PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Battle For Giostone

05 November 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 Battle For Giostone on the Avalanche 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	Battle For Giostone
URL	https://battleforgiostone.com/
Network	Avalanche
Language	Solidity

1.2 Contracts Assessed

Name	Contract	Live Code Match
AdvisorsContract	0x69ae264162c32FEFFE1656751ABD69Ba76e2d306	MATCH
ReserveFund	0x872443A10843fFf012ef75a2daC6fDC7f5580932	MATCH
PlayerRewardsContract	0x70d2f5877eD1Ae0AF659a3aeC46799568c8DCc2F	MATCH
SeedContract	0xF394effBb63C925f7b2533cEc03cbca0311F597d	MATCH
PrivateRoundContract	0x9973c0Fff6e0c3aE683337e8B5760b2a0A437647	MATCH
AirdropContract	0xea75db6b617e6d3375372b9fa2aaf49e5b01cc81	MATCH
LiquidityContract	0xb7b608F6c3719aD61EDFcc2F6C2fD5378940e859	MATCH
EcosystemFundContract	0x98A10009Ce0c795Ba8779E518fd115dB6A902EcE	MATCH
TeamContract	0x4cfA9ECA7317d099d447032f700F43E5cE2D30C3	MATCH

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	3	3	-	-
Medium	2	2	-	-
Low	7	7	-	-
Informational	13	12	1	-
Total	**25**	**24**	**1**	**-**

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 Global Issues

ID	Severity	Summary	Status
01	LOW	itoken is private	✓ RESOLVED
02	INFO	The contract pragmas are extremely wide	✓ RESOLVED
03	INFO	Typographical errors and gas optimizations	PARTIAL
04	INFO	Contracts do owner management manually	✓ RESOLVED
05	INFO	Lack of `safeTransfer` usage	✓ RESOLVED

1.3.2 AdvisorsContract, ReserveFund and PlayerRewardsContract

ID	Severity	Summary	Status
06	HIGH	Contract will likely gridlock and prevent any further claiming with a large amount of users (~90+ recipients)	✓ RESOLVED
07	MEDIUM	Recipients who claim their whole allocation at the end of the vesting will receive less than expected and mess up the contract logic due to an overflow exception	✓ RESOLVED
08	LOW	Contract can theoretically be re-initialized once everyone withdraws	✓ RESOLVED
09	LOW	Claiming delays a user their vesting schedule by up to 24 hours due to rounding	✓ RESOLVED
10	LOW	Recipients can instantly unlock their whole allocation through a reentrancy exploit if the token allows for reentrancy	✓ RESOLVED
11	INFO	Typographical errors	✓ RESOLVED
12	INFO	Division before multiplication increases rounding error slightly within the function which calculates the amount of tokens to receive	✓ RESOLVED

1.3.3 SeedContract and PrivateRoundContract

ID	Severity	Summary	Status
13	LOW	`addWhiteList` does not adhere to the checks-effect-interaction pattern and the owner could potentially withdraw more tokens than expected on TGE	✓ RESOLVED

1.3.4 AirdropContract and LiquidityContract

ID	Severity	Summary	Status
14	INFO	`init` function logic is also executed within allocate and therefore seems redundant	✓ RESOLVED
15	INFO	`token` is never set	✓ RESOLVED
16	INFO	Typographical errors and gas optimizations	✓ RESOLVED

1.3.5 EcosystemFundContract

ID	Severity	Summary	Status
17	MEDIUM	Using uint16 for the vestingCycles variable type could potentially eventually gridlock the contract	✓ RESOLVED
18	LOW	Vesting cycle business logic appears unnecessary	✓ RESOLVED
19	INFO	Typographical errors and gas optimizations	✓ RESOLVED

1.3.6 TeamContract

ID	Severity	Summary	Status
20	HIGH	The contract will fully grid-lock and prevent all tokens from ever being withdrawn if no unlock has been made for just three days due to a literal being coerced into a uint8, causing an overflow reversion	✓ RESOLVED
21	HIGH	An unexpectedly high amount of tokens can be allocated to team members, exceeding the actual contract balance, preventing them from withdrawing as the token transfer would fail	✓ RESOLVED
22	LOW	`WithdrawToMember` function does not follow the checks-effects-interactions pattern, which would allow a single member to drain the contract if the itoken contract allows for reentrancy	✓ RESOLVED
23	INFO	Unclear `calc` function name and parameters	✓ RESOLVED
24	INFO	Contract owner is added as a member without any validation check	✓ RESOLVED
25	INFO	Typographical errors and gas optimizations	✓ RESOLVED

2 Findings

2.1 Global Issues

The issues in this section apply across several contracts within the protocol. We have consolidated them into one section to keep the report more readable.

2.1.1 Issues & Recommendations

Issue #01	`itoken` is private
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 throughout all contracts in the codebase.
Resolution	RESOLVED `itoken` is now `public` on every contract.

Issue #02	The contract pragmas are extremely wide
Severity	INFORMATIONAL
Location	`pragma solidity >=0.4.22 <0.9.0;`
Description	It is unclear to a reader of the repository what version the contracts will be deployed on. We ourselves had to ask the developer which version they were planning to deploy on, as there are very different types of issues between these wide ranges of solidity versions.
Recommendation	Consider fixing the pragma to a specific version, eg.: `pragma solidity =0.8.13;`
Resolution	RESOLVED The pragma has been specified as `pragma solidity ^0.8.9` or `pragma solidity =0.8.7`.

Issue #03	Typographical errors and gas optimizations
Severity	INFORMATIONAL

Description	We have consolidated the typographical errors and the sections which can be further optimized for gas usage below.
	
	`import "@openzeppelin/contracts/token/ERC20/ERC20.sol";`
	
	Since we do not actually need ERC20 and only its interface, it is sufficient to import `IERC20` instead throughout the codebase.
	
	`event TransferSent(address _from,address _destAddr,uint _amount);`
	
	The first parameter of this event can be removed as its redundant since the address is known, and the second parameter can be indexed. Additionally, this event uses a `uint` parameter while the rest of the codebase uses `uint256`. Although it is not a problem, it is inconsistent and we recommend being consistent.
	
	`modifier onlyOwner() {`
	
	It is best practice to put modifiers above the functions of the contract — typically these are never written inline within the function logic. This modifier can be removed altogether once moved to OpenZeppelin's Ownable.
	
	The codebase is also presently badly formatted. We recommend running the codebase through an automated formatter before deploying it as unformatted code can be a a turn off for readers.
	
	Presently the `token` and `itoken` values which are set in the construction of most of the contracts can almost always be marked as `immutable` to save gas.

Recommendation	Consider fixing the typographical errors and implementing the recommended gas optimizations.
Resolution	PARTIALLY RESOLVED Some of these issues have been resolved.

Issue #04	Contracts do owner management manually
Severity	INFORMATIONAL
Description	Presently the contracts all do their own owner management — this is unnecessarily verbose as OpenZeppelin provides an Ownable library.
Recommendation	Consider removing the owner logic from the various contracts and instead extending OpenZeppelin's Ownable.
Resolution	RESOLVED All the contracts now use the OpenZeppelin implementation.

Issue #05	Lack of `safeTransfer` usage
Severity	INFORMATIONAL
Location	`itoken.transfer(_address,amountConverted);`
Description	In the codebase, the `transfer` method is used to transfer tokens from various contracts 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` (from the SafeERC20 library) instead of `transfer` as is done throughout most of this contract. `itoken.safeTransfer(_address,amountConverted);`
Resolution	RESOLVED All the contracts now use `safeTransfer`.

2.2 AdvisorsContract, ReserveFund and PlayerRewardsContract

The `AdvisorsContract`, `ReserveFund` and `PlayerRewardsContract` are strictly vesting contracts which put the recipients of the tokens within the contract on a strict vesting schedule which cannot be re-configured once all tokens have been allocated to recipients.

These contracts are used to manage the vesting of tokens to a whitelisted set of users. The owner can add new users to the whitelist as long as there are tokens remaining to allocate (no tokens can be allocated to multiple recipients). Recipients vest tokens based on how many days have passed since the start of the vesting.

The three contracts have the following schedules:

- For the `AdvisorsContract`, there is a 12 month cliff, followed by a linear daily vesting for 24 months.
- For the `PlayerRewardsContract`, there is a 3 month cliff, followed by a linear daily vesting for 48 months.
- For the `ReserveFund`, there is a 6 month cliff, followed by a linear daily vesting for 6 months.

All of these contracts use identical code except for the durations which are different for each contract.

Note that throughout the codebase, years are assumed to be 360 days and not 365 days. We assume this has been done to ease calculations as months are always 30 days this way and therefore we did not raise an issue for this. Recipients should of course keep this in mind as their schedule will not strictly align with calendar months.

2.2.1 Privileged Functions

- `addWhiteList [ owner ]`
- `transferOwnership [ owner ]`
- `Withdraw [ whitelisted users ]`

2.2.2 Issues & Recommendations

Issue #06	**Contract will likely gridlock and prevent any further claiming with a large amount of users (~90+ recipients)**
Severity	HIGH SEVERITY
Location	Line 73 `vestingCycles = vestingCycles + daysPast;`
Description	There is an error within the codebase as the days that have passed are aggregated for each user into a global variable. This causes this global variable to not only lose all of its meaning, it also causes it to increase very quickly and more rapidly with the more users and days that have passed. Since `vestingCycles` is only a `uint16`, it will break due to overflow after reaching 65536. Given that every day for every recipient is added to this, it would only take about 90 recipients to eventually cause line 73 to overflow and prevent any further claims from happening.
Recommendation	Consider moving to `uint256` for all variables, consider rethinking the purpose of `vestingCycles`. The business logic requirement that caused this variable to be introduced can be solved in a simpler manner. Consider removing the whole vesting cycle logic and instead initializing users' `lastUnlockTime` at the appropriate time within the `addWhiteList` function: `lastUnlockTime[user] = 1665243000 + 360 days;` Through this method, everyone can simply start claiming after 360 days since the start time without any statefull behavior (no need for `constUnlockTime`, no need for `vestingCycles` logic). The contract becomes significantly simpler and less error prone.
Resolution	RESOLVED All variables are now `uint256` and the `vestingCycles` variable has been removed.

Issue #07	**Recipients who claim their whole allocation at the end of the vesting will receive less than expected and mess up the contract logic due to an overflow exception**
Severity	MEDIUM SEVERITY
Location	Line 99 `uint8 daysPast = uint8((block.timestamp - lastUnlockTime[msg.sender]) / 60 / 60 / 24);`
Description	The contract calculates the days that have passed since the last vest of the recipient (or the start time). This result is then cast to an 8 byte unsigned integer. As such integers can at most reach 255, this cast overflows silently if a recipient does not claim for a longer period. In that case, the recipient receives significantly less tokens than were expected and various aspects of the contract malfunction.
Recommendation	Consider strictly using `uint256` exclusively throughout the codebase to avoid casting issues. As a side-note, consider also dividing by "1 days" to make the code a bit more readable.
Resolution	RESOLVED All variables are now `uint256`.

Issue #08	Contract can theoretically be re-initialized once everyone withdraws
Severity	LOW SEVERITY

Location	Line 38-42
	`if (balance == 0){`
	`  uint256 _balance = itoken.balanceOf(address(this));`
	`  balance = _balance;`
	`  maxBalance = _balance;`
	`}`

Description	The above check is done whenever a new user is whitelisted to initialize the contract variables. However, whenever users claim tokens, the claim amount is deducted from balance again. This causes this check to be callable again once all tokens have been claimed. In this scenario, new users would become addable but the contract would not be able to deal with them properly, eg. the following line would underflow and revert: `require(maxBalance-allOwned >= amountConverted, "not enough BFG available to send.");` It is therefore better to explicitly prevent this scenario.
Recommendation	Consider guarding the if-statement with the `maxBalance` variable: `if (maxBalance == 0){`
Resolution	RESOLVED The if statement now uses `maxBalance`.

Issue #09	**Claiming delays a user their vesting schedule by up to 24 hours due to rounding**
Severity	LOW SEVERITY
Description	Users receive tokens for every 24 that pass. However, if someone for example claims after 47 hours, they would receive a claim for a single day while having to wait another 24 hours until their next claim. In this example, they lose 23 hours. In the extreme worst case where someone always harvests at the near end of the second day, they would need to wait twice as long as someone who always harvests at the start of the second day.
Recommendation	Consider incrementing `lastUnlockTime` with `daysPast * 1 days` instead of resetting it.
Resolution	RESOLVED `lastUnlockTime` is incremented with `daysPast * 1 days`.

Issue #10	**Recipients can instantly unlock their whole allocation through a reentrancy exploit if the token allows for reentrancy**
Severity	LOW SEVERITY
Description	The `getDaysUnlocked` function does not adhere to checks-effects-interactions, which means that various values are only updated after the token is transferred to the recipient. If this token transfer would allow the recipient to execute code, eg. with ERC777 tokens, this would allow the recipient to reenter and claim again, without their last claim time having been updated yet. Through such a reentrancy exploit, the malicious recipient can instantly claim their whole locked amount. This issue has been marked as low severity as we assume that the project will be using a simple token, and not something like ERC-777. However, the checks-effects-interactions pattern is crucial and we ourselves never violate it as it prevents so many exploits. We therefore definitely still recommend the client to fix this issue regardless of which token they are using.
Recommendation	Consider moving to checks-effects-interactions, this is as simple as moving the token transfer to the bottom of the function.
Resolution	RESOLVED `withdraw` now adheres to the checks-effects-interactions pattern.

Issue #11	Typographical errors
Severity	INFORMATIONAL

Description	We have consolidated the typographical errors into a single issue to keep the report brief and readable.
	
	Line 18
	`mapping(address => uint256) public whiteList;`
	
	This mapping appears to be completely unused. A bool type would also be more appropriate than an integer.
	
	Line 31
	`constUnlockTime = 1665243000;`
	
	This can be set at the top of the contract, directly with the declaration.
	
	Line 32-33
	`vestingCycles = 0;`
	`allOwned = 0;`
	
	These two declarations are unnecessary and can be removed, since these values are the defaults.
	
	Line 37
	`uint256 amountConverted = amount * 1000000000000000000;`
	
	Typically, since functions are not called by users, contracts just take a raw "wei" input without converting it by 1e18. In our opinion, doing this conversion on the frontend/scripts suffices, as is common practice. Additionally, readability can be improved by using 1e18 or 1 ether instead of the large number. It should be noted that this current logic might also cause issues with tokens with different decimal numbers.

Line 57

```
function getDaysUnlocked(uint8 daysPast, address _receiver)
internal{
```

This function name is misleading as it transfers the tokens as well. It should likely be renamed to increase auditability. We were surprised to see it do a lot of different things but the thing it's said to do.

The `getDaysUnlocked` function is quite wasteful with gas as it reads some variables many times from storage, eg. `lockedBFG`.

Line 99

```
uint8 daysPast = uint8((block.timestamp -
lastUnlockTime[msg.sender]) / 60 / 60 / 24);
```

This can be simplified by dividing by 1 days instead.

Line 100

```
require(daysPast > 0, "Too early for unlock");
```

This can be moved to an assertion due to the requirement above it which prevents this from ever actually failing.

Various functions lack events: `addWhiteList`, `Withdraw` in the early return case and `transferOwnership`.

Finally, `addWhiteList`, `Withdraw` and `transferOwnership` can be made external.

Recommendation Consider fixing the typographical errors.

Resolution RESOLVED

Most of the recommendations have been implemented.

Issue #12	**Division before multiplication increases rounding error slightly within the function which calculates the amount of tokens to receive**
Severity	INFORMATIONAL

Description	Within `getDaysUnlocked`, division occurs before multiplication when calculating the total reward for a user:
	`uint256 newTokens = ownedBFG[_receiver] * 139 / 100000;`
	`uint256 calTokens = newTokens * daysPast;`
	This causes the rounding error from the division to be amplified by `daysPast` unnecessarily.
	It should be noted that this rounding error is still extremely minor and will likely remain unnoticable.

Recommendation	Consider rewriting this into a single setter: `uint256 calTokens =  ownedBFG[_receiver] * daysPast * 139 / 100000;` 139 and 100000 could also be made constants to improve readability and forkability.
Resolution	RESOLVED

2.3 SeedContract and PrivateRoundContract

`SeedContract` and `PrivateContract` are nearly identical to the vesting contracts in the previous section. They only differ in the fact that the whitelisted users receive an instant TGE unlock as soon as the contract owner whitelists them.

Whitelisted users receive 10% of their allocation when they are whitelisted (considered as the Token Generation Event).

Both have a 6 month cliff, followed by a linear daily vesting for 12 months.

These two contracts are extremely similar to the `AdvisorsContract`, `ReserveFund` and `PlayerRewardsContract` from a logical and implementation perspective. All issues mentioned in the three previous contracts equally apply here, and should be resolved here as well. We will not repeat them in order to keep the report short and readable.

Note: During the resolution round, `PrivateContract` was renamed to `PrivateRoundContract`.

2.3.1 Privileged Functions

- `addWhiteList [ owner ]`
- `transferOwnership [ owner ]`
- `Withdraw [ whitelisted users ]`

2.3.2 Issues & Recommendations

Issue #13	**`addWhiteList` does not adhere to the checks-effect-interaction pattern and the owner could potentially withdraw more tokens than expected on TGE**
Severity	LOW SEVERITY

Location	Line 50~
	`itoken.transfer(user,TGE);`
	`balance-=TGE;`
	
	`allOwned += amountConverted;`
	`ownedBFG[user] += amountConverted;`
	`lockedBFG[user] += (amountConverted - TGE);`
	`//starting 8.10 17:30`
	`lastUnlockTime[user] = 1665243000;`
	
	`whiteList[user] = 1;`

Description	All the user values should be updated before the token transfer to follow the checks-effect-interaction pattern. If this is not done, the owner could self whitelist and send tokens multiple times through a reentrancy attack by calling `addWhiteList` iteratively on the token transfer hook (this is possible due to `allOwned`, which is used in a safeguard, only being incremented after the token transfer). Note that this issue has only been marked as low severity as we expect that the client will not deploy this contract with a reentrancy-vulnerable token. However, the CEI pattern is absolutely vital to high-quality code and we recommend the client to resolve this regardless of them using a reentrancy-prone token or not. If the client is using an ERC-777 token or similar, this must definitely be resolved.
Recommendation	Consider moving the token transfer at the end of the `addWhiteList` function.
Resolution	RESOLVED `addWhitelist` now adheres to the checks-effects-interactions pattern.

2.4 AirdropContract and LiquidityContract

`AirdropContract` is a simple utility contract which allows the contract owner to distribute tokens to any recipient. Tokens sent are taken from the contract balance.

The contract exposes a `maxBalance` variable which denotes the initial token balance before distribution starts. Once the current balance becomes insufficient for someone to receive their full airdrop amount, they will still receive whatever balance is left in the contract.

`LiquidityContract` follows the same logic as `AirdropContract` but the allocate function is named `Withdraw`. Although the implementations are identical, we expect that the `LiquidityContract` will be used for a completely different purpose than the `AirdropContract` will be used for. Given the name of the contract, the `LiquidityContract` will likely be used to hold the funds for eventually adding liquidity to a token pair.

2.4.1 Privileged Functions

- `init`
- `allocate [ AirdropContract ] / Withdraw [ LiquidityContract ]`
- `transferOwnership`

2.4.2 Issues & Recommendations

Issue #14	`init` function logic is also executed within allocate and therefore seems redundant
Severity	INFORMATIONAL
Description	The init function is used to initialize the `maxBalance` and `balance` variables. However, this is also done at the beginning of the allocate function.
Recommendation	Consider refactoring `init` to avoid code repetition by re-using an internal function in both locations (or removing init altogether if it's not necessary). In addition, it seems unnecessary to store the contract balance into a storage variable. This could also be done using a `getBalance` view function if it is solely done for UI purposes. Consider also fixing all this within the `EcosystemFundContract`.
Resolution	RESOLVED The `init` function has been refactored.

Issue #15	`token` is never set
Severity	INFORMATIONAL
Location	Line 11 `address public token;`
Description	The contract defines a token variable which is never set — this value is therefore redundant and can be removed. There is no point in keeping it given that `itoken` has the same purpose.
Recommendation	Consider fixing removing `token`.
Resolution	RESOLVED `token` has been removed.

Issue #16	Typographical errors and gas optimizations
Severity	INFORMATIONAL

Description	We have consolidated the typographical errors and the sections which can be further optimized for gas usage below.
	
	Line 13
	`IERC20 itoken;`
	
	The `itoken` can be marked as immutable to save on gas usage.
	
	Line 38-39
	`require(amount > 0 , "Need to request more than 0 BFG");`
	`require(balance > 0 , "No more BFG to collect");`
	
	These requirements can be moved up to the top of the function (right below the `balanceOf` call) to revert earlier and save gas in case of reversion. The second check should also use `newBalance` instead to save on gas as reading from memory is cheaper (line 43-45 need to be adjusted as well).
	
	Line 41
	`uint256 amountConverted = amount * 1000000000000000000;`
	
	Typically, since users do not really directly interact with the codebase, we tend to not scale inputs. This means if someone wants to grant 1e18 tokens (a single real token), they just provide 1e18 into the function parameters. You can therefore remove the 1e18 multiplier. Apart from this, people tend to not write large numbers like the one above. Instead, they tend to write the number as either 1e18 or 1 ether. This is more readable. It should be noted that this current logic might also cause issues with tokens with different decimal numbers.
	
	Line 48
	`balance-= amountConverted;`
	
	`balance` is stored twice within the allocate function, this unnecessarily wastes gas as it only really needs to be stored once.

	`init` and `transferOwnership` are presently both missing events. Though we recommend simply moving to Ownable by OpenZeppelin for the latter, which of course emits an event. Another gas optimization which could be considered is having an `allocateMulti` function which allocates to multiple users at once. This would save some gas as the ownership and `balanceOf` calls would only need to be made once. Finally, `init`, `allocate` and `transferOwnership` can be marked as external to communicate to the reader that these are not used internally.
Recommendation	Consider fixing the typographical errors. Consider also fixing all these issues within the `EcosystemFundContract`.
Resolution	RESOLVED Most of the recommendations have been implemented.

2.5 EcosystemFundContract

`EcosystemFundContract` is used to manage ecosystem funds. The funds deposited into the contract are locked for the first three months. After that, the owner of the contract can withdraw funds at any time and send them to any recipient using the `withdraw` function.

The contract exposes a `vestingCycles` variable that accounts for the number of withdrawals, and a `lastUnlockTime` that stores the timestamp of the latest withdrawal.

`EcosystemFundContract` is extremely similar to the `AirdropContract` from a logical and implementation perspective. All issues mentioned in the `AirdropContract` equally apply here, and should be resolved here as well. We will not repeat them in an effort to keep the report short and readable.

2.5.1 Privileged Functions

- `init`
- `withdraw`
- `transferOwnership`

2.5.2 Issues & Recommendations

Issue #17	**Using `uint16` for the `vestingCycles` variable type could potentially eventually gridlock the contract**
Severity	MEDIUM SEVERITY

Description	A counter called `vestingCycles` which increments on any withdrawal in defined in the codebase.
	
	`Line 15`
	`uint16 public vestingCycles;`
	
	However, as this type is incremented on any withdrawal, this means that the increment will fail due to overflow after being called 65,535 times. This is severe since there is no other way to withdraw any funds from this contract but calling `withdraw`.
	
	Once withdrawals potentially start eventually failing due to overflow, all funds still present in the contract are stuck forever.
	
	Additionally, using `uint16` for a storage variable outside of a struct has no advantage since the whole slot needs to be stored and loaded still. Using `uint256` instead in fact saves gas as no conversions need to be made!

Recommendation	Consider using `uint256` instead.
Resolution	RESOLVED `vestingCycles` has been removed.

Issue #18	Vesting cycle business logic appears unnecessary
Severity	LOW SEVERITY
Description	Although the basic idea behind the contract is extremely simple, the code still has some complexity. This is because the code does special accounting with `vestingCycles` and various variables like a `lastUnlockTime`. That being said, the only requirement which is really being written with all those lines of code is "the owner cannot unlock for the first 90 days after the initial `lastUnlockTime`". This requirement does not need dozens of lines of code and could be written in a single line of code.

Recommendation	The vesting cycles logic seems to give a sense of there being multiple cliff unlocks. This is not true as after the first 90 days cliff, anything can be withdrawn freely at any time by the owner. We can therefore simplify the `withdraw` function significantly to account for this:
	
	`function withdraw(address _address, uint256 amount) external onlyOwner {`
	`  require(block.timestamp > unlockTime, "Too early for unlocking tokens");`
	`  uint256 balance = itoken.balanceOf(address(this));`
	`  require(amount > 0 , "Need to request more than 0 BFG");`
	`  require(balance > 0 , "No more BFG to collect");`
	
	`  if(amount > balance){`
	`  amount = balance;`
	`  }`
	
	`  lastUnlockTime = block.timestamp;`
	`  vestingCycles++;`
	`  emit TransferSent(_address,amount);`
	
	`  itoken.safeTransfer(_address, amount);`
	`}`

	It should be noted that this recommended ordering also writes the contract in checks-effects-interactions, which is of course desired. The client can also remove `lastUnlockTime` and `vestingCycles` completely, as we believe these are remnants from forking the `TeamContract`.
Resolution	RESOLVED The logic of the `withdraw` function has been simplified.

Issue #19	Typographical errors and gas optimizations
Severity	INFORMATIONAL

Description	We have consolidated the typographical errors and the sections which can be further optimized for gas usage below.
	
	Line 11
	`address public token;`
	
	As `token` does in fact have a value here, it could be made immutable to save on gas when it is being called. It can alternatively be removed as it is redundant with `IToken`.
	
	Line 23
	`lastUnlockTime = 1665243000;`
	
	This initialization does not need to occur in the constructor as it can instead be moved to the top of the contract, as is common practice (it might make sense to set this value to `block.timestamp` or a constructor parameter though): `uint256 public lastUnlockTime = 1665243000;`
	
	Line 24
	`vestingCycles = 0;`
	
	This is unnecessary as the value is already zero on deployment.
	
	Line 34
	`function Withdraw(address _address, uint256 amount) public onlyOwner{`
	
	This function is capitalized which is not considered best practice within Solidity. We recommend starting function names with a lowercase.

Line 48

```
lastUnlockTime = lastUnlockTime + 90 days;
```

Throughout the withdrawal function, `lastUnlockTime` is read from memory several times resulting in excessive gas usage. It could be cached in memory to save on gas. We also do not understand the value of shifting the amount here as nothing is really transferred at this moment.

Line 53

```
if(vestingCycles > 0)
```

This `if` statement could be simplified to an `else` statement to make the code more readable and save on gas usage.

Finally, `Withdraw` presently does not emit an event if it returns in the "3 months cliff" section.

Recommendation	Consider fixing the typographical errors and gas optimizations.
Resolution	RESOLVED All these issues have been resolved.

2.6 TeamContract

`TeamContract` is used to distribute tokens to the different team members. After deployment of the contract, the owner will be able to add up to 6 other members and set their shares. The owner of the contract will also be added by default as a member, on the contract creation.

Once the 6 members have been added and the total shares distributed are equal to 100% and no one will ever be able to add additional recipients, the contract is considered configured.

The distribution of the vested tokens will start after a vesting cliff of 12 month. After this period, tokens will be unlocked daily for a total period of 32 month, meaning 0,104% of the total team team allocation will be unlocked daily. Any team member will be able to call Unlock after this 1 year cliff period to unlock tokens for the whole team. These tokens will then become claimable by the team members individually when they call the withdrawal function themselves.

2.6.1 Privileged Functions

- `init`
- `AddMember`
- `transferOwnership`
- `WithdrawToMember [OnlyMember]`
- `Unlock [OnlyMember]`

2.6.2 Issues & Recommendations

Issue #20	**The contract will fully grid-lock and prevent all tokens from ever being withdrawn if no unlock has been made for just three days due to a literal being coerced into a `uint8`, causing an overflow reversion**
Severity	HIGH SEVERITY

Location	Lines 101-104
	`uint8 daysPast = uint8((block.timestamp - lastUnlockTime) / 60 / 60 / 24);`
	`require(daysPast > 0, "Too early for unlock");`
	`calc(104 * daysPast, daysPast * 1 days);`

Description	During each `Unlock` call, the code calculates how many days have passed since the last call and grants emissions to the members according to the time which has passed, based on a factor of 0.104% per day which passes. Although this logic has been implemented correctly, we noticed that the client unnecessarily limits the `daysPast` to a `uint8` which would cause the function to malfunction if no claim occurs for 256 days. Given that not claiming for so long is rather unlikely, this would have only been a lower severity issue. However, after more thorough testing and inspection by our team, it became clear that the issue is much more serious: the whole contract permanently breaks after just three days of not claiming anything! The reason for this being the case is because of the multiplication being passed into the first parameter of the `calc` call on line 104. The Solidity documentation states: *Number literal expressions are converted into a non-literal type as soon as they are used with non-literal expressions.* This means that as soon as we see `104 * daysPast` in the codebase, `104` will be coerced into a `uint8` and the multiplication will occur within the `uint8` space.

	Since the `uint8` space is so limited, this multiplication will already revert through an overflow panic as soon as 3 days have elapsed. This means that if the team ever, just for a single time, forgets to claim for just a few days, the contract funds would be permanently and irrevocably stuck forever. The only way to prevent this issue in the current codebase is by consistently and diligently claiming every single day.
Recommendation	Consider moving away from all "small types" completely. These do not save any gas and therefore do not bring any benefit to the codebase. Instead, they make the codebase more expensive and more complex. Worst of all, they introduce errors like the one explained above. Once all variables are `uint256`, this issue no longer presents itself. We also definitely recommend changing the `calc(uint256 x, uint256 y)` function to `calc(uint256 daysPassed)` as presently the signature of that function is unnecessarily cryptic and redundant. It should be noted that the second parameter in `calc` would overflow after 195 days. This is because 1 days gets coerced to a minimum of `uint24`, causing the multiplication to occur in 24 bits since it is the largest value between the two types. This multiplication then overflows after 195 days.
Resolution	 All the variables are now `uint256`.

Issue #21	**An unexpectedly high amount of tokens can be allocated to team members, exceeding the actual contract balance, preventing them from withdrawing as the token transfer would fail**
Severity	HIGH SEVERITY

Location	Line 96-99
	`if (balance == 0){`
	`  uint256 newBalance = itoken.balanceOf(address(this));`
	`  balance = newBalance;`
	`}`

Description	This part of the code updates the balance variable when it reaches zero, meaning that all tokens have been allocated, but not that they have been distributed, as members can claim their tokens separately using `claim`. Thus, when reaching the end of the vesting schedule, `balance` would reach zero and on the next `unlock` call, it would be updated to the amount of tokens still unclaimed. This would add the same tokens a second time into the member's balances. While the first members to claim would receive more tokens than expected, the last ones to claim would not be able to claim anything, as their allocation would exceed what is left on the contract as rewards.
Recommendation	Consider removing this piece of code as there is no apparent utility for it.
Resolution	RESOLVED The code has been removed.

Issue #22	**`WithdrawToMember` function does not follow the checks-effects-interactions pattern, which would allow a single member to drain the contract if the `itoken` contract allows for reentrancy**
Severity	LOW SEVERITY

Location	Line 67
	`itoken.transfer(msg.sender,balances[msg.sender]);`
	`balances[msg.sender] = 0;`

Description	The balance of the message sender should be updated before the token transfer to follow the checks-effect-interaction pattern. If this is not done, the recipient of these tokens can claim multiple times through a reentrancy attack by calling `WithdrawToMember` iteratively on the token transfer hook. Note that this issue has only been marked as low severity as we expect that the client will not deploy this contract with a reentrancy-vulnerable token. However, the CEI pattern is absolutely vital to high-quality code and we recommend the client to resolve this regardless of them using a reentrancy-prone token or not. If the client is using an ERC-777 token or similar, this must definitely be resolved.

Recommendation	Consider moving the balance update above the token transfer:
	
	`function WithdrawToMember() public onlyMember {`
	`  uint256 amount = balances[msg.sender];`
	`  require(amount > 0,"Not enough unlocked tokens");`
	`  balances[msg.sender] = 0;`
	`  emit TransferSent(msg.sender, amount);`
	`  itoken.safeTransfer(msg.sender, amount);`
	
	Note how the function is also made more gas-efficient by only reading from storage once.

Resolution	RESOLVED The `WithdrawToMember` function, now called `claim`, adheres to the checks-effects-interactions pattern.

Issue #23	Unclear `calc` function name and parameters
Severity	INFORMATIONAL

Location	Line 86 and 104
	`calc(0, 360 days);`
	`calc(104 * daysPast, daysPast * 1 days);`

Description	The calc function name is quite cryptic and does not say what it is doing outright. Additionally, its two parameters x and y are both derived from `daysPast`, which could be used as the single input of the function.
Recommendation	Consider renaming the `calc` function and using a single parameter: `function _calculatePending(uint16 daysPast) internal {`
Resolution	RESOLVED `calc` has been renamed as `_calculatePending`.

Issue #24	Contract owner is added as a member without any validation check
Severity	INFORMATIONAL

Location	Line 27-29
	`members.push(_owner);`
	`totalShares = share;`
	`shares[_owner] = share;`

Description	In the constructor, the deployer address is added as a team member. The same logic as `AddMember` is implemented but with none of the validation checks done in the function.
Recommendation	Consider moving the logic for adding a member into a private function `_addMember` that would be called by the constructor and by the external function `AddMember`.
Resolution	RESOLVED The code has been refactored into a `_addMember` function.

Issue #25	Typographical errors and gas optimizations
Severity	INFORMATIONAL
Description	We have consolidated the typographical errors and the sections which can be further optimized for gas usage below.

Line 13

```
mapping(address => uint8) public shares;
```

`shares` values are stored as `uint8`. Storage values that are not structs are always taking the same 256 bits storage slot, and storing smaller variables has no effect. Since it increases the risk of overflows and actually increases gas, it is always better to use `uint256` in these cases.

Line 15

```
address[] public members;
```

This array needs a length getter, otherwise the only way of fetching all the members from the exterior is to loop on the members until it fails. There is already a length function, but if the contract is initialized with say 5 members summing to 100%, this length function would be wrong.

Line 16

```
uint public vestingCycles;
```

The code both uses `uint` and `uint256` for the same integer type. Even though it is not a problem at all, it is rather inconsistent to not stick to a single notation, preferably `uint256`.

Line 30

```
lastUnlockTime = 1665243000;
```

This can be moved to the variable initialization, line 9.

Line 31

```
vestingCycles = 0;
```

This is not necessary as `vestingCycles` is declared with no initial value.

Lines 41, 64 and 74

```
function AddMember(address member,uint8 share)
function WithdrawToMember() public onlyMember
function Unlock() public onlyMember
```

The `AddMember`, `WithdrawToMember` and `Unlock` functions should be in camelcase. They should also be declared as `external`, and emit an event. We also believe that `WithdrawToMember` would be more appropriately named `claim`, as it does not allow you to withdraw to another member, only claim for yourself.

Line 45

```
require(members.length <= memberLength-1,"All team members added");
```

This could be written slightly more efficiently and readable by doing `members.length < memberLength`.

Line 53

Some code has been commented out and should be removed.

Lines 78 and 90

```
if (maxBalance <= 0){
if (balance <= 0)
```

As `balance` is an `uint`, this can just be `balance == 0` as it can never go negative. The same applies for `maxBalance`.

Line 94

```
//unlock 3.5% each month
```

The actual monthly unlock is more like 3.125%, this comment is therefore incorrect.

Line 96

```
if (vestingCycles > 0){
```

This clause is unnecessary as an early return already occurred if it is not the case. It is therefore impossible to break this requirement and thus it can be removed to save gas and simplify the contract.

Line 97

```
if(lastUnlockTime == 1665243000){
   lastUnlockTime= 1665243000 + 360 days;
}
```

As the `lastUnlockTime` is updated on the first unlock, this will never be true and can be removed.

Line 101

```
uint8 daysPast = uint8((block.timestamp - lastUnlockTime) /
60 / 60 / 24);
```

This can be simplified to `uint256 daysPast = (block.timestamp - lastUnlockTime) / (1 days);`.

Line 102

```
require(daysPast > 0, "Too early for unlock");
```

This seems to not be breakable due to an earlier requirement and could be replaced with an assertion to signal the fact that this is always the case.

Line 115

```
for (uint8 i = 0; i < members.length; i++)
```

Caching `members.length` outside of the for loop in a memory value is more gas efficient.

Line 123

```
if(x==0){
    lastUnlockTime += y;
    vestingCycles ++;
}
```

This could be put in an `else` statement.

	Throughout the contract, magic values are often being used ("100", "104", "100000"...), if the client ever changes some of the business logic, they might forget to adjust these magic values. Consider instead making them constants at the top of the contract to explicitly re-use values like "100%" and "0.104%". Within `WithdrawToMember`, the `msg.sender` balance is often re-used and should be cached to save gas.
Recommendation	Consider fixing the typographical errors and implementing the gas optimizations.
Resolution	 Most of the recommendations have been implemented.


PALADIN
BLOCKCHAIN SECURITY