PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Galaxy Goggle DAO

17 January 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 Galaxy Goggle DAO on the Binance Smart Chain. 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	Galaxy Goggle
URL	https://app.galaxygoggle.money/#/
Platform	Binance Smart Chain
Language	Solidity

1.2 Contracts Assessed

Note: The client forked their contracts from Wonderland Money but did not rename the contract names, and thus still reference the names from the Wonderland protocol. The contract names are included as-is below with the Galaxy Goggle equivalent in parenthesis.

Name	Contract	Live Code Match
TimeERC20Token (Galaxy Goggle Token (GG))	0xcAf23964Ca8db16D816eB314a56789F58fE0e10e	MATCH
MEMOries (Staked Galaxy Goggle Token (sGG))	0x98868ebA22Ce858a19d67E984211fa80aB74DF8B	MATCH
wsGG	0xb18b51dE676f466e3758f9e3B0739CBa564c7209	MATCH
TimeBondDepository	BUSD: 0x4d60C0b9eC47440e412A8080cc040A9364E1EAb4 BUSD/GG LP: 0x43EA89C2Ea35bfb44536623A2a0d6b54729258B3 CAKE: 0xa1fdCb89c02B7cF17676ce376cfdCa841Ae5D840	MATCH
EthTimeBondDepository	BNB: 0x50A0DdE8734Ac2B68013825FCE6958BBbF86DAC2	MATCH
TimeStaking	0x97209Cf7a6FccC388eEfF85b35D858756f31690d	MATCH
Distributor	0xCc2e2134b74C70Acd64Bb297d18Ba25087789D8d	MATCH
StakingHelper	0x2863C5D73510c671F5D16687BDfFcc3De6f14576	MATCH
StakingWarmup	0x137D4755071844DcE186aD6df8c07e017137D3c7	MATCH
TimeBonding	0xE4677ba3fc3CBB92633e6c636F073E76F3A76a61	MATCH
TimeTreasury	0xF76C9753507B3Df0867EB02D86d07C6fFcEecaf1	MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	3	1	-	2
Medium	3	2	-	1
Low	18	3	-	15
Informational	32	1	-	31
Total	**56**	**7**	**-**	**49**

Classification of Issues

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

1.3.1 Global Issues

ID	Severity	Summary	Status
01	HIGH	Gov Privilege: Governance can change crucial aspects of the protocol to potentially drain the contracts of all supplied tokens	RESOLVED
02	MEDIUM	The last owner can be reclaimed across multiple contracts	ACKNOWLEDGED
03	LOW	New owner variable is private	ACKNOWLEDGED
04	INFO	`permit` can be frontrun and cause denial of service	ACKNOWLEDGED
05	INFO	The contracts become unusable in the year 2106	ACKNOWLEDGED
06	INFO	The contracts do not work with tax on transfer tokens	ACKNOWLEDGED

1.3.2 TimeERC20Token (GalaxyGoggle Token [GG])

ID	Severity	Summary	Status
07	LOW	`_burnFrom` is marked as `public`	ACKNOWLEDGED
08	INFO	Transactionless approval permit mechanism still references zero swap	ACKNOWLEDGED
09	INFO	Vault can be set to the zero address	ACKNOWLEDGED

1.3.3 MEMOries (Staked GalaxyGoggle)

ID	Severity	Summary	Status
10	LOW	Infrequent rebases incentivize malicious parties to strategically (re)order transactions for arbitrage and steal all rebased tokens off the LP pairs	ACKNOWLEDGED
11	INFO	`DOMAIN_SEPARATOR` can be made immutable	ACKNOWLEDGED
12	INFO	Under a constant and small circulating supply, the non-circulating supply starts increasing more rapidly with every rebase	ACKNOWLEDGED

1.3.4 wsGG

ID	Severity	Summary	Status
13	INFO	Gas optimization: Usage of decimals as a precision multiplier	ACKNOWLEDGED

1.3.5 TimeBondDepository

ID	Severity	Summary	Status
14	HIGH	Vested amount is relocked on deposit, even if the deposit is made by third-parties allowing for targeted Denial of Service	ACKNOWLEDGED
15	MEDIUM	The maximum debt can be exceeded by at most `maxPayout`	RESOLVED
16	LOW	Adjustment target is never reached	ACKNOWLEDGED
17	LOW	`deposit` is vulnerable to reentrancy if the principle has a reentrancy vector	ACKNOWLEDGED
18	INFO	`bondPriceInUSD` is denominated in the decimals of the other token in the LP and might not be correct for non-stablecoin LPs	ACKNOWLEDGED
19	INFO	`initializeBondTerms` has no validation	ACKNOWLEDGED
20	INFO	Contract does not work with a zero `vestingTerm`	ACKNOWLEDGED
21	INFO	Contract could theoretically run out of GG	ACKNOWLEDGED

1.3.6 EthTimeBondDepository

ID	Severity	Summary	Status
22	LOW	`priceFeed` is internal	ACKNOWLEDGED
23	INFO	Phishing: Frontend could trick users into sending too much BNB for deposits	ACKNOWLEDGED
24	INFO	`bondPrice` calculation is inconsistent with TimeBondDepository	ACKNOWLEDGED
25	INFO	The protocol will likely malfunction if ChainLink feeds for BNB will be different than 8 decimals	ACKNOWLEDGED

1.3.7 TimeStaking

ID	Severity	Summary	Status
26	HIGH	Staked amount is relocked on subsequent stakes, even if the stake is made by third-parties allowing for targeted Denial of Service	ACKNOWLEDGED
27	LOW	Lock logic might be wrongly defined	RESOLVED
28	INFO	Rebases can be arbitraged/frontran	ACKNOWLEDGED

1.3.8 Distributor

ID	Severity	Summary	Status
29	LOW	Adjustment target is never reached	ACKNOWLEDGED
30	LOW	Unbounded gas usage due to extensive for-loop usage	ACKNOWLEDGED
31	LOW	Lack of validation	ACKNOWLEDGED
32	INFO	Adjustments are not reset when recipient is removed	ACKNOWLEDGED

1.3.9 StakingHelper

ID	Severity	Summary	Status
33	LOW	Phishing: `stake` function allows to stake towards a different address	ACKNOWLEDGED

1.3.10 StakingWarmup

No issues found.

1.3.11 TimeBondingCalculator

ID	Severity	Summary	Status
34	MEDIUM	Does not support LP pairs where the second currency has less than 9 decimals	RESOLVED
35	LOW	TimeBondingCalculator can only value pairs in which the two tokens have equal "value"	ACKNOWLEDGED
36	LOW	Markdown will misbehave if an LP with two non-GG tokens is provided	ACKNOWLEDGED
37	INFO	UI `markdown` function is vulnerable to price manipulation	ACKNOWLEDGED

1.3.12 TimeTreasury

ID	Severity	Summary	Status
38	LOW	Treasury does not have sGG set as a contract	RESOLVED
39	LOW	`incurDebt` can be used to drain the TimeTreasury from ReserveToken	RESOLVED
40	LOW	Lack of component safeguards in a system that plans to increase in number of components over time is considered brittle	ACKNOWLEDGED
41	LOW	Adding a token as both a liquidity and reserve token would cause it to be double counted in the treasury value	ACKNOWLEDGED
42	LOW	`repayDebtWithOhm` has inconsistent privilege requirements which allows for slight privilege escalation	ACKNOWLEDGED
43	INFO	Unnecessary comparison to `true` on `withdraw` function	ACKNOWLEDGED
44	INFO	Reserve value mechanism could cause withdrawals and other operations to temporarily fail	ACKNOWLEDGED
45	INFO	Lack of `safeTransfer` usage within `incurDebt`	ACKNOWLEDGED
46	INFO	Manage will always do an `excessReserve` check even if the token is not within the liquidity or reserves tokens	ACKNOWLEDGED

1.3.13 Code style-related issues

ID	Severity	Summary	Status
47	INFO	Inconsistency: Unused `mint` function emits an event from address(this) while the mint logic during initialization emits an event from the zero address	ACKNOWLEDGED
48	INFO	Various functions can be made external	ACKNOWLEDGED
49	INFO	Lack of events for various functions	ACKNOWLEDGED
50	INFO	Typographical errors	ACKNOWLEDGED
51	INFO	Unused variables/dependencies throughout the contracts	ACKNOWLEDGED
52	INFO	Gas optimization: Contract uses hardcoded strings in SafeMath functions	ACKNOWLEDGED
53	INFO	Uncast addresses make the code more verbose than it needs to be	ACKNOWLEDGED
54	INFO	Ambiguous errors	ACKNOWLEDGED
55	INFO	Gas optimization: Storage variables are frequently unnecessarily reread	ACKNOWLEDGED

1.3.14 Inapplicable deployment Issues

ID	Severity	Summary	Status
56	INFO	Inapplicable deployment related issues	RESOLVED

2 Findings

2.1 Global Issues

The issues in this section are applicable to the entire protocol.

2.1.1 Issues & Recommendations

Issue #01	**Gov Privilege: Governance can change crucial aspects of the protocol to potentially drain the contracts of all supplied tokens**
Severity	HIGH SEVERITY
Description	GalaxyGoggle is a protocol that is responsible for the issuance and management of an algorithmic, free-floating stable asset, GalaxyGoggle, which is backed by a treasury. As the system has many components which need to be governed, like how the treasury is potentially used, which assets could be used as bonds and the very important parameters of the bond issuance protocol, there is by nature an extreme amount of governance privilege. Essentially, if governance cannot be controlled, both all GG and all funds in the treasury can be considered compromised. It is therefore of utmost importance that the team addresses this concern seriously. Some of the most important governance privileges are that the treasury manager can add new contracts that can mint any amount of GalaxyGoggle (up to the maximum allowed by the reserve value), the manager can furthermore add contracts that can potentially withdraw all funds stored in the treasury. Finally, within the GG token, "vault" ownership could be moved by the GG token owner to a new address which can then again mint as many GG tokens as they want, in this case without limit. Other potential risk vectors include depositing bad tokens into the treasury which allow privileged contracts to take out valuable assets in return and sGG tokens in the Staking contract can be taken out by governance through the lock bonus mechanism. Due to the anonymous nature of DeFi, users have become quite wary of protocols with large privileges and it will likely boost investor confidence to address this seriously.
Recommendation	Consider designing a strong governance structure where it is unlikely and ideally impossible for the governance to abuse these privileges. A decent short-term solution is doxx-ing or KYC'ing oneself to parties trusted by the community as one will be less inclined to steal funds when their identities are known.

Resolution	RESOLVED The client has introduced multi-sig wallet control on both the GG main token contract and treasury contract to ensure that 3/5 signers must sign to submit these changes referenced. This prevents one malicious actor (deployer address) from making contract changes.

Issue #02	**The last owner can be reclaimed across multiple contracts**
Severity	MEDIUM SEVERITY

Description	Within the ownership implementation of the different contracts, ownership can be renounced, however, the last owner can reclaim this at any moment as the new owner variable was never reset.
	
	It should also be noted that before the first ownership transfer is made, the zero address can claim ownership over the contract. This is hardly problematic as the zero contract is not known to be owned by anyone and probabilistically speaking, under the current address scheme, the chances of anyone ever owning it are negligible.
	
	Note: On Distributor, the Ownernership implementation is implemented as Policy.
	
	MEMOries::992-995
	`function renounceManagement() public virtual override onlyManager {`
	`emit OwnershipPushed(_owner, address(0));`
	`_owner = address(0);`
	`}`
	
	MEMOries::1003-1007
	`function pullManagement() public virtual override {`
	`require(msg.sender == _newOwner, "Ownable: must be new owner to pull");`
	`emit OwnershipPulled(_owner, _newOwner);`
	`_owner = _newOwner;`
	`}`

TimeBondDepository::40-43

```
function renounceManagement() public virtual override
onlyManager {
emit OwnershipPushed(_owner, address(0));
    _owner = address(0);
}
```

TimeBondDepository::51-55

```
function pullManagement() public virtual override {
    require(msg.sender == _newOwner, "Ownable: must be new
owner to pull");
    emit OwnershipPulled(_owner, _newOwner);
   _owner = _newOwner;
}
```

TimeStaking::538-541

```
function renounceManagement() public virtual override
onlyManager {
    emit OwnershipPushed(_owner, address(0));
    _owner = address(0);
}
```

TimeStaking::549-553

```
function pullManagement() public virtual override {
    require(msg.sender == _newOwner, "Ownable: must be new
owner to pull");
    emit OwnershipPulled(_owner, _newOwner);
   _owner = _newOwner;
}
```

Distributor::348-351

```
function renouncePolicy() public virtual override
onlyPolicy() {
    emit OwnershipTransferred( _policy, address(0) );
    _policy = address(0);
}
```

Distributor::354-358

```
function pullPolicy() public virtual override {
    require( msg.sender == _newPolicy );
    emit OwnershipTransferred( _policy, _newPolicy );
    _policy = _newPolicy;
}
```

TimeTreasury::167-170

```
function renounceManagement() public virtual override
onlyManager {
    emit OwnershipPushed(_owner, address(0));
    _owner = address(0);
}
```

TimeTreasury::178-182

```
function pullManagement() public virtual override {
    require(msg.sender == _newOwner, "Ownable: must be new
owner to pull");
    emit OwnershipPulled(_owner, _newOwner);
   _owner = _newOwner;
}
```

Recommendation Consider using BoringOwnable instead.

Resolution ACKNOWLEDGED

The client does not plan to renounce ownership of any contracts. This issue is thus a non-factor. All contracts in the protocol are owned/protected by a gnosis multi-sig wallet which can be verified on chain.

Issue #03	New owner variable is private
Severity	LOW SEVERITY
Description	Throughout the contracts that implement the Ownership pattern the variable that denotes the new owner is private. 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. - MEMOries: `_newOwner` variable - TimeBondDepository: `_newOwner` variable - TimeStaking: `_newOwner` variable - Distributor: `_newPolicy` variable - TimeTreasury: `_newOwner` variable
Recommendation	Consider marking the aforementioned variables as `public`.
Resolution	ACKNOWLEDGED

Issue #04	`permit` can be frontrun and cause denial of service
Severity	INFORMATIONAL
Description	Many of the tokens contain a transactionless approval scheme based on EIP-2612. This mechanism is most well-known by users when they break up uniswap LP tokens without having to explicitly send an approval transaction, instead they just have to make a signature. Just like with Uniswap permits, if permit is executed twice, the second execution will be reverted. 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 issue with this is that the rest of said contract functionality would be lost as well. This could allow for denial of service.
Recommendation	Within derivative protocols, one can consider using try-catch for permit and validating the approval afterwards.
Resolution	ACKNOWLEDGED

Issue #05	The contracts become unusable in the year 2106
Severity	INFORMATIONAL
Location	TimeStaking::710 (example) `epoch.endTime = epoch.endTime.add32( epoch.length );`
Description	Throughout the contract, timestamps are supposed to fit in 32 bit integers. However, once the year 2106 is reached, this is no longer possible and most functionality will fail and revert. For example, once the timestamp of the next epoch does not fit into an unsigned 32 bit integer, the rebase function will revert. Furthermore, as soon as this timestamp is reached, even if the epoch hasn't caught up yet, `rebase()` will no longer be callable on previous epochs since `uint32(block.timestamp)` overflows to a low number.
Recommendation	Consider whether the contract will survive for this long, if so, consider using a larger integer for the timestamp or adding an overflow mechanism (Uniswap for example designed their timestamps to still work with overflows).
Resolution	ACKNOWLEDGED The client stated that the TimeStaking Contract at that point will not be needed as emissions will likely be capped.

Issue #06	The contracts do not work with tax on transfer tokens
Severity	INFORMATIONAL
Description	The whole GalaxyGoggle system is completely incompatible with any transfer-tax tokens. Albeit as principle tokens, or forked versions of GG or sGG, transfer-taxes are not supported.
Recommendation	Consider avoiding any tokens with transfer-taxes, rebase mechanisms or other special logic going on. These can be wrapped in a simple wrapped equivalent that has no auxiliary transfer logic going on.
Resolution	ACKNOWLEDGED

2.2 TimeERC20 (Galaxy Goggle Token [GG])

The GG token is the main token within the GalaxyGoggle ecosystem, it is a simple ERC-20 token which is extended with EIP-2612 permit capabilities. Users undoubtedly know such permit capabilities from when they break up uniswap LP tokens. In this instance, instead of explicitly needing to transmit an approve transaction, they can simply sign it without any gas cost or transaction.

The token can be minted freely by the vault address, which is settable at any time by the contract owner.

2.2.1 Token Overview

Address	0xcAf23964Ca8db16D816eB314a56789F58fE0e10e
Token Supply	Unlimited
Decimal Places	9
Transfer Max Size	No maximum
Transfer Min Size	No minimum
Transfer Fees	None
Pre-mints	None

2.2.2 Privileged Roles

The following functions can be called by the owner of the contract:

- `mint`
- `setVault`
- `transferOwnership`
- `renounceOwnership`

2.2.3 Issues & Recommendations

Issue #07	`_burnFrom` is marked as `public`
Severity	LOW SEVERITY
Description	The contract contains a function, `burnFrom`, to allow burning GG from another account given that this account has given you approval. As is common with ERC-20 implementations, the public `burnFrom` function calls the internal function `_burnFrom`. Within this implementation however, `_burnFrom` is marked as public by accident. This does not have side-effects and therefore does not affect investors in any way. However, it might signal to third-party reviewers that the code was not carefully reviewed before deployment.
Recommendation	Consider marking the `_burnFrom` function as internal.
Resolution	ACKNOWLEDGED

Issue #08	Transactionless approval permit mechanism still references zero swap
Severity	INFORMATIONAL
Location	Line 807 `require(signer != address(0) && signer == owner, "ZeroSwapPermit: Invalid signature");`
Description	For single transaction approvals, EIP-2612-based permits are used. These are famously known by users when they break up uniswap LPs and only need to sign the approval instead of actually creating a transaction. However, within the implementation used by GalaxyGoggle, an error message still references ZeroSwap. This might signal that the codebase was not reviewed before deployment to third-party reviewers and is therefore best adjusted.
Recommendation	Consider adjusting the error message.
Resolution	ACKNOWLEDGED

Issue #09	Vault can be set to the zero address
Severity	INFORMATIONAL
Description	The `setVault` function allows for the vault, which is the only account that can mint GG, to be set to zero. It is common practice to add a non-zero function to such set functions as to prevent potential error to go undetected.
Recommendation	Consider adding non-zero checks to `setVault`.
Resolution	ACKNOWLEDGED

2.3 MEMOries (Staked Galaxy Goggle [sGG])

The Staked GalaxyGoggle (sGG) token is a rebasing token which increases the sGG supply and therefore the user balances whenever the staking contract calls rebase on it. It is kept somehow backed by MIM. It has a different approach than a normal stablecoin that is usually pegged to a certain asset.

sGG follows a similar implementation to Ampleforth and is likely inspired by this protocol.

2.3.1 Token Overview

Address	0x98868ebA22Ce858a19d67E984211fa80aB74DF8B
Token Supply	Unlimited
Decimal Places	9
Transfer Max Size	No maximum
Transfer Min Size	No minimum
Transfer Fees	None
Pre-mints	5,000,000 [to Staking contract]

2.3.2 Privileged Roles

The following functions can be called by the owner of the contract:

- `initialize [ callable once, already called ]`
- `setIndex [ callable once, already called ]`
- `rebase`
- `renounceManagement`
- `pushManagement`
- `pullManagement`

2.3.3 Issues & Recommendations

Issue #10	**Infrequent rebases incentivize malicious parties to strategically (re)order transactions for arbitrage and steal all rebased tokens off the LP pairs**
Severity	LOW SEVERITY
Description	The contract periodically increases the user balances as part of the rebases. If these rebases were to occur sufficiently infrequently, say every week, it might be an incentive for either miners or advanced users to strategically order their transactions in a way that they temporarily hold a balance right before the rebase to receive rewards on it. Furthermore, even if rebases were to be made frequently, each time a rebase occurs, the balance of the LP pairs will increase. If `skim()` is called on the LP pairs right after this occurs, the skimmer will receive all tokens of that rebase. As there are many bots that do this as soon as such an opportunity arises, going as far as using the mempool to be sufficiently fast, it is almost certain that all rebases on the LP pair tokens have been skimmed and dumped. Such arbitrage is done with reasonable profit in production on less competitive ohm forks: Buy 2 seconds before rebase: https://snowtrace.io/tx/0x39bb05011dd6f4362914768dc9b045a9240801627a010c886695c32a81f35d2d Sell same amount 2 seconds after: https://snowtrace.io/tx/0x8e803f356766efe5d3c66cf4a386f6fbbf0fb03e1f2fd3ce44fd8b587a1c98a8 Sell rebased amount for profit: https://snowtrace.io/tx/0x8dc5ed8a922ef0cfb5e956398c765d9a5045cfe9f4d812e4c36e9dd21af24025

Recommendation	Consider frequently rebasing and ensuring that no unprivileged user can rebase from a contract which would allow them to flashloan sGG temporarily. Furthermore, consider manually calling `sync()` or skim() on the LP pairs through a contract that calls the rebase. This way, the tokens can either be incorporated in the reserves or taken out of the pairs again to prevent unnecessary selling pressure. It is important that this last step is done within a single transaction by a contract as to not have someone frontrun the governance attempt to take the tokens out again.
Resolution	ACKNOWLEDGED

Issue #11	**`DOMAIN_SEPARATOR` 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. It should be noted that within the EIP-712 draft implementation by OpenZeppelin, the `DOMAIN_SEPARATOR` is somewhat mutable as to allow the `chainId` to evolve. This logic has however not been implemented within this implementation of EIP-712. It should finally be noted that the Permit typehash uses `value` as a parameter, while the function uses amount. It could be better to use value as the permit parameter as well.
Recommendation	Consider making this variable explicitly `immutable`.
Resolution	ACKNOWLEDGED

Issue #12	**Under a constant and small circulating supply, the non-circulating supply starts increasing more rapidly with every rebase**
Severity	INFORMATIONAL
Location	Line 1093 `rebaseAmount = profit_.mul( _totalSupply ).div( circulatingSupply_ );`
Description	The rebase amount is based upon the profit to rebase multiplied by the total supply divided by the circulating supply. This is because the sGG contract is unable to discriminate against sGG within the staking contract during rebases. Contrary to implementations like SafeMoon, there is no way to exclude accounts from rebases. If no adjustment would be made, a portion of the profit would be lost to the sGG that is sitting in the Staking contract. To account for this, the rebase amount is increased to ensure that the circulating supply exactly gets that profit. If then for some reason the `circulatingSupply_` is kept very low, let's say at a nominal 1, the `_totalSupply` increases more rapidly with every rebase. If profit is also 1, and `_totalSupply` is 10, `_totalSupply` would increase to about 20, during the next rebase of 1 profit, `_totalSupply` would increase to 40. In this situation the `MAX_SUPPLY` could be reached rather quickly. This could be a potential denial of service attack during the bootstrapping of an Ohm protocol fork, while there are no stakers yet. We do not see any immediate way to abuse this within the GalaxyGoggle deployment as the situation where there is barely any circulating supply seems extremely unlikely.
Recommendation	Consider this situation carefully. Consider the rate of (exponential) growth of `_totalSupply` under the current setup. This issue will be resolved on the notice that the client has inspected this rate of growth and that `MAX_SUPPLY` is not to be reached in an extremely long time, even if a majority of the stakers decides to unstake.
Resolution	ACKNOWLEDGED

2.4 wsGG

The wsGG token wraps the sGG token in a fixed balance alternative. Instead of increasing in balance with every rebase, wsGG can be liquidated for more and more sGG over time.

2.4.1 Token Overview

Address	0xb18b51dE676f466e3758f9e3B0739CBa564c7209
Token Supply	Unlimited
Decimal Places	9
Transfer Max Size	No maximum
Transfer Min Size	No minimum
Transfer Fees	None
Pre-mints	None

2.4.2 Issues & Recommendations

Issue #13	Gas optimization: Usage of decimals as a precision multiplier
Severity	INFORMATIONAL
Description	The contract uses the wsGG decimals as a precision multiplier. However, as this variable is not immutable, this creates a small ~200 gas overhead every time it is read from memory.
Recommendation	Consider either using a constant precision multiplier, caching the decimals in an immutable variable or making decimals immutable.
Resolution	ACKNOWLEDGED

2.5 TimeBondDepository

The BondDepository is one of the main contracts within Galaxy Goggle. It allows users to sell their LP tokens for GG futures which vest linearly over the next period. Periodically, the rate at which GG is given for LP tokens adjusts upwards or downwards which can be freely configurable by the governance. No more bonds can be issued than a certain maximum. The contribution to this maximum decays over time allowing for more bonds to be issued. Vested GG can be instantly staked if desired by the user. The DAO receives a percentage of the minted GG according to the terms.fee parameter.

2.5.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `initializeBondTerms`
- `setBondTerms`
- `setAdjustment`
- `setTimeStaking`
- `renounceManagement`
- `pushManagement`
- `pullManagement`

2.5.2 Issues & Recommendations

Issue #14	**Vested amount is relocked on deposit, even if the deposit is made by third-parties allowing for targeted Denial of Service**
Severity	HIGH SEVERITY
Description	Currently the `deposit` function, which is used to deposit LP tokens into a bond, will reset the vesting term of any previous deposits. The vested duration since the last redemption would therefore be lost if the user deposits again. This can be used by malicious parties to create griefing for different wallets. The griefing goes as follow: 1. Listen to the BondRedeemed method in the mempool. This way you know when a user is about to claim their vested portion. 2. As soon as you detect it, you send a deposit to them with a tiny amount. This resets their timer. 3. They now need to wait a whole bond duration again Repeat this whenever you detect BondRedeemed in the mempool and you have effectively locked in all GG.
Recommendation	Consider either removing the functionality to deposit to another account or making this a whitelisted operation. The same could be considered for the redeem function to reduce the attack vector. We also recommend removing this functionality from the `redeem` method.
Resolution	ACKNOWLEDGED

Issue #15	The maximum debt can be exceeded by at most `maxPayout`
Severity	MEDIUM SEVERITY
Location	Line 867 `require( totalDebt <= terms.maxDebt, "Max capacity reached" );`
Description	Currently, the check that the maximum amount of debt is not exceeded does not include the newly created debt — this allows for the maximum debt to be exceeded by at most `maxDebt`.
Recommendation	Consider including `value`, which is the new debt, in this requirement. `require( totalDebt.add(value) <= terms.maxDebt, "Max capacity reached" );`
Resolution	RESOLVED The client has stated that they will ensure their bonds will not be maxed out on debt, and they will simply introduce new bonds if the old ones will be maxed out.

Issue #16	Adjustment target is never reached
Severity	LOW SEVERITY
Location	Lines 977-995

```
function adjust() internal {
    uint timeCanAdjust =
adjustment.lastTime.add( adjustment.buffer );
    if( adjustment.rate != 0 && block.timestamp >=
timeCanAdjust ) {
        uint initial = terms.controlVariable;
        if ( adjustment.add ) {
            terms.controlVariable =
terms.controlVariable.add( adjustment.rate );
            if ( terms.controlVariable >=
adjustment.target ) {
                adjustment.rate = 0;
            }
        } else {
            terms.controlVariable =
terms.controlVariable.sub( adjustment.rate );
            if ( terms.controlVariable <=
adjustment.target ) {
                adjustment.rate = 0;
            }
        }
        adjustment.lastTime = uint32(block.timestamp); //
contract description: adjustments are made periodically
        emit ControlVariableAdjustment( initial,
terms.controlVariable, adjustment.rate, adjustment.add );
    }
}
```

Description

The code contains an `adjust` function which allows adjusting the control variable with a fixed increment or decrement after a fixed period. It also contains a target after which the adjustment stops once it is reached.

However, due to the code implementation, the target might be slightly missed, as the adjustment will only stop after it is passed due to the increments being rather large. In addition, if the target would be set close to zero, the subtraction might cause this to revert.

Recommendation Consider setting the info rate to the target once the target has been reached. Consider also resetting the target as to have a cleaner state.

It should be noted that this adjustment method is also slightly wasteful in gas as it often re-reads `terms.controlVariable` from storage. If gas usage is a concern, consider caching some of these variables.

A possible implementation for this recommendation is:

```
function adjust() internal {
    uint256 timeCanAdjust =
adjustment.lastTime.add( adjustment.buffer );
    if( adjustment.rate != 0 && block.timestamp >=
timeCanAdjust ) {
        uint initial = terms.controlVariable;
        uint bcv = terms.controlVariable;
        if ( adjustment.add ) {
            bcv = bcv.add( adjustment.rate );
            if ( bcv >= adjustment.target ) {
                bcv = adjustment.target;
                adjustment.rate = 0;
            }
        } else {
            bcv = bcv > adjustment.rate ?
bcv.sub( adjustment.rate ) : 0;
            if ( bcv <= adjustment.target ) {
                bcv = adjustment.target;
                adjustment.rate = 0;
            }
        }
        adjustment.lastTime = uint32(block.timestamp);
        terms.controlVariable = bcv;
        emit ControlVariableAdjustment( initial, bcv,
adjustment.rate, adjustment.add );
    }
}
```

Resolution ACKNOWLEDGED

Issue #17	**`deposit` is vulnerable to reentrancy if the principle has a reentrancy vector**
Severity	LOW SEVERITY
Description	The `deposit` function currently does adjustments of the `totalDebt` and `value` calculations after the principle has been transferred. This allows an external party to inject code to avoid the `maxDebt` calculation and furthermore manipulate (the current calculator only allows expensive increment-only manipulation by sending tokens to the pair and calling sync) the value in deposit compared to the local value if a token which allows reentrancy is added. With such a token, `maxDebt` could be completely circumvented in the current design. This issue is marked as low severity as we expect principle tokens to be LP pairs mostly — however, we did notice at the Galaxy Goggle website that these can be single asset as well.
Recommendation	Consider reorganizing the `deposit` function to adhere to checks-effects-interactions.
Resolution	ACKNOWLEDGED

Issue #18	**`bondPriceInUSD` is denominated in the decimals of the other token in the LP and might not be correct for non-stablecoin LPs**
Severity	INFORMATIONAL
Description	`bondPriceInUSD` is denominated in the decimals of the other token in the LP and might not be correct for non-stablecoin LPs. As this function is primarily used on the frontend this issue has been marked as informational. ! `standardizedDebtRatio` has similar behavior.
Recommendation	Consider handling this correctly on the frontend.
Resolution	ACKNOWLEDGED

Issue #19	`initializeBondTerms` has no validation
Severity	INFORMATIONAL
Description	The `initializeBondTerms` function has no validation of the parameters that are used for initialization of the bond terms when the contract is first deployed. ❗ `controlVariable` can go to zero with the adjustments, making the `initializeBondTerms` callable again. We are also unsure why there should be an `initialDebt` on initialization function. ❗ The `setAdjustment` function on the other hand becomes completely locked out if `controlVariable` ever reaches zero, which is strange behavior to have defined so implicitly.

Recommendation	Consider adding proper validation for the above function and remove the `initialDebt` parameter if there is no need for an initial debt.
	
	A possible implementation of this recommendation is:
	`require( terms.controlVariable == 0, "Bonds must be initialized from 0" );`
	`require( _controlVariable > 0, "Bonds CV must be initialized greater than 0" );`
	`require( _maxPayout > 0 && _maxPayout <= 1000, "Payout cannot be above 1 percent or zero");`
	`require( _fee <= 10000, "DAO fee cannot exceed payout" );`
	`require( _vestingTerm >= 129600, "Vesting must be longer than 36 hours" );`
	`require( lastDecay == 0, "Bond has already been initialized" );`

Resolution	ACKNOWLEDGED

Issue #20	Contract does not work with a zero `vestingTerm`
Severity	INFORMATIONAL
Description	The `debtDecay` function reverts due to a division by zero if `terms.vestingTerm` is set to zero. In addition, the `percentVestedFor` function will always return a zero vested percentage if the remaining vesting duration is zero (eg. with a zero `vestingTerm`). This should more accurately return 10,000 (100%) as at this point the bonds instantly vests. The contract would therefore become unusable if the `vestingTerm` is zero.
Recommendation	Consider making the requirement of a non-zero vesting term explicit when the term is set.
Resolution	ACKNOWLEDGED

Issue #21	Contract could theoretically run out of GG
Severity	INFORMATIONAL
Description	There is currently no guarantee that the number of GG that the depository receives from the treasury is sufficient to cover the payouts. This is because a profit is withheld by the treasury and a fee is sent to the DAO.
Recommendation	Consider making the requirements within the parameters more explicit as to prevent the situation where more GG can be allocated to payouts than is maintained in the depository. A crude check is to simply reduce the payout to at most the amount received during the deposit function.
Resolution	ACKNOWLEDGED

2.6 EthTimeBondDepository

The EthTimeBondDepository is similar to the TimeBondDepository but differs in that it allows for BNB and WBNB to be deposited. It uses ChainLink to calculate the UI price.

As the EthTimeBondDepository contract is extremely similar to TimeBondDepository, any recurring issues have been omitted from this section of the audit. Users can assume that most if not all of the issues within TimeBondDepository, are also present within EthTimeBondDepository.

2.6.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `initializeBondTerms`
- `setBondTerms`
- `setAdjustment`
- `setTimeStaking`
- `renounceManagement`
- `pushManagement`
- `pullManagement`

2.6.2 Issues & Recommendations

Issue #22	`priceFeed` is `internal`
Severity	LOW SEVERITY
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	ACKNOWLEDGED

Issue #23	**Phishing: Frontend could trick users into sending too much BNB for deposits**
Severity	INFORMATIONAL
Description	Currently, the deposit requires that at least sufficient BNB is sent to cover the provided amount the user wants. However, there is no check that it must be equal. If the frontend is ever compromised, this could be abused to send excessive BNB to the contract without the users earning GG futures in return. The comment on `refundETH` also indicates that the excess BNB goes to the user, while it ironically goes to the DAO.
Recommendation	Consider fixing the above errors.
Resolution	ACKNOWLEDGED

Issue #24	`bondPrice` calculation is inconsistent with TimeBondDepository
Severity	INFORMATIONAL
Description	

TimeBondDepository	EthBondDepository
`price_ = terms.controlVariable.mul( debtRatio() ).add( 1000000000 ).div( 1e7 );`	`price_ = terms.controlVariable.mul( debtRatio() ).div( 1e5 );`

The two `bondPrice` calculations differ in the fact that the TimeBondDepository still adds 1 to the price while the EthTimeBondDepository does not. As these variables are simply used as control variables in production and not as much as real prices we do not see potential production impact of this but we are unsure why this addition was made within the left-hand contract but not on the right hand.

Note that the difference in division is not a problem since both contracts correctly account for the different precision rate.

Recommendation	Consider explaining this inconsistency and if it is not necessary, consider making both contracts more consistent.
Resolution	ACKNOWLEDGED

Issue #25	The protocol will likely malfunction if ChainLink feeds for BNB will be different than 8 decimals
Severity	INFORMATIONAL
Description	Throughout the contract, the assumption is made that ChainLink feed returns a BNB price with 8 decimals. If Chainlink changes this in the future, it will cause multiple UI issues.
Recommendation	Consider upgrading the protocol to dynamically fetch the token's `.decimals()`. It should be noted that this update needs to be made in multiple locations. A first modification should be made by adding a check on the constructor and `setPriceFeed` to not initialize a `priceFeed` with decimals > 8. `require( priceFeed.decimals() == 8, "Price feed decimals > 0");` This check can be added to `assetPrice()` so it will revert in case decimals changes after the contract's deployment.
Resolution	ACKNOWLEDGED

2.7 TimeStaking

TimeStaking is a contract that lets investors stake their GG and receive an equal amount of sGG, which is essentially staked GG, that increases in quantity over time through rebases. When funds are deposited, they are locked into the StakingWarmup for a number of epochs, and after this period, the sGG (including potentially rebased amounts) becomes claimable using the claim contract. As GalaxyGoggle has chosen for a zero length locking period, they have created a StakingHelper contract that stakes and immediately calls claim within a single transaction. This is so that users do not have to make these two transactions themselves. If the lock were to be present, users could call `forfeit()` to retrieve their initial GG and forgo any increase in the locked sGG amount.

Users can call the `unstake` function to trade in an identical amount of sGG for GG. It should be noted that sufficient GG needs to be present in the TimeStaking contract, but this is governed by the other components of the systems.

After the zero length warmup period has expired, anyone can call `claim` on your behalf to move the now unlocked sGG to your wallet. Users should be mindful of this behavior in case they interact with any protocols that blindly take their whole sGG balance. The contract contains a locker which can still be initialized but is presently uninitialized.

Finally, the contract can send a portion of its sGG balance to the locker contract. The locker contract is out of scope of this audit but the basics of this functionality is that the staking contract will see this as an outstanding loan (an asset) and a debt (new sGG in circulation), therefore there is no direct effect on the rebasing. However, as the sGG which is rebased onto the outstanding debt cannot be withdrawn into the staking contract anymore, we therefore assume that this rebased sGG would be used for other means.

2.7.1 Privileged Roles

The following functions can be called by the owner of the contract:

- `setContract`
- `setWarmup`
- `giveLockBonus`
- `renounceManagement`
- `pushManagement`
- `pullManagement`

2.7.2 Issues & Recommendations

Issue #26	**Staked amount is relocked on subsequent stakes, even if the stake is made by third-parties allowing for targeted Denial of Service**
Severity	HIGH SEVERITY
Description	Currently, the `stake` method which is used to stake GG to receive sGG has a capability to use a WarmUp strategy on staking. Everytime someone stakes, their stake is locked by a duration called the `warmupPeriod`. After this period has ended, the staker can claim their rewards. However, since people can stake for others, this can be used to create griefing for different wallets. The griefing goes as follow: 1. Listen to the `unstake()` method being called in the mempool. This way you know when a user is about to claim their staking rewards. 2. As soon as you detect it, you can call `stake()` with a small amount. This resets their warmup timer. 3. They now need to wait a whole new warmup period again. Repeat this whenever you detect a `stake` call in the mempool and you have effectively locked in all the rewards in the staking contract. This issue has been marked as High because within the GalaxyGoggle deployment, the `warmupPeriod` is set to 2 epochs, which means the staking functionality is using a warmup approach.
Recommendation	Consider removing the functionality to stake to another account or making this a whitelisted operation or never use a warmup approach on staking by setting the `warmupPeriod` to `0`.
Resolution	ACKNOWLEDGED The client has acknowledged the issue and indicated that they will keep the warmup period to non-zero so it can not open a possibility to arbitrage. The client stated that it will monitor the contract in case someone starts a griefing process.

Issue #27	Lock logic might be wrongly defined
Severity	LOW SEVERITY
Description	The contract contains two functions: `giveLockBonus` and `returnLockBonus` which facilitate loaning out some of the sGG tokens to the lock contract. However, when the loan is repaid, the lock retains all rebased rewards. This is not necessarily incorrect but as no lock contract was included within the scope of this audit, we are unsure about whether this is desired. From a technical perspective, the current implementation does make sense as if the "interest" would be sent back, `returnLockBonus` would add credit to the staking contract balance which might have unintended side-effects.
Recommendation	Consider whether this is desired behavior.
Resolution	RESOLVED The client has stated that this is desired behavior.

Issue #28	Rebases can be arbitraged/frontran
Severity	INFORMATIONAL
Description	Although the contract protects against flash-loaning GG to use it to capture rebases, an advanced party could still purchase GG the block before a rebase occurs to then sell it afterwards. If this party has some control over their timing or some control to prevent other users from arbitraging their purchase, this could be profitable and result in less sGG for the other stakers.
Recommendation	Consider rebasing very frequently or using a staking method where GG staked is directly incorporated. Locking stakes as is done in Ohm is also possible.
Resolution	ACKNOWLEDGED The client stated that they may introduce OHM logic in the future.

2.8 Distributor

The distributor is a contract that mints GG to the governance-configured recipients every time an epoch ends. The amount of GG to mint is a percentage set by the governance of the total GG supply. The distributor therefore has the ability to trigger a minting from the TimeTreasury to all the recipients added by the governance at every epoch. Currently, the only recipient is the staking contract. Therefore, every time a rebase is done at the end of an epoch, the total supply of GG increases. After each distribution, the rate of the distribution is adjusted based on an adjustment variable that is set by the governance. Currently there is no adjustment variable set within the Galaxy Goggle deployment so the rate remains the same after each distribution.

Users should carefully keep an eye on this contract as it has the power to distribute the whole GG supply (and more!) to recipients at every epoch. Currently only the TimeStaking contract is set as a recipient earning 1.1% of the GG supply at every distribution.

2.8.1 Privileged Roles

The following functions can be called by the Staking contract:

- `addRecipient [ ! dangerous ]`
- `removeRecipient`
- `setAdjustment [ ! dangerous ]`
- `renouncePolicy`
- `pushPolicy`
- `pullPolicy`

2.8.2 Issues & Recommendations

Issue #29	Adjustment target is never reached
Severity	LOW SEVERITY
Location	Lines 448-463

```
function adjust( uint _index ) internal {
    Adjust memory adjustment = adjustments[ _index ];
    if ( adjustment.rate != 0 ) {
        if ( adjustment.add ) { // if rate should increase
            info[ _index ].rate =
info[ _index ].rate.add( adjustment.rate ); // raise rate
            if ( info[ _index ].rate >= adjustment.target )
{ // if target met
                adjustments[ _index ].rate = 0; // turn off
adjustment
            }
        } else { // if rate should decrease
            info[ _index ].rate =
info[ _index ].rate.sub( adjustment.rate ); // lower rate
            if ( info[ _index ].rate <= adjustment.target )
{ // if target met
                adjustments[ _index ].rate = 0; // turn off
adjustment
            }
        }
    }
}
```

Description

The code contains an `adjust` function which allows for the adjustment of the emission rate towards a recipient with a fixed increment or decrement after every distribution. It also contains a target where the adjustment stops once it is reached.

However, due to the code implementation, the target might be slightly missed, as the adjustment will only stop after it is passed due to the increments being rather large.

In addition, if the target is set close to zero, the subtraction might cause this to revert.

Recommendation Consider setting the info rate to the target once the target has been reached. Consider also resetting the target as to have a cleaner state.

```
info[ _index ].rate = adjustment.target;
delete adjustments[ _index ];
```

It should be noted that this adjustment method is also slightly wasteful in gas as it often re-reads `info[index].rate` from storage. If gas usage is a concern, consider caching some of these variables.

A possible implementation of this recommendation is:

```
function adjust( uint _index ) internal {
    Adjust memory adjustment = adjustments[ _index ];
    if ( adjustment.rate != 0 ) {
        uint initial = info[ _index ].rate;
        uint rate = initial;
        if ( adjustment.add ) { // if rate should increase
            rate = rate.add( adjustment.rate ); // raise rate
            if ( rate >= adjustment.target ) { // if target met
                rate = adjustment.target;
                delete adjustments[ _index ];
            }
        } else { // if rate should decrease
            rate = rate > adjustment.rate ?
rate.sub( adjustment.rate ) : 0;
 // lower rate
            if ( rate <= adjustment.target ) { // if target met
                rate = adjustment.target;
                delete adjustments[ _index ];
            }
        }
        info[ _index ].rate = rate;
        emit LogAdjust(initial, rate, adjustment.target);
    }
}
```

Resolution ACKNOWLEDGED

Issue #30	Unbounded gas usage due to extensive for-loop usage
Severity	LOW SEVERITY
Description	Many `for` loops are used to iterate over the recipients. If there are many recipients, this results in a high gas cost and could increase in gas cost to the point where the distribute function would become uncallable. Since `removeRecipient` does not reduce the loop size, there could be a point in time where a new Distributor would have to be deployed as gas cost has risen so much. This issue is marked as low severity since currently there is only a single recipient, the TimeStaking contract. This issue therefore does not present itself within the Galaxy Goggle deployment just yet.
Recommendation	Consider enforcing a limit of recipients within `addRecipient` as a reminder that this is not unbounded. Consider also reusing indices if recipients are removed by using the traditional array index deletion pattern where the last index is moved into the deleted index, and the array is shortened by one. This pattern requires a re-linking of the adjustments mapping to the new index.
Resolution	ACKNOWLEDGED

Issue #31	Lack of validation
Severity	LOW SEVERITY
Description	Currently there is no upper limit to the rate which a recipient can receive newly minted GG at. If the governance of this contract was ever compromised, or user error occurred, the whole supply and more could be minted to a recipient by accident.
Recommendation	Consider requiring the rates and target to be within reasonable limits, eg. 5% of the total supply at most. Consider also limiting the number of recipients to a reasonable number, eg. 5 recipients at most.
Resolution	ACKNOWLEDGED

Issue #32	Adjustments are not reset when recipient is removed
Severity	INFORMATIONAL
Description	Within the `removeRecipient` function, the relevant adjustment struct is not deleted. Deleting this struct might be considered cleaner and could also reduce the gas cost of `removeRecipient`. If the code is ever updated to reuse the empty index on deletion (array index deletion pattern), deleting the adjustment would also be a defensive move if the new codebase forgets to also move the adjustment into the empty index.

Recommendation	Consider resetting the adjustment on `removeRecipient`. A possible implementation of this recommendation is:
	`function removeRecipient( uint _index, address _recipient ) external onlyOwner {`
	`  require( _recipient == info[ _index ].recipient, "Invalid recipient" );`
	`  require( _index < info.length, "Index out of range" );`
	`  if ( _index < info.length - 1 ) {`
	`  info[ _index ] = info[ info.length - 1 ];`
	`  adjustments[ _index ] = adjustments[ info.length - 1 ];`
	`  }`
	`  delete adjustments[ info.length - 1 ];`
	`  info.pop();`
	`  emit RecipientRemoved( _recipient, _index );`
	`}`

Resolution	ACKNOWLEDGED

2.9 StakingHelper

The StakingHelper is a simple utility contract that has just 1 method that does a stake and a claim within one transaction for the staking contract. As Galaxy Goggle does not use the locking functionality introduced in Ohm, they want to avoid the requirement that people have to call "claim" themselves manually after each staking.

2.9.1 Issues & Recommendations

Issue #33	Phishing: `stake` function allows to stake towards a different address
Severity	LOW SEVERITY
Description	The `stake` function withdraws GG tokens from the transaction sender. However, these tokens are granted to the recipient which is a parameter of the `stake` function. If the frontend is ever compromised, it could be expected that the hacker might simply set the recipient parameter to their wallet to steal all newly staked sGG. Most advanced users have become adept enough to check the contract which they are interacting with, but not yet the parameters, as these are displayed in bytecode by MetaMask. This issue applies to the TimeStaking contract as well.
Recommendation	Consider only allowing staking to one's own account. Consider also making this fix in TimeStaking if it is a user-facing contract. ❗ If this recommendation is not implemented, we recommend non-zero validation in the staking contract to provide an explicit error message if the recipient is set to the zero address to prevent user errors. It should be noted that due to the overrides within the sGG token, these tokens no longer revert on transfers to the zero address, GG tokens still do.
Resolution	ACKNOWLEDGED

2.10 StakingWarmup

The StakingWarmup contract is a very simple helper contract that is used to store the staked GG balances of users. It is used exclusively by the TimeStaking contract though everyone can of course transfer both sGG and GG to it manually.

2.10.1 Privileged Roles

The following functions can be called by the Staking contract:

- `retrieve`

2.10.2 Issues & Recommendations

No issues found.

2.11 TimeBondingCalculator

The TimeStakingBondingCalculator was designed by Olympus DAO as an LP-valuing contract that would use 1 OHM = 1 DAI as the values of the individual components of the LP pair. It uses the correct approach of valuing LP pairs by rebalancing the pair as to have equally valued reserves. It was, however, only designed to work for OHM+stable pairs and assumes that OHM is worth $1 to derive the value of the pair.

2.11.1 Issues & Recommendations

Issue #34	Does not support LP pairs where the second currency has less than 9 decimals
Severity	MEDIUM SEVERITY
Location	Line 278 `uint decimals = token0.add( token1 ).sub( IERC20( _pair ).decimals() );`
Description	The TimeBondingCalculator does a decimal adjustment to make sure that whatever the decimals of the two LP tokens, the resulting number of decimals is 18. This calculation is: `decimals(token0) + decimals(token1) - decimals(pair)` As GG has 9 decimals and the pair 18 decimals, the paired token must have at least 9 decimals or this calculation will revert due to underflow. This is notoriously not the case for most stablecoins on avalanche making this contract unusable for these.
Recommendation	Consider adjusting the logic to start multiplying instead of dividing if the decimals would be negative. The client could consider an if-else branch for if the pair decimals are smaller than the sum of the token decimals and invert the logic for the new branch.

A possible implementation of this recommendation is:

```
function getKValue( address _pair ) public view returns( uint
k_ ) {
    uint token0Decimals =
IERC20( IUniswapV2Pair( _pair ).token0() ).decimals();
    uint token1Decimals =
IERC20( IUniswapV2Pair( _pair ).token1() ).decimals();
    uint pairDecimals = IERC20( _pair ).decimals();
    (uint reserve0, uint reserve1, ) =
IUniswapV2Pair( _pair ).getReserves();
    uint decimalsDelta;
    if (token0Decimals.add(token1Decimals) < pairDecimals) {
       decimalsDelta =
pairDecimals.sub(token0Decimals.add(token1Decimals));
        k_ = reserve0.mul(reserve1).mul( 10 ** decimalsDelta );
    } else {
        decimalsDelta =
token0Decimals.add(token1Decimals).sub(pairDecimals);
        k_ = reserve0.mul(reserve1).div( 10 ** decimalsDelta );
    }
}
```

Resolution

RESOLVED

The client has stated that no LP will have less than 9 decimals in the current setup. If in the future an LP that goes against this constraint will need to be supported, the client will deploy a new instance of the TimeBondingCalculator.

Issue #35	**TimeBondingCalculator can only value pairs in which the two tokens have equal "value"**
Severity	LOW SEVERITY
Description	The TimeStakingBondingCalculator was designed by Ohm as an LP valuing oracle that would use 1 OHM = 1 DAI as the oracle value to value the number of OHMs (or DAI) the LP is worth. It was only designed to work for OHM+stable pairs. The bonding calculator is therefore insufficiently equipped for tokens with unequal 'value' (within parentheses as the value of GG is not equal to $1, however the system uses this to calculate the value).
Recommendation	Consider this carefully and consider using different oracles if other LP pairs need to be priced, or if pricing needs to occur at the current GG value. The client should remember that pricing LPs is notoriously difficult and that an approach involving K and oracle prices would still be required. Furthermore the client should remember that within the TimeTreasury system, the LPs are not valued at their present value, instead they are valued at their eventual $1 value.
Resolution	ACKNOWLEDGED

Issue #36	Markdown will misbehave if an LP with two non-GG tokens is provided
Severity	LOW SEVERITY

Location	Lines 299-303
	`if ( IUniswapV2Pair( _pair ).token0() == Time ) {`
	`    reserve = reserve1;`
	`} else {`
	`    reserve = reserve0;`
	`}`

Description	Currently the markdown function assumes it is being provided a GG/OTHER LP. However, if it were to be provided an OTHER/OTHER LP, it would wrongly assume that the second token is in fact GG.

Recommendation	Consider this carefully in all locations where markdown is used, or consider making this function explicitly revert in this circumstance.
	
	A possible implementation of this recommendation is:
	`function markdown( address _pair ) external view returns ( uint ) {`
	`    ( uint reserve0, uint reserve1, ) = IUniswapV2Pair( _pair ).getReserves();`
	`    uint reserve;`
	`    if ( IUniswapV2Pair( _pair ).token0() == GIZA ) {`
	`        reserve = reserve1;`
	`    } else {`
	`        require(IUniswapV2Pair( _pair ).token1() == GIZA, "Not a GIZA LP pair");`
	`        reserve = reserve0;`
	`    }`
	`    return reserve.mul( 2 * ( 10 ** IERC20( GIZA ).decimals() ) ).div( getTotalValue( _pair ) );`
	`}`

Resolution	ACKNOWLEDGED

Issue #37	UI `markdown` function is vulnerable to price manipulation
Severity	INFORMATIONAL
Description	The markdown function can be manipulated at a relatively low cost by wrapping the call in a buy and sell (or vice-versa) to adjust the reserves. This leads to the markdown function, which is used to calculate the relative value of the pair (compared to the long-term value), being unuseable for any oracle functionality as it can be manipulated. This issue is marked as informational as it is presently only used for UI functionality. However, if it were to ever be used in new contracts as a genuine oracle, this would likely lead to exploitation, hence Paladin has decided to include this as an informational issue.
Recommendation	No action is required. Never use this function as an oracle or a trusted source.
Resolution	ACKNOWLEDGED

2.12 TimeTreasury

The treasury is one of the central components within GalaxyGoggle. It keeps all the underlying assets that are deposited through the bonds and keeps track of debt if any other components borrow these treasuries.

It uses a queue approach to change the governance addresses for different actions inside the treasury which is very similar to a timelock. It also allows for certain addresses to borrow from the Treasury (against sGG) and repay the borrowed amount, and it also allows for the debt to be repaid with GG. Finally and most importantly, it allows any reward manager (eg. the staking distributor) to mint GG. It should be noted that no more GG can be minted than the total number of reserves in $. If GG would be freely exchangeable for the reserves, this puts a lower limit of $1 on the value of GG as long as no reserves are lost.

It should be noted that the treasury code is currently written extremely verbose and implementing a dependency like AccessControl (RBAC) might simplify the contract greatly.

2.12.1 Privileged Roles

The following functions can be called by the Staking contract:

- `auditReserves`
- `toggle`
- `incurDebt`
- `withdraw`
- `deposit`
- `repayDebtWithReserve`
- `manage`
- `mintRewards`
- `renounceManagement`
- `pushManagement`
- `pullManagement`

2.12.2 Issues & Recommendations

Issue #38	Treasury does not have sGG set as a contract
Severity	LOW SEVERITY
Description	Inside the OHM protocol, the Treasury has a functionality where an allowed borrower can borrow assets from the Treasury, against the total supply of staked token. This expands the functionality of the OHM protocol by using those assets to be traded outside OHM protocol. The Galaxy Google has also this functionality inside the Treasury but the staked token is not set. sGG is used inside the Treasury to calculate the `incurDebt` that determines the max amount of allowed borrow reserves based on the sGG supply. Currently the sGG is set to address 0 which means this functionality will revert.
Recommendation	Consider setting sGG to the staked GG contract once allowed borrowers want to use this functionality.
Resolution	RESOLVED

Issue #39	`incurDebt` can be used to drain the TimeTreasury from ReserveToken
Severity	LOW SEVERITY
Description	The treasury can borrow assets marked as ReserveToken using the borrower's balance in sGG. This functionality can be used by the protocol to borrow assets that can be used for external actions. Due to the fact that the maximum debt that an allowed borrower can borrow it is based on their balance in sGG, if by any chance a malicious actor manages to whitelist multiple addresses as borrowers, he can use the same balance to drain the TimeTreasury from ReserveToken because the only limit is defined by the borrower's balance. For example: A malicious actor manages to whitelist multiple addresses, let's call them address A, B, C. He has 1000 sGG in address A, he borrows the amount of ReserveToken from the treasury, then he transfers the sGG to address B, uses address B to borrow another batch and so on.
Recommendation	Consider locking the balance of sGG of the borrower until he pays the debt or implement some sort of maximum amount/percentage that can be borrowed across the whole TimeTreasury.
Resolution	RESOLVED The client has stated that currently there is no `debtManager` defined in the TimeTreasury. If in the future the client plans to introduce one, it would take a queue/toggle submitted by 3/5 signers on the multisig.

Issue #40	**Lack of component safeguards in a system that plans to increase in number of components over time is considered brittle**
Severity	LOW SEVERITY
Description	The treasury is responsible for minting new GG. Any account with the reward manager role can do this. This however also means that if any single of these accounts or contracts would be compromised, the whole system would fail. Such a practice is not bad in itself, but it is a setup we like to call 'brittle'. In general, when the security of a system is based upon all components acting correctly, and this set of components is planned to increase over time, odds are that one day a component will misbehave and the whole system goes under. This has been witnessed with Cream recently on Ethereum and more traditionally with PancakeBunny (and many of their forks) on BSC and other chains.
Recommendation	Consider incorporating hourly limits to all functions within the treasury, each account can only mint/borrow/... up to their hourly limit every hour. Permissions should be pausable instantly by the DAO. With such a setup, if a new component ever turns out to have a vulnerability, only a few hours of mints might be stolen.
Resolution	ACKNOWLEDGED

Issue #41	**Adding a token as both a liquidity and reserve token would cause it to be double counted in the treasury value**
Severity	LOW SEVERITY
Description	Currently the function that calculates the value of the reserves does not validate that a token has already been counted. This means that if the same token is added twice to the reserves lists, this token would be double counted. The client has considered this possibility by not allowing a token to be added twice to either the reserve tokens list or the liquidity tokens list. However, the possibility remains open that the token is added to both the reserve and liquidity tokens list once, which would cause double counting.
Recommendation	Consider either not double counting in the reserve value calculating function, or consider not allowing a token to be added to either of these lists. A possible implementation of recommendation by not double counting it is:

```
function auditReserves() external onlyManager() {
    uint256 reserves;
    for( uint256 i = 0; i < reserveTokens.length; i++ ) {
       if ( isReserveToken[ reserveTokens[ i ] ] ) {
           reserves = reserves.add (
               valueOf( reserveTokens[ i ],
IERC20( reserveTokens[ i ] ).balanceOf( address(this) ) )
           );
        }
    }
    for( uint256 i = 0; i < liquidityTokens.length; i++ ) {
        if ( !isReserveToken[ liquidityTokens[ i ] ] &&
isLiquidityToken[ liquidityTokens[ i ] ] ) {
            reserves = reserves.add (
                  valueOf( liquidityTokens[ i ],
IERC20( liquidityTokens[ i ] ).balanceOf( address(this) ) )
            );
        }
    }
    totalReserves = reserves;
    emit ReservesUpdated( reserves );
    emit ReservesAudited( reserves );
}
```

Resolution ACKNOWLEDGED

Issue #42	**`repayDebtWithOhm` has inconsistent privilege requirements which allows for slight privilege escalation**
Severity	LOW SEVERITY
Description	Currently, the `repayDebtWithOhm` function requires the sender to have the "debtor" role, which essentially means they can borrow from the reserve. However, this operation also does a withdrawal from the reserve which normally requires the "reserve spender" role. This role verification is not made however. To clarify on this: `repayDebtWithOhm` is essentially a combination of `withdraw`, which allows you to withdraw reserves if you burn an equivalent amount of GG and `repayDebtWithReserve`, which allows you to repay your debt by transferring tokens to the reserve. `repayDebtWithOhm` combines these by having you repay your debt by burning GG. As the roles already have very large privileges within the system, this issue is only marked as low risk since it hardly increases the risk profile.
Recommendation	Consider also requiring the `reserveSpender` role for the `repayDebtWithOhm` function, as this behavior seems inconsistent with what the roles should be allowed to do. A possible implementation of this recommendation is to add the following require in the `repayDebtWithOhm` function: `require( isDebtor[ msg.sender ] && isReserveSpender[ msg.sender ], "Not approved as debtor or spender" );`
Resolution	ACKNOWLEDGED

Issue #43	Unnecessary comparison to `true` on `withdraw` function
Severity	INFORMATIONAL
Location	Line 368 `require( isReserveSpender[ msg.sender ] == true, "Not approved" );`
Description	In the `withdraw` function, the `require` checks to see if the `msg.sender` is in the `isReserveSpender` mapping, which is a mapping of `address=>bool`. Therefore comparison to `true` is redundant.
Recommendation	Consider removing the comparison to `true`. `require( isReserveSpender[ msg.sender ], "Not approved" );`
Resolution	ACKNOWLEDGED

Issue #44	Reserve value mechanism could cause withdrawals and other operations to temporarily fail
Severity	INFORMATIONAL
Description	Whenever tokens are added or deleted to the Treasury, the present (USD) value of them is added or subtracted to the `totalReserves`. However, if their value increases over time for some reason, withdrawals might revert because the functions try to reduce `totalReserves`. This could also be abused if ever an oracle (`bondCalculator`) is used that turns out to be manipulatable, in this case a malicious party can always increase the value of the currency before it is withdrawn to potentially cause that withdrawal to revert.
Recommendation	As this is a purely informational issue, no specific steps need to be taken unless the client considers this prohibitive. The client should simply make sure to call `auditReserves` if withdrawals start failing and build no functionality that relies on withdrawals to work 100% of the time (otherwise this functionality should call `auditReserves`). If withdrawals start failing often due to DoS, the client should remember this issue and double check if some oracle is malfunctioning.
Resolution	ACKNOWLEDGED

Issue #45	Lack of `safeTransfer` usage within `incurDebt`
Severity	INFORMATIONAL
Location	Line 402 `IERC20( _token ).transfer( msg.sender, _amount );`
Description	In the `incurDebt` function, the `transfer` method is used to transfer tokens from Treasury to the `msg.sender`. This will not work for tokens that will return false on `transfer` (or malformed tokens that do not have a return value).
Recommendation	Consider using `safeTransfer` instead of `transfer` as is done throughout most of this contract.
Resolution	ACKNOWLEDGED

Issue #46	**Manage will always do an excessReserve check even if the token is not within the liquidity or reserves tokens**
Severity	INFORMATIONAL

Location	Lines 455-456
	`uint value = valueOf(_token, _amount);`
	`require( value <= excessReserves(), "Insufficient reserves" );`

Description	Currently, the reserves are only comprised of "liquidity" and "reserve" tokens. Therefore, if an asset which is not within these two categories is withdrawn from the Treasury, it should not affect the excess reserves and the excess reserves validation is therefore redundant.

Recommendation	Consider wrapping the excess reserve check in an `if` statement that only executes if the token that is being withdrawn is part of the liquidity or reserves tokens. Otherwise consider requiring the token being withdrawn to be within either of these categories.
	
	A possible implementation of this recommendation is replacing the require of the `excessReserves()` (Line 456) in the `manage` function with:
	`if ( isLiquidityToken[ _token ] || isReserveToken[ _token ])`
	`    require( value <= excessReserves(), "Insufficient reserves" );`

Resolution	ACKNOWLEDGED

2.13 Code style-related issues

The following are coding style issues that Paladin reported throughout the contracts of the GalaxyGoggle protocol. Paladin has aggregated the ones that occurred frequently into this section to shorten the report.

2.13.1 Issues & Recommendations

Issue #47	**Inconsistency: Unused `mint` function emits an event from `address(this)` while the mint logic during initialization emits an event from the zero address**
Severity	INFORMATIONAL
Description	The `_mint` function is inconsistent with the way tokens are actually minted in that it sets the token contract itself as the transfer origin. This would likely mislead explorers and third-party tools into thinking that tokens were taken out of the token contract itself. - TimeERC20: _mint - MEMOries: _mint This issue has been marked as informational as `_mint` is presently unused, making this a purely informational concern.
Recommendation	Consider making `_mint` consistent with the recommended practice of using address zero as the origin for the mint transfer.
Resolution	ACKNOWLEDGED

Issue #48	**Various functions can be made external**
Severity	INFORMATIONAL
Description	Functions that are not used within the contract but only externally can be marked as such with the `external` keyword. Apart from being a best practice when the function is not used within the contract, this can lead to a lower gas usage in certain cases. - TimeERC20: `burn` and `burnFrom` - TimeStaking: `claim` and `index` - Distributor: `nextRewardFor`
Recommendation	Consider marking the above variables as `external`.
Resolution	ACKNOWLEDGED

Issue #49	Lack of events for various functions
Severity	INFORMATIONAL
Description	Functions that affect the status of sensitive variables should emit events as notifications. - **MEMOries**: `setIndex` ❗ The return value is furthermore unnecessary unless this is some highly standard interface like ERC-20. - **wsGG**: `wrap` and `unwrap` - **TimeStaking**: `stake`, `claim`, `forfeit`, `toggleDepositLock`, `unstake`, `rebase`, `giveLockBonus`, `returnLockBonus`, `setContract` and `setWarmup` - **TimeBondDepository**: `initializeBondTerms`, `setBondTerms`, `setAdjustment`, `setTimeStaking` and `recoverLostToken` - **Distributor**: `distribute`, `adjust`, `addRecipient`, `removeRecipient` and `setAdjustment` - **StakingHelper**: `stake`
Recommendation	Add events for the above functions. Consider removing the return variable.
Resolution	ACKNOWLEDGED

Issue #50	Typographical errors
Severity	INFORMATIONAL

Description	**Distributor**: The contract still references the OHM token while the token is in fact named GG within the Galaxy Goggle deployment.
	
	MEMOries: Token comments mention ERC-777 which is not relevant to sGG.
	
	Line 544 (example)
	`// Present in ERC777`
	
	Throughout the ERC20 dependency of the tokens, references to EIP-777 are made. This improved token standard is known to cause many exploits as it allows for reentrancy on any transfer, it could therefore mislead less-adept third-party reviewers into thinking this is an ERC-777 token while it in fact is not.
	
	Line 633 furthermore contains a typographical error:
	`// Overridden in ERC777`
	
	TimeStaking: Throughout the contract, references to OHM and sOHM, TIME, MEMO are made within the comments or variables. However, the tokens within this deployment are called GG and sGG. `setWarmup` is also an ambiguous function name and should be renamed to `setWarmupPeriod`.
	
	TimeBondDepository: The contract still references the OHM throughout the comments while the token is in fact named Time within the Galaxy Goggle deployment.
	
	There is also an unnecessary conversion of `treasury` to `address`:
	Line 889
	`address( treasury )`
	
	TimeStaking: Throughout the contract, references to OHM and sOHM, TIME, MEMO are made within the comments or variables. However, the tokens within this deployment are called GG and sGG. `setWarmup` is also an ambiguous function name and should be renamed to `setWarmupPeriod`.

Recommendation	Consider fixing the above typographical errors.
Resolution	ACKNOWLEDGED

Issue #51	Unused variables/dependencies throughout the contracts
Severity	INFORMATIONAL

Description	The contract includes unused variables. These unnecessarily increase the contract source code size and gas consumption, also it can make third-party reviewing more cumbersome.
	
	TimeERC20:
	- `ERC20TOKEN_ERC1820_INTERFACE_ID`
	- unused dependency EnumerableSet
	- Line 610
	`bytes32 constant private ERC20TOKEN_ERC1820_INTERFACE_ID = keccak256( "ERC20Token" );`
	
	The custom token dependency contains a constant variable containing a hashed interface identifier. However, this variable is not used throughout the contract.
	
	There are furthermore many references to ERC-777 which could mislead less adept code reviewers into believing this is an ERC-777 token. As ERC-777 tokens are traditionally associated with exploit vulnerability, it is best to avoid such confusion.
	
	wsGG:
	- `Address` and `SafeERC20`
	`Line 749-750`
	`using SafeERC20 for ERC20;`
	`using Address for address;`
	
	TimeBondDepository: `ERC20`, `ERC20Permit`, `IERC2612Permit` and `Counters`
	
	Distributor: `SafeERC20`

Recommendation	Consider removing the above variables.
Resolution	ACKNOWLEDGED

Issue #52	Gas optimization: Contract uses hardcoded strings in SafeMath functions
Severity	INFORMATIONAL

Location	TimeERC20::Lines 900-903 (example)
	`uint256 decreasedAllowance_ =`
	`  allowance(account_, msg.sender).sub(`
	`  amount_,`
	`  "ERC20: burn amount exceeds allowance"`
	`  );`

Description	The contract injects the error message into SafeMath. This is known to cost extra gas, even on the happy path, as it causes memory allocation.
Recommendation	Consider checking the identity explicitly using a require statement and then using non-safe math to do the subtractions and additions instead. SafeMath has also created the trySub and tryAdd functions in more recent versions to address this gas usage concern.
Resolution	ACKNOWLEDGED

Issue #53	Uncast addresses make the code more verbose than it needs to be
Severity	INFORMATIONAL
Location	TimeTreasury:Line 435 (Example) `IOHMERC20( Time ).burnFrom( msg.sender, _amount );`
Description	Throughout the contracts, addresses are stored using the address type instead of the interface type. This requires the code to cast them to the correct interface every time these addresses are used and makes them prone to typing errors. It should be noted that address typing is purely syntactic and does not have any runtime benefits (either through performance security).
Recommendation	Consider casting all addresses to the correct types within the storage portions of the contract. This is done in an extreme number of locations so the client will have to simply go over all contracts to do this. Sometimes an address has multiple types (eg. IsGG + IERC20) — in this case we recommend making an aggregate interface that inherits both of them or making IsGG inherit IERC20 in this example.
Resolution	ACKNOWLEDGED

Issue #54	Ambiguous errors
Severity	INFORMATIONAL

Location	**Examples**
	MEMOries::Line 1015
	`require(msg.sender == stakingContract);`
	
	MEMOries::Line 1074
	`require(INDEX == 0);`

Description	The contract contains locations of code which do not revert with an error message, instead they revert ambiguously leaving users to potentially wonder what happened with their transaction. It makes writing coverage tests furthermore difficult as these cannot explicitly check for the reversion method. Within the sGG token, often transfers also revert with ambiguous errors if the user does not have enough allowance or tokens. This is likely the most severe location of this issue as these events might occur frequently.
Recommendation	Consider adding explicit reversion messages to the above locations and any other reversion locations which could cause a worse user experience.
Resolution	ACKNOWLEDGED

Issue #55	Gas optimization: Storage variables are frequently unnecessarily reread
Severity	INFORMATIONAL

Location	MEMOries::Lines 1195-1196 (example)
	`_allowedValue[ msg.sender ][ spender ] =`
	`_allowedValue[ msg.sender ][ spender ].add( addedValue );`
	`emit Approval( msg.sender, spender,`
	`_allowedValue[ msg.sender ][ spender ] );`

Description	The contract often unnecessarily re-reads variables from storage when they could be derived from variables stored in memory. This causes gas to be wasted unnecessarily (about 200 gas per read). This issue is aggregated into a single issue as we wish to not unnecessarily clutter the report with a high issue count given that the client is unlikely to redeploy. Upon request by GalaxyGoggle, our internal documentation with all locations of code that can be optimized can be provided either in the report or privately.
Recommendation	Consider caching variables that are reread multiple times.
Resolution	ACKNOWLEDGED

2.14 Inapplicable deployment Issues

2.14.1 Issues & Recommendations

Issue #56	Inapplicable deployment related issues
Severity	INFORMATIONAL
Description	Under the following deployment circumstances, this contract might malfunction. As the following contracts has already been deployed and these circumstances are not present, this issue has been automatically marked as resolved. **TimeBondDepository:** GG tokens which return `false` are not supported, tokens which do not return a boolean are not supported. Consider using `safeTransferFrom`. It should be noted that `safeTransferFrom` is consistently used throughout the rest of the protocol except in this contract and a few others making this an inconsistency issue as well. **EthTimeBondDepository**: WETH tokens which return `false` are not supported, tokens which do not return a boolean are not supported. Consider using `safeTransferFrom`. It should be noted that `safeTransferFrom` is consistently used throughout the rest of the protocol except in this contract and a few others making this an inconsistency issue as well. **StakingHelper**: GG tokens which return `false` are not supported, tokens which do not return a boolean are not supported. Consider using `safeTransferFrom`. It should be noted that `safeTransferFrom` is consistently used throughout the rest of the protocol except in this contract and a few others making this an inconsistency issue as well. **StakingWarmup**: MEMOries tokens which return `false` are not supported, tokens which do not return a boolean are not supported. Consider using `safeTransfer`. It should be noted that `safeTransfer` is consistently used throughout the rest of the protocol except in this contract and a few others making this an inconsistency issue as well.

Recommendation	As the contracts are correctly deployed this issue does not apply to the main Galaxy Goggle deployment on Binance Smart Chain.
Resolution	RESOLVED


PALADIN
BLOCKCHAIN SECURITY