accrue rewards	✓ RESOLVED
27	INFO	No zero value check in `redeem`	✓ RESOLVED

2 Findings

2.1 gstLend

`gstLend` is a liquidation-free lending and staking protocol that uses the Synthetix reward logic for staking rewards.

Users can deposit USDC in order to a) receive funding fees from the lending logic and b) receive `esGHA` and `GHA` rewards. `gmdUSDC` can also be deposted as collateral to borrow USDC.

After a user successfully deposits `gmdUSDC`, this position can then be used to borrow USDC with an LTV of up to 80%. The funding fee depends on the current `borrowAPR` which increases with the `borrowRate` and is highly dependent on the correct parameters which can be set in `updateInterestRateCurve`. It is capped at 1.25e17 if we assume poolID = 0 in `gmdVault` is used.

Each time a USDC deposit and withdrawal happens, the funding fee is updated which automatically transfers a share of the fee to the treasury address. It must be considered, if no repayment happens, there is no actual accumulated USDC as it is necessary to collect the fee during a user's repayment. Currently the share is 25% but can be raised up to 100% of the calculated funding fee.

Unfortunately, the developer did not execute the tests we required from them after the first audit. Therefore we do not consider this project as safe until a there is a test coverage of at least 90%. Test-coverage should include several borrow/lending participants.

2.1.1 Privileged Functions

- `updateMaxRate`
- `setDepositCaps`
- `updateInterestRateCurve`
- `changeFees`
- `changeEsRatio`
- `governanceEmergencyLiquidate`
- `governanceRecoverUnsupported`
- `setRewards`

2.1.2 Issues & Recommendations

Issue #01	Withdrawals send out USDC twice
Severity	HIGH SEVERITY
Description	The `withdraw` function transfers out USDC twice — this is a clear sign of an untested codebase and therefore we do not deem this codebase as safe for mainnet deployment until 90% test coverage.
Recommendation	Consider removing the second transfer and test the codebase thoroughly.
Resolution	RESOLVED The additional transfer has been removed.

Issue #02	Users can profit from their own borrowing position
Severity	HIGH SEVERITY
Description	It is possible for a user to build a huge borrow position. This position will then accumulate funding fees that *should* be paid, however, they are also claimable if they have not been repaid yet, and USDC lenders will bear that amount. This can be abused by the same user to build up a huge lending position in an effort to receive funding fees. All this can be done until the user is in profit (w.r.t 80% LTV) and potentially way longer. To combat this issue, a proper liquidation function is necessary. Unfortunately, the current liquidation function does not repay the borrowed amount which results in fees still being accumulated from this "dead" position. Moreover it seems hard for the contract owner to identify such a behavior since there is no functionality to display a user's HF. In practice, it might be hard to identify such a behavior.
Recommendation	Consider implementing functionality to easily determine such a HF, and also consider improving the liquidation functionality to in fact repay the borrowed amount.
Resolution	RESOLVED Users can now liquidate other users once their LTV increases above `redLTV`.

Issue #03	`setRewards` does not set `updatedAt`
Severity	HIGH SEVERITY
Description	`setRewards` does not set `updatedAt` to the current `block.timestamp`. The `updateReward` modifier is not updated either since `lastTimeRewardApplicable` is 0 at this point. Therefore, the first depositor will receive rewards granted from 0 to `block.timestamp` which will effectively break the contract since this amount is mostly not covered. Moreover, this will also distribute rewards retroactively if a reward period got finalized and a new one is initiated.
Recommendation	Consider setting `updatedAt` during `setRewards`. https://github.com/Synthetixio/synthetix/blob/develop/contracts/StakingRewards.sol#L129
Resolution	RESOLVED

Issue #04	Owner can liquidate all users
Severity	HIGH SEVERITY
Description	The owner has the freedom to simply liquidate any position. This is a huge governance issue and should not be part of an immutable contract.
Recommendation	Consider changing the functionality to allow all users to liquidate a position when it is under a specific HF. At least consider implementing a proper multi-signature set-up with known participants as the owner.
Resolution	RESOLVED The function has been removed.

Issue #05	Owner can steal users' USDC
Severity	HIGH SEVERITY
Description	There are several ways in which the owner can steal users' USDC. The first way is similar to issue 1, however, this is even simpler since the owner cannot get liquidated and will therefore be able to build a huge borrow position and accumulate fees from their own "dead" position. The owner can additionally manipulate the `depositCap` to such an amount that ensures that he has the majority of USDC deposits. The second way is the easiest way: the owner can simply liquidate users and use these positions to borrow USDC. This is possible because the owner can liquidate all positions, not only those exceeding the LTV. A third, more sophisticated way is possible by changing the `maxRate` which can potentially allow the owner to manipulate the collateral value using the external vault contract.
Recommendation	Consider implementing a liquidation functionality that is publicly callable when a specific HF is reached. Consider using a trusted multi-signature wallet as the owner. Consider implementing an upper limit for `maxRate`.
Resolution	FAILED RESOLUTION The owner can now withdraw all tokens using the `yoink` function.

Issue #06	Contract owner can DoS different functionalities
Severity	HIGH SEVERITY
Description	The treasury address can be freely set to any address including `address(0)`. This will result in various functions reverting, including locking all user funds. The owner can use `setReward` to set an insanely high `_rewardRate` with `_finishAt` to the current timestamp. This will then also result in a revert in various functionalities as well. As a third issue, the owner can steal all reward tokens via the recover function which will DoS `getReward`.
Recommendation	Consider validating the treasury address appropriately. Consider executing a check that `_ rewardRate` is a reasonable value as well as ensure `_finishAt` is in the future. Consider validating that no reward tokens are can be withdrawn as well.
Resolution	PARTIALLY RESOLVED The treasury address is now validated.

Issue #07	Collateral value can be manipulated
Severity	MEDIUM SEVERITY

Description	Currently, the collateral value is determined as follows:
	
	`uint256 totalBorrowable = (valueOfDeposits(msg.sender) * LTV) / MAX_BPS;`
	
	`function valueOfDeposits(address _user) internal view returns (uint256) {`
	`    return (gmdDeposits[_user] * usdPerGmdUSDC()) / 1e18;`
	`}`
	
	As one can see from the code snippet above, `usdPerGmdUSDC` determines the value of the user's deposit.
	
	`function usdPerGmdUSDC() internal view returns (uint256) {`
	`    uint256 totalShares = gmdUSDC.totalSupply();`
	`    uint256 calculatedRate = GMDVault.poolInfo(poolId).totalStaked * 1e18 / totalShares;`
	`    return _min(calculatedRate, maxRate);`
	`}`
	
	In an effort to mitigate a potential manipulation which can occur due to the usage of a third party contract, a `maxRate` was implemented which sets the maximum return value to 1.1e18.
	
	However, there is still potential space to manipulate the `valueOfDeposits`, for example when the current `usdPerGmdUSDC` is 0.1e18, it can be still be increased by x11 to 1.1e18, effectively manipulating the collateral worth by a lot.

Recommendation	A fix is non-trivial since the third-party contract is out of scope. We highly recommend keeping this in mind. It might also make sense to include the third-party contract into the audit in an effort to find a possible solution for this problem.
Resolution	ACKNOWLEDGED This issue is even more present since it is now potentially possible to manipulate the collateral value and liquidate users.

Issue #08	Change of `esRatio` can result in DoS of `getReward`
Severity	MEDIUM SEVERITY
Description	Whenever new rewards are being set, the contract ensures the correct balances are maintained. However, if `esRatio` is changed afterwards, the contract might run out of funds for the set reward period.
Recommendation	Consider only allowing `esRatio` to be adjusted within `setRewards`.
Resolution	RESOLVED `esRatio` is now only determined within `setRewards`. This ensures that the balances are sufficient.

Issue #09	`xamount` lacks a zero check
Severity	MEDIUM SEVERITY
Description	During a deposit the following calculation is executed: `uint256 xamount = (_amount * decimalAdj * 1e18) / usdPerxDeposit();` `xamount` represents the user's position. If `usdPerXDeposit` becomes very high, this calculation will round down, thus `xamount` can become zero. This issue is more theoretical than practical as an incredible amount of funding fees needs to be accumulated with a very low overall deposited amount which will most likely not happen during the normal business logic. Since the fix is very trivial, we encourage the team to fix this potential edge-case.
Recommendation	Consider executing a check that `xamount` cannot be zero.
Resolution	RESOLVED

Issue #10	Potential leftover amount when all is repaid/withdrawn
Severity	LOW SEVERITY

Description	There is an edge-case where `totalBorrows` will have a leftover amount due to rounding.
	
	Consider the following scenario:
	
	A. Alice borrows 100 USDC
	B. 111111111 WEI USDC will be accrued
	C. Alice repays her whole debt, in total 111.1... USDC
	D. `totalXBorrows` now becomes zero
	E. However, `totalBorrows` will have a leftover amount
	
	This is due to the rounding down in the following line:
	
	`uint256 borrows = (xborrows[msg.sender] * usdPerxBorrow()) / 1e18;`
	
	which is then deducted here:
	
	`totalBorrows -= borrows;`
	
	Effectively leaving a leftover amount.
	
	While we could not find a way to exploit this with consecutive calls, there still might be an edge-case where this can result in a problem.
	
	*The same issue applies to the `deposit` / `withdraw` logic, where `totalDeposits` can have a leftover amount.

Recommendation	The fix is non-trivial. The team could implement logic which sets `totalBorrows` to zero once it is < 100 WEI. This might be possible due to the general adjustment to a minimum of 12 decimals however, we recommend executing extensive tests to mitigate this issue.
Resolution	ACKNOWLEDGED

Issue #11	Excess borrowing is possible
Severity	LOW SEVERITY
Description	There is no check if the total borrowed amount exceeds the total deposited amount. While that is prevented in the normal business logic because the contract does not have any funds left to transfer out, a sophisticated user can transfer USDC directly to the contract and continue borrowing, thus manipulating the `utilizationRatio`. This would then artificially increase the `borrowAPR` to something which was potentially not thought out by the team.
Recommendation	Consider if this can become an issue, and if so, consider implementing such a check to make this manipulation impossible.
Resolution	RESOLVED A `borrowCap` has been added.

Issue #12	Missing documentation for `borrowAPR`
Severity	LOW SEVERITY

Description	`borrowAPR` is calculated based on the borrowed amount and the interest rate variables. However, proper documentation on what is desired is missing. Therefore, we will reproduce one example to show the `borrowAPR` for one specific case, and we assume the borrow variables which are set as standard:
	
	`totalDeposits = 1_000_000e18`
	`totalBorrows = 1_000e18`
	`utilizationRatio = 1e15`
	`calculated borrowAPR: 2.01e16`
	
	Based on these variables, the reward per second is 6.37e11, thus per day is 55068493150627200 converted, resulting in 55068 nominal token whereas 1e6 = 1 USDC.
	
	Therefore, the current `borrowApr` = 2.01e16 represents 2.01% APR.
	
	However, for example, with a `totalDeposits` of 110000 and `totalBorrows` of 100000, `borrowAPR` returns 1.25e17 (assuming e.g 1000 APR in third-party contract). However, the APR is in fact lower than 12.5% because 110000 tokens are deposited, not 100000.

Recommendation	Consider if it is the desired outcome to have this `borrowAPR` for these circumstances, and consider adding extensive tests for this logic as well as for edge-cases and consider providing proper documentation for this logic.
Resolution	ACKNOWLEDGED The team stated that this logic works as intended.

Issue #13	Ultra low borrow rate can result in griefing/loss of funds
Severity	LOW SEVERITY

Description	The following function sends tokens to the treasury:
	`function accureInterest() internal {`
	`uint256 reward = pendingInterest();`
	`lastUpdate = block.timestamp;`
	`uint256 fees = reward * feeRate / MAX_BPS;`
	`USDC.transfer(treasury, fees / decimalAdj);`
	`totalDeposits += (reward - fees);`
	`totalBorrows += reward;`
	`}`
	If the borrow rate is very low, fees can be smaller than `decimalAdj`, which results in a zero transfer to the treasury.
	A malicious user/bot could abuse this on networks with very low gas fees.
	The same issue is also present for USDC withdrawals.

Recommendation	Consider if this will become an issue for the project and consider thinking about the borrow rate in the example above.
Resolution	ACKNOWLEDGED

Issue #14	`DepositCap` is incorrectly validated
Severity	LOW SEVERITY
Description	The current logic for the `depositCap` is flawed as `_amount` is in 1e6 while the other variables are in 1e18.
Recommendation	Consider bringing all variables to the same decimals and then executing a check.
Resolution	RESOLVED

Issue #15	USDC conversion might round down
Severity	LOW SEVERITY
Description	Within the `repay` function, the following calculation is done: `USDC.transferFrom(msg.sender, address(this), borrows / decimalAdj);` This will round down whenever borrows is < 1e12. While we could not identify a way to exploit this because the smallest possible borrowable position is 1WEI which translates to 1e12, it should still be considered.
Recommendation	Consider rounding it up.
Resolution	RESOLVED

Issue #16	Users can borrow up to LTV
Severity	LOW SEVERITY

Description	Typically it should not be possible for a user to borrow until the HF threshold is reached. Below is an example in AAVE V3:
	
	`require( vars.healthFactor > HEALTH_FACTOR_LIQUIDATION_THRESHOLD, Errors.HEALTH_FACTOR_LOWER_THAN_LIQUIDATION_THRESHOLD );`
	
	As seen above, the `healthFactor` should never cross the liquidation threshold. Unfortunately, this is possible here:
	
	`require( _amount * decimalAdj <= borrowable, "GHALend: Insufficient collateral" );`

Recommendation	Consider validating that the amount is in fact lower than borrowable.
Resolution	RESOLVED

Issue #17	`borrowAPR` displays base for `totalBorrows = 0`
Severity	LOW SEVERITY
Description	`borrowAPR` displays the base number when no borrows are made. While this is not an issue in the business logic since it will be zero anyway, it can confuse users as a view-function.
Recommendation	Consider also returning 0 when no borrows are made.
Resolution	RESOLVED

Issue #18	Lack of validation for address during liquidation
Severity	LOW SEVERITY
Description	There is no check that the owner will not liquidate themselves. This would effectively lock the owner's whole liquidated positions as well as result in discrepancies with `totalxBorrows`.
Recommendation	Consider executing such a check.
Resolution	RESOLVED This function has been removed.

Issue #19	Checks-effects-interactions pattern is not respected
Severity	INFORMATIONAL

Location	L97
	`USDC.transferFrom(msg.sender, address(this), _amount);`
	
	L131
	`gmdUSDC.transferFrom(msg.sender, address(this), _amount);`
	
	L234
	`USDC.transfer(treasury, fees / decimalAdj);`

Description	The contract contains spots where the checks-effects-interactions pattern is not respected. While this not exposes any risk since reentrancy checks are present it is still considered best-practice to follows this pattern.
Recommendation	Consider following the checks-effects-interactions pattern.
Resolution	PARTIALLY RESOLVED `deposit` and `depositGMD` functions do not adhere to CEI.

Issue #20	Unnecessary logic for LTV = 0
Severity	INFORMATIONAL

Description	The LTV value can never be changed, thus any logic for `LTV=0` is unnecessary.
	`if (LTV == 0) { require( xborrows[msg.sender] == 0, "GHALend: Insufficient collateral" ); freeUSD = valueOfDeposits(msg.sender);`
	
	`if (LTV == 0) { return APR; }`

Recommendation	Consider removing the unnecessary logic.
Resolution	RESOLVED

Issue #21	Insufficient spec on interest rate calculation
Severity	INFORMATIONAL

Description	When calculating the `utilizationRatio` within `borrowAPR()` and the `earnRateSec` in `updateAPR()`, the `totalDeposits` and `totalBorrows` variables are used.
	
	`earnRateSec = totalBorrows*borrowAPR()/1e18/(365 days);`
	`uint256 utilizationRatio = totalBorrows*1e18/totalDeposits;`
	
	Since both `totalBorrows`, `totalDeposits` and `totalxBorrows`, `totalxDeposits` are used to account for the deposits and borrows but the first are incremented by the rewards, it is unclear which exactly should be used for the calculation.

Recommendation	Consider thinking about the consequences of using `totalDeposits` and `totalBorrows` versus `totalxDeposits` and `totalxBorrows`. Consider adding extensive tests for this logic as well as for edge-cases and consider providing proper documentation for this logic.
Resolution	RESOLVED The following comment has been added: *`// function is always called after accrueInterest() so totalBorrows refers to the true value`*

Issue #22	Lack of `safeTransfer` usage
Severity	INFORMATIONAL
Description	The contract is not using `safeTransfer/From`. This will not work with non-compliant tokens and tokens that return `false` on transfers. While we acknowledge that the tokens are in fact known, we still encourage the use of the safeERC20 library as a best practice.
Recommendation	Consider using `safeTransfer/From`.
Resolution	RESOLVED

Issue #23	Typographical issues
Severity	INFORMATIONAL

Description	L34
	`IGMDVault public GMDVault = IGMDVault(0x8080B5cE6dfb49a6B86370d6982B3e2A86FBBb08);`
	This can be constant.
	
	L35
	`uint256 public poolId;`
	This can be constant and hardcoded to PID 0.
	
	L36-37
	`IERC20 public gmdUSDC;`
	`IERC20 public USDC;`
	These variables can be immutable.
	
	L56
	`uint256 public LTV = 0.8 * 1e4;`
	This can be constant.

Recommendation	Consider fixing the typographical issues.
Resolution	RESOLVED

2.2 Scope Extension

The Ghast team implemented a liquidation mechanism that allows users to liquidate the positions of other users once the LTV raises above a specific `redLTV`.

The incentive for this liquidation mechanism is that liquidators will receive `gmdUSDC` for the exact USDC value whereas the converting mechanism would take a 0.5% fee on this transaction. There is no penalty applied on a user's collateral.

2.2.1 Issues & Recommendations

Issue #24	Owner has full privileges over liquidation and funds
Severity	HIGH SEVERITY

Description	The owner currently has the privilege to change the loan-to-value variables, `LTV` and `redLTV`, using the `updateLTVs` function:
	
	`function updateLTVs(uint256 _LTV, uint256 _redLTV) external onlyOwner {`
	`  require(_LTV <= MAX_BPS, "out of range");`
	`  require(redLTV <= MAX_BPS, "out of range");`
	`  require(redLTV > _LTV, "too low");`
	`  LTV = _LTV;`
	`  redLTV = _redLTV;`
	`}`
	
	Setting LTV to 0 is possible and also handled in multiple places around the contract. However, this also means that `redLTV` can be set as low as 1 at any point in time by the owner.
	
	This can be abused to almost fully liquidate any user since the following check in redeem will be easily passed:
	
	`require(`
	`  userLTV(_user) > (redLTV * 1e18) / MAX_BPS,`
	`  "GHALend: too much redemption"`
	`);`
	
	The owner can steal all tokens via the `yoink` function.

Recommendation	A fix is non-trivial with the current implementation. Consider using a trusted multi-signature wallet as the owner and timelocking the function.
Resolution	ACKNOWLEDGED

Issue #25	Users can liquidate their own positions
Severity	LOW SEVERITY
Description	For complex lending protocols, it is important to reduce the potential state transitions as much as possible. Since there is no reason why a user would want to liquidate their own position, they should not be allowed to do so.
Recommendation	Consider implementing a check that prevents a user from liquidating their own position.
Resolution	RESOLVED

Issue #26	Fee change should accrue rewards
Severity	LOW SEVERITY
Description	The `changeFees` function changes the fee percentage which is allocated to the treasury. Since there is no update beforehand, it will change the allocation retroactively.
Recommendation	Consider calling `accrueRewards` and `updateAPR` before the fees are adjusted.
Resolution	RESOLVED

Issue #27	No zero value check in `redeem`
Severity	INFORMATIONAL
Description	The `_amount` parameter in the redeem function is an arbitrary uint256 value passed by users. If 0 is passed as `_amount`, the function will execute without performing any liquidation-related logic but will still emit the Redeem event which may cause problems off-chain.
Recommendation	Consider a zero value check for the `_amount` parameter in redeem.
Resolution	RESOLVED


PALADIN
BLOCKCHAIN SECURITY