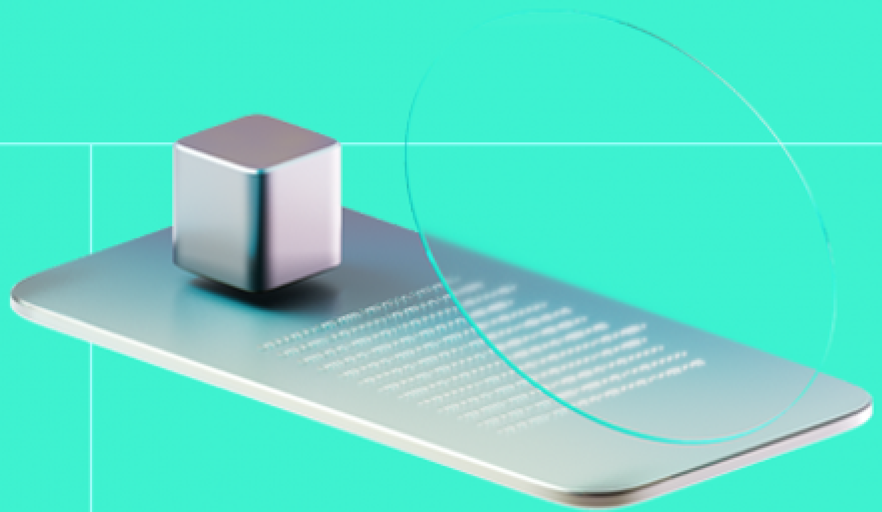
Smart Contract Code Review And Security Analysis Report



Customer: Gitswarm

Date: 08/05/2024

We express our gratitude to the Gitswarm team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

GitSwarm is a platform designed to bring together developers, project owners, and investors in a collaborative environment for creating and investing in innovative token-governed projects. It serves as a versatile ecosystem where freelancers can showcase their skills and bid on tasks, project owners can find talented individuals to bring their ideas to life, and investors can discover and support emerging projects before they become mainstream.

Platform: EVM

Language: Solidity

Tags: Voting, Governance, ERC20, Upgradeability, Proxy

Timeline: 30/04/2024 - 06/05/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/GitSwarmOrg/contracts
Commit	a07999c

Audit Summary

10/10	10/10	100%	10/10
Security score	Code quality score	Test coverage	Documentation quality score

Total 10/10

The system users should acknowledge all the risks summed up in the risks section of the report

5	3	1	1
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Severity	Count
Critical	0
High	0
Medium	2
Low	3

Vulnerability	Status
F-2024-2204 - Missing zero address validation	Mitigated
F-2024-2353 - Proposals can be executed with spam votes	Accepted
F-2024-2218 - Unfinished code can result in too short voting periods	Fixed
F-2024-2226 - Missing checks on transferring non-standard ERC20 tokens	Fixed
F-2024-2270 - Missing storage gaps in upgradeable contracts	Fixed

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for Gitswarm
Audited By	David Camps Novi, Viktor Lavrenenko
Approved By	Przemyslaw Swiatowiec
Website	https://gitswarm.com
Changelog	08/05/2024 - Preliminary Report; 22/05/2024 - Final Report

Table of Contents

System Overview **6**
Privileged Roles 6
Executive Summary **8**
Documentation Quality 8
Code Quality 8
Test Coverage 8
Security Score 8
Summary 8
Risks **9**
Findings **10**
Vulnerability Details 10
Observation Details 24
Disclaimers 39
Appendix 1. Severity Definitions **40**
Appendix 2. Scope **41**

System Overview

GitSwarm is a project that allows users to create DAO projects with their voting tokens. It contains the following contracts:

- Common.sol - this contract integrates shared logic and initializations.
- Constants.sol - defines immutable constants used across the GitSwarm contracts, which includes the burn address and numerical representations of proposal types.
- ERC20Base.sol - this contract is a standard ERC20 token implementation. It provides a solid foundation for the deployment of either `ExpandableSupplyToken.sol` or `FixedSupplyToken.sol`, which include constructors that are missing in this base contract.
- ERC20Interface.sol - defines the standard functions for an ERC20 token, enabling interoperability across different platforms and contracts.
- ExpandableSupplyTokenBase.sol - this contract extends ERC20Base to support expandable token supplies. It allows
- Interfaces.sol - defines interfaces for `ContractsManager.sol`, `Delegates.sol`, `FundsManager.sol`, `GasStation.sol`, `Proposal.sol`, and `Parameters.sol`.
- MyTransparentUpgradeableProxy.sol - an implementation of the transparent proxy pattern for upgradable contract logic without losing state.
- SelfAdminTransparentUpgradeableProxy.sol - the same as MyTransparentUpgradeableProxy.sol with a key difference: this version initializes itself as its own admin.
- ContractsManager.sol - the contract manages governance-related functionalities including upgrades, handling of trusted addresses, modifications of voting tokens, and management of burn addresses through proposal mechanisms. It leverages OpenZeppelin's transparent upgradeable proxy pattern for upgradability.
- Delegates.sol - the contract allows participants to delegate and undelegate their voting power. Through this system, users can delegate their voting power to others whom they trust to vote on their behalf, enhancing the flexibility and reach of their influence within the project. Delegation spam management is handled in this same contract.
- ExpandableSupplyToken.sol - an ERC20 token, which can be used as a voting token with a supply that can be expanded via proposals.
- FixedSupplyToken.sol - an ERC20 token, which can be used as a voting token with a fixed supply.
- FundsManager.sol - the contract that handles the Ether and ERC-20 token funds for each project created on GitSwarm.
- GasStation.sol - the contract that allows projects to buy gas and to manage transfers to gas addresses through proposals.
- Parameters.sol - this contract manages the parameters and trusted addresses for projects. It allows for proposing and executing changes to parameters and trusted addresses.
- Proposal.sol - this contract enables token holders with sufficient project voting tokens to participate in the direction of projects through a structured proposal flow. This flow encompasses the creation, voting, contesting (optional), and execution phases of proposals.
- UpgradeableToken.sol - the contract which is used for the GitSwarm token. It is the same as `ExpandableSupplyToken.sol`, with the key distinction being its upgradeability.

Privileged roles

- The project has only one role which is a trusted address. Any address from the following list can be the trusted address:
  - the addresses of the contracts: Delegates.sol, FundsManager.sol, Parameters.sol, Proposal.sol, GasStation.sol, ContractsManager.sol.
  - all voting tokens of the created projects.
  - the array of trusted addresses that can be added via proposals.
- The trusted address can use the following project's functionality:
  - `FundsManager.sol::sendEther(), sendToken(),updateBalance()` - allows to send Ether/ERC20 from the project's balance or update the project's balance.
  - `Parameters.sol::initializeParameters()` - initialize parameters for a project.
  - `Proposal.sol::setActive(), setWillExecute(), createProposal(), deleteProposal()` - allows to change the status of a proposal manually. In addition to that, it enables trusted addresses to create or delete the proposals.

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation quality score is **10** out of **10**.

- Functional requirements are partially provided:
  - System roles are not described.
- Technical descriptions are sufficient:
  - Deployment instructions are defined.
  - Technical specification is provided.
  - The code has the necessary NatSpec comments.

Code quality

The total Code quality score is **10** out of **10**.

- The code adheres to the best practices.
- The Solidity Style Guide is followed.

Test coverage

Code coverage of the project is **100%** (branch coverage).

- Deployment and basic user interactions are covered with tests.

Security score

Upon auditing, the code was found to contain **0** critical, **0** high, **2** medium, and **3** low severity issues. Out of these, **3** issues have been addressed and resolved, leading to a security score of **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **10**. This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- The project utilizes Solidity version 0.8.20 or higher, which includes the introduction of the PUSH0 (0x5f) opcode. This opcode is currently supported on the Ethereum mainnet but may not be universally supported across other blockchain networks. Consequently, deploying the contract on chains other than the Ethereum mainnet, such as certain Layer 2 (L2) chains or alternative networks, might lead to compatibility issues or execution errors due to the lack of support for the PUSH0 opcode. In scenarios where deployment on various chains is anticipated, selecting an appropriate Ethereum Virtual Machine (EVM) version that is widely supported across these networks is crucial to avoid potential operational disruptions or deployment failures.
- The project allows users to use any ERC20/non-ERC20 tokens as a voting token, which can lead to unexpected behavior. This will affect new projects that are not going to use the possible ERC20 tokens created in the system scope (i.e. `FixedSupplyToken`, `ExpandableSupplyToken`, `UpgradeableToken`).
- Iterating over a dynamic array populated with custom length can lead to gas limit denial of service if the number of elements goes out of control. This scenario is possible in:
  - `Delegates::undelegateAddress, undelegateAllFromAddress.`
  - `Proposal:: getVoteCount, removeSpamVoters.`
- The parameter `creatorSupply` have no restrictions when initializing the project's ERC20 tokens and allows the token creator to hold up to 100% of the total token supply initially. Thus the creator controls the voting power of the corresponding project, being able to: sending tokens, changing burn and trusted addresses, etc. Such address can also control the token supply for those tokens that can mint more tokens. Additionally, those tokens may not be checked to be ERC20 compliant, as explained deeper in `F-2024-2355`.
- The addresses added into the `trustedAddresses` array hold the privilege to interact with several functions of the project (e.g. `setWillExecute`). This provides such addresses high power to affect the project outcomes. For example, a trusted address can send tokens at their will via `sendToken` and `sendEther`.
- Any address can be added into `trustedAddresses` as long as it is voted to be. Thus it is possible that a malicious address will receive this privilege.
- The addresses included in `burnAddresses` via `ContractsManager::executeProposal` can be updated to addresses controlled by some users to gain some extra tokens. For example, a user having control of the voting power, can use their own address as a burn address and avoid burning their tokens in `FundsManager:: reclaimFunds`.
- The voting token of a project can be changed after the project is started via `ContractsManager::proposeChangeVotingToken()`. Hence it is possible that users collected a certain token to vote in a project but it is later changed and such users would lose their voting power and token value.
- GitSwarm token holders have control of the upgrade process via `proposeUpgradeContracts`.
- The contracts of the project are `upgradeable`. As such, it is possible that new implementations are used that have changes from the audited code and may not be secure.

Findings

Vulnerability Details

F-2024-2218 - Unfinished code can result in too short voting periods - Medium

Description:

As part of the process of preparing the application for the production stage, it is important to address developer comments, especially `TODOs`. These comments often highlight missing functionalities or necessary validation checks that should be implemented.

The TODO comments were found in the following functions:

- `Parameters::initializeParameterMinMax()`
- `Parameters::internalInitializeParameters()`

The `TODO` comments found in the previous functions correspond to the necessity to update several time shifts to larger, feasible times. The current timestamps are too short to make the voting viable and should be updated.

Assets:

- Parameters.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Classification

Impact: 3/5

Likelihood: 5/5

Exploitability: Semi-Dependent

Complexity: Simple

Likelihood [1-5]: 5

Impact [1-5]: 3

Exploitability [0-2]: 1

Complexity [0-2]: 0

Final Score: 3.0 (Medium)

Severity: Medium

Recommendations

Remediation: Update the timestamps and remove the `TODO` comments.

Resolution: The timestamps in `initializeParameterMinMax` were updated, deleting the `TODO` comments too, in the commit `f830c0c`.

F-2024-2270 - Missing storage gaps in upgradeable contracts - Medium

Description: When working with upgradeable contracts, it is necessary to introduce storage gaps to allow for storage extension during upgrades. Storage gaps are a convention for reserving storage slots in a base contract, allowing future versions of that contract to use up those slots without affecting the storage layout of child contracts. Otherwise, it may be very difficult to write new implementation code. Without a storage gap, the variable in the child contract might be overwritten by the upgraded base contract if new variables are added to the base contract. This could have unintended and very serious consequences for the child contracts.

Assets:

- ContractsManager.sol [https://github.com/GitSwarmOrg/contracts]
- Delegates.sol [https://github.com/GitSwarmOrg/contracts]
- FundsManager.sol [https://github.com/GitSwarmOrg/contracts]
- GasStation.sol [https://github.com/GitSwarmOrg/contracts]
- Proposal.sol [https://github.com/GitSwarmOrg/contracts]
- UpgradableToken.sol [https://github.com/GitSwarmOrg/contracts]
- Parameters.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Classification

Impact: 5/5

Likelihood: 2/5

Exploitability: Semi-Dependent

Complexity: Simple

Likelihood [1-5]: 2

Impact [1-5]: 5

Exploitability [0-2]: 1

Complexity [0-2]: 0

Final Score: 2.7 (Medium)

Severity: Medium

Recommendations

Remediation: It is recommended to introduce the storage gaps in the affected contracts (i.e. base contracts of the upgradeable contracts).

To create a storage gap, declare a fixed-size array in the base contract with an initial number of slots. This can be an array of `uint256` so that each element reserves a 32 byte slot. Use the name `__gap`or a name starting with `__gap_` for the array so that OpenZeppelin Upgrades will recognize the gap.

To help determine the proper storage gap size in the new version of your contract, you can simply attempt an upgrade using `upgradeProxy` or just run the validations with `validateUpgrade` (see docs for Hardhat or Truffle). If a storage gap is not being reduced properly, you will see an error message indicating the expected size of the storage gap.

Resolution: The issue was fixed by implementing storage gaps in the `Common` and `ExpandableSupplyTokenBase` contracts, in the commit `f830c0c`.

F-2024-2204 - Missing zero address validation - Low

Description: In Solidity, the Ethereum address 0x0000000000000000000000000000000000000000 is known as the "zero address". This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address. The "
Missing zero address control" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.
For instance, a contract might allow tokens to be sent to the zero address without any checks, which essentially burns those tokens as they become irretrievable. While sometimes this is intentional, without proper control or checks, accidental transfers could occur.

The functions and parameters which lack zero address checks:

- `Common::_init() → _delegates, _fundsManager, _parameters, _proposal, _gasStation, _contractsManager.`
- `ERC20Base:: transferFrom() → _from, approve() → _spender.`
- `Proposal:: privateCreateProposal() → voterAddress.`
- `Parameters:: initialize() → _gitswarmAddress.`
- `GasStation:: proposeTransferToGasAddress() → to.`
- `ContractsManager:: proposeUpgradeContracts() → _delegates, _fundsManager, _parameters, _proposal, _gasStation, _contractsManager.`
- `FundsManager:: sendEther.`

Assets:

- Common.sol [https://github.com/GitSwarmOrg/contracts]
- ERC20Base.sol [https://github.com/GitSwarmOrg/contracts]
- ContractsManager.sol [https://github.com/GitSwarmOrg/contracts]
- FundsManager.sol [https://github.com/GitSwarmOrg/contracts]
- GasStation.sol [https://github.com/GitSwarmOrg/contracts]
- Proposal.sol [https://github.com/GitSwarmOrg/contracts]
- Parameters.sol [https://github.com/GitSwarmOrg/contracts]

Status: Mitigated

Classification

Impact: 2/5

Likelihood: 2/5

Exploitability: Semi-Dependent

Complexity: Simple

Likelihood [1-5]: 2
Impact [1-5]: 2
Exploitability [0-2]: 1
Complexity [0-2]: 0
Final Score: 1.7 (Low)

Severity: Low

Recommendations

Remediation: It is recommended to introduce checks in the aforementioned cases in order to avoid using the zero address.

Resolution: Zero address checks were implemented for token-receiving addresses in commit **2df1e28**. Other checks will be implemented off-chain, as reported by the development team:

> We have addressed the cases where this could result in an immediate loss of a lot of funds. The remaining cases only result in a waste of Gas and we plan to only address them on the front-end.

F-2024-2226 - Missing checks on transferring non-standard ERC20 tokens - Low

Description: The contract `FundsManager.sol` uses `require()` statements to handle the return boolean values of `transfer()` and `transferFrom()` methods of ERC20 tokens.

However, not all tokens follow the `ERC20` standard and can `return` no value at all. In such cases, all calls will `revert`, resulting in unexpected behavior.

The `FundsManager.sol`'s affected functions:

- `FundsManager::depositToken()`
- `FundsManager::executeTransactionProposal()`
- `FundsManager::sendToken()`
- `FundsManager::reclaimFunds()`

Assets:

- FundsManager.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Classification

Impact: 4/5

Likelihood: 1/5

Exploitability: Independent

Complexity: Medium

Likelihood [1-5]: 1

Impact [1-5]: 4

Exploitability [0-2]: 0

Complexity [0-2]: 1

Final Score: 1.8 (Low)

Severity: Low

Recommendations

Remediation: Use SafeERC20 library to interact with tokens safely. The `SafeERC20` library checks the `return` value call to `ERC20` tokens, but also considers

the case in which tokens do not `return` any value.

Resolution: The `SafeERC20` was incorporated into the reported `FundsManager` contract functions. Hence, the issue was fixed in the commit `f830c0c`.

Evidences

Proof of Concept

Reproduce:

The following Proof of Concept demonstrates that calling the `Test::handleTransfer()` will not lead to the revert, if the `ERC20Token::transfer()` will be successful. The opposite scenario will be that the call of the `Test::handleTransfer()` will revert if the code does not use the `SafeERC20` library and `IERC20(token).transfer()` is used instead of the `IERC20(token).safeTransfer()`.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.25;
import "@openzeppelin/contracts@5.0.2/token/ERC20/IERC20.sol";
import "@openzeppelin/contracts@5.0.2/token/ERC20/utils/SafeERC20.so
l";

contract Test {
using SafeERC20 for IERC20;
function handleTransfer(address token, address to, uint256 value) ex
ternal {
IERC20(token).safeTransfer(to, value);
}
}


contract ERC20Token {
uint public transferHappens;
function transfer(address to, uint256 value) public {
// imitation of a transfer
transferHappens += 1;
}}
```

The output:

```
[vm] from: 0x5B3...eddC4 to: Test.handleTransfer(address,address,uint256) 0x9d8...a5692 value: 0 wei data: 0x461...00064 logs: 0 hash: 0x0e2...825f5
```

The second scenario can be found below.

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.25;
import "@openzeppelin/contracts@5.0.2/token/ERC20/IERC20.sol";


contract Test {
function handleTransfer(address token, address to, uint256 value) ex
ternal {
require(IERC20(token).transfer(to, value), "");

}
}

contract ERC20Token {
uint public transferHappens;
function transfer(address to, uint256 value) public {
transferHappens += 1;
```

```
  }
  }
```

The output:

```
[vm] from: 0x5B3...eddC4 to: Test.handleTransfer(address,address,uint256) 0x5FD...9d88D value: 0 wei data: 0x461...00064 logs: 0 hash: 0x675...916b5
transact to Test.handleTransfer errored: Error occurred: revert.

revert
        The transaction has been reverted to the initial state.
Note: The called function should be payable if you send value and the value you send should be less than your current balance.
You may want to cautiously increase the gas limit if the transaction went out of gas.
```

See more

F-2024-2353 - Proposals can be executed with spam votes - Low

Description:

When a user wants to vote for a proposal, they will call the method `vote`. Such function contains the modifier `hasMinBalance`, which will ensure the user's voting power surpasses a certain threshold required to vote.

```
function vote(uint projectId, uint proposalId, bool choice) public h
asMinBalance(projectId, msg.sender) {
...
}

/**
* @notice Checks if an address holds the minimum required balance of
the voting token to create a proposal.
* @return True if the address holds the minimum required balance, fa
lse otherwise.
*/
function hasMinBalance(uint projectId, address addr) external view r
eturns (bool) {
if (addr == parametersContract.gitswarmAddress()) {
// GitSwarm is exempt, used for payroll proposals
return true;
}
uint required_amount = minimumRequiredAmount(projectId);
return delegatesContract.checkVotingPower(projectId, addr, required_
amount);
}

/**
* @notice Determines the minimum required amount of voting tokens ne
eded to create or vote on a proposal.
* @return The minimum required amount of voting tokens.
*/
function minimumRequiredAmount(uint projectId) public view returns (
uint) {//audit
return votingTokenCirculatingSupply(projectId) / parametersContract.
parameters(projectId, keccak256("MaxNrOfVoters"));
}
```

Later on, once the voting period ends, the method `lockVoteCount` will be executed. This function will lock the voting and move to the next voting stage or, in case the required voting power percentage is not meet, delete the proposal.

```
/**
* @notice Locks the vote count for a proposal and starts the contest
ing phase or deletes it if it did not pass
*/
function lockVoteCount(uint projectId, uint proposalId) external {
...
uint yesVotes;
uint noVotes;
bool willExecute;
if (p.typeOfProposal == CREATE_TOKENS) {
(yesVotes, noVotes, willExecute) = checkVoteCount(projectId, proposa
lId, parametersContract.parameters(projectId, keccak256("RequiredVot
ingPowerPercentageToCreateTokens")));
} else {
(yesVotes, noVotes, willExecute) = checkVoteCount(projectId, proposa
lId, 50);
}
if (!willExecute) {
delete proposals[projectId][proposalId];
} else {
p.willExecute = true;
p.votingAllowed = false
p.endTime = block.timestamp + parametersContract.parameters(projectI
d, keccak256("BufferBetweenEndOfVotingAndExecuteProposal"));
}
```

```
emit LockVoteCount(projectId, proposalId, proposals[projectId][propo
salId].willExecute, yesVotes, noVotes);
}
```

The required voting power is evaluated via `checkVoteCount`, `getVoteCount` and `calculateTotalVotingPower`. The latest function, `calculateTotalVotingPower` will gather the voting power of each user as the sum of their voting token balance and their delegated voting token power.

```
function getVoteCount(uint projectId, uint proposalId) public view r
eturns (uint, uint) {
ERC20interface votingTokenContract = contractsManagerContract.voting
TokenContracts(projectId);
uint yesVotes;
uint noVotes;
for (uint64 i = 0; i < proposals[projectId][proposalId].nrOfVoters;
i++) {
address a = proposals[projectId][proposalId].voters[i];
Vote storage choice = proposals[projectId][proposalId].votes[a];
uint votingPower = calculateTotalVotingPower(projectId, a, proposalI
d, votingTokenContract);
if (choice.votedYes) {
yesVotes += votingPower;
} else {
noVotes += votingPower;
}
}

return (yesVotes, noVotes);
}

function calculateTotalVotingPower(uint projectId, address addr, uin
t id, ERC20interface tokenContract) view public returns (uint) {
return tokenContract.balanceOf(addr) + getDelegatedVotingPowerExclud
ingVoters(projectId, addr, id, tokenContract);
}
```

However, the minimum balance required for voting is not checked for users who recorded their vote, opening a door to inconsistency.

One possible scenario is the exposed below: tokens can be transferred from user to user, allowing the vote as long as each user reaches the required voting power.

> A given `userA` can vote for the `YES` option at any given time. Let's assume their voting power is `1000` and the minimum voting power required is `500`. Later on, whilst the voting period is still active, `userA` can `transfer` a `700` voting power to `userB`.
>
> The later,decides to vote too, in this case for the `NO` option, since `700` voting power will meet the minimum requirement of `500` voting power. `UserB` can still send `500` voting power to another `userC`.
>
> Finally `userC` can still vote for the third time using the same tokens. With their `500` voting power, `userC` can vote for `NO` too.
>
> As a result of this case, given the fact that the minimum voting power is not enforced in the `lockVoteCount` stage of the voting process, the results are not the expected. Whilst the resulting voting power for this example will be `YES = 300` and `NO = 700`, the desired result would be `YES = 0` and `NO = 500`.

Another scenario is also possible, derived from the previous one: a user holding several addresses can change their mind and vote for the other option after transferring tokens from address to address:

> A `userA` holding `addressA` and `addressB` wallets has `2000` voting power in `addressA`. This user votes for the `YES` option. If the minimum voting power is `800`, the option will be recorded.
>
> Later on, whilst the voting period is still active, `userA` `transfers` a `1500` voting power to `addressB`. If `addressB` is used to vote the option `NO`, they will expect a contribution of `1500` voting power. However, the total contribution will be equivalent to `1000` voting power, since the votes for the other option should be subtracted from the total amount.

The function `removeSpamVoters()` removes votes from users who no longer meet the minimum voting power. Thus, it can be used for cases like the former exposed, as it is its purpose. However, calling this method is not enforced in the `lockVoteCount` stage, which allows such inconsistency.

```
/**
* @notice Removes votes from voters who no longer meet the minimum v
oting power requirement.
* This can be necessary for projects with a high MaxNrOfVoters.
* @dev Helps maintain integrity by removing spam or irrelevant votes
*/
function removeSpamVoters(uint projectId, uint proposalId, uint[] me
mory indexes) external {
uint minimum_amount = contractsManagerContract.votingTokenCirculatin
gSupply(projectId) / parametersContract.parameters(projectId, keccak
256("MaxNrOfVoters"));
emit RemovedSpamVoters(projectId, proposalId, indexes);
ProposalData storage p = proposals[projectId][proposalId];
for (uint64 index = uint64(indexes.length); index > 0; index--) {
// avoiding underflow when decrementing, that would have happened fo
r value 0
uint64 i = uint64(indexes[index - 1]);
require(i < p.nrOfVoters, "Index out of bounds");
if (!delegatesContract.checkVotingPower(projectId, p.voters[i], mini
mum_amount)) {
p.nrOfVoters--;
delete p.votes[p.voters[i]];
p.voters[i] = p.voters[p.nrOfVoters];
delete p.voters[p.nrOfVoters];
}
}
}
```

Another side effect is possible during the contesting period. This is the period during which recount is allowed in order to apply a veto or cancelling a proposal if more `NO` votes than `YES` votes are applied.

```
function contestProposal(uint projectId, uint proposalId, bool doRec
ount) external returns (bool) {
require(contractsManagerContract.hasMinBalance(projectId, msg.sender
), "Not enough voting power.");
ProposalData storage proposal = proposals[projectId][proposalId];
require(proposal.willExecute, "Can not contest this proposal, it is
not in the phase of contesting");
internal_vote(projectId, proposalId, false);

if (!doRecount) {
return false;
}
```

```
return processContest(projectId, proposalId);
}

function processContest(uint projectId, uint proposalId) internal re
turns (bool) {
(uint yesVotes, uint noVotes) = getVoteCount(projectId, proposalId);
if (noVotes >= parametersContract.neededToContest(projectId)) {
delete proposals[projectId][proposalId];
emit ContestedProposal(projectId, proposalId, yesVotes, noVotes);
return true;
}

if (noVotes > yesVotes) {
delete proposals[projectId][proposalId];
emit ContestedProposal(projectId, proposalId, yesVotes, noVotes);
return true;
}
return false;
}
```

In order to call the function `contestProposal`, the sender must meet the requirement of minimum voting power. However, this is not checked for the votes that are already recorded, resulting in a system blind to the minimum voting power of already-emitted votes unless the function `removeSpamVoters` is manually executed.

The following scenario opens up, as a result of the interaction with the `VetoMinimumPercentage`: if options for `NO` are recorded for users with residual voting power (i.e. lower than the required minimum) after a token `transfer` was performed, a proposal may become cancelled without meeting the actual requirements.

Finally, the `executeProposal` function in `ExpandableSupplyTokenBase` is affected by this issue, since it performs a call to `checkVoteCount` which, as exposed, will not consider the minimum amount of voting power.

As a result, if the function `removeSpamVoters` has never been called, a proposal can go through with unexpected voting power.

Assets:

- Proposal.sol [https://github.com/GitSwarmOrg/contracts]

Status: Accepted

Classification

Impact: 3/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Likelihood [1-5]: 2

Impact [1-5]: 3

Exploitability [0-2]: 0

Complexity [0-2]: 1

Final Score: 2.3 (Low)

Hacken Calculator Version: 0.7

Severity: Low

Recommendations

Remediation: It is recommended to call `removeSpamVoters()` within `lockVoteCount()` to make sure only those users that meet the required voting power are accounted for the proposal vote. Alternatively, a check can be added into the function `calculateTotalVotingPower` in order to return `0` when the minimum voting power is not met.

Observation Details

F-2024-2227 - Missing disableInitializers() can lead to unwanted initialization - Info

Description: The project contains upgradable contracts: `ContractsManager.sol`, `Delegates.sol`, `FundsManager.sol`, `GasStation.sol`, `Proposal.sol` and `UpgradeableToken.sol` After an implementation contract is deployed on the blockchain, its implementation must be called by the development team to set up the basic functionalities of the contract. However, the absence of a _disableInitilizers call in the `constructor` of the implementation contract opens the possibility for each implementation to be directly initialized by an external actor.

Assets:

- ContractsManager.sol [https://github.com/GitSwarmOrg/contracts]
- Delegates.sol [https://github.com/GitSwarmOrg/contracts]
- FundsManager.sol [https://github.com/GitSwarmOrg/contracts]
- GasStation.sol [https://github.com/GitSwarmOrg/contracts]
- Proposal.sol [https://github.com/GitSwarmOrg/contracts]
- UpgradableToken.sol [https://github.com/GitSwarmOrg/contracts]
- Parameters.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Recommendations

Remediation: It is recommended to introduce a call to `_disableInitializers()` within implementations constructor.

Resolution: The recommended `_disableInitializers()` call was added to the aforementioned assets. The issue was fixed in the commit `f830c0c`.

F-2024-2258 - Explicit uint256 best practice violation - Info

Description: The contracts declare unsigned integers as `uint`, which, by default, is set to `256` bits. However, it is a best practice to explicitly set them to `uint256` instead of using the default `uint`.

Assets:

- Common.sol [https://github.com/GitSwarmOrg/contracts]
- Constants.sol [https://github.com/GitSwarmOrg/contracts]
- ERC20Base.sol [https://github.com/GitSwarmOrg/contracts]
- ERC20interface.sol [https://github.com/GitSwarmOrg/contracts]
- ExpandableSupplyTokenBase.sol [https://github.com/GitSwarmOrg/contracts]
- Interfaces.sol [https://github.com/GitSwarmOrg/contracts]
- MyTransparentUpgradeableProxy.sol [https://github.com/GitSwarmOrg/contracts]
- SelfAdminTransparentUpgradeableProxy.sol [https://github.com/GitSwarmOrg/contracts]
- ContractsManager.sol [https://github.com/GitSwarmOrg/contracts]
- Delegates.sol [https://github.com/GitSwarmOrg/contracts]
- ExpandableSupplyToken.sol [https://github.com/GitSwarmOrg/contracts]
- FixedSupplyToken.sol [https://github.com/GitSwarmOrg/contracts]
- FundsManager.sol [https://github.com/GitSwarmOrg/contracts]
- GasStation.sol [https://github.com/GitSwarmOrg/contracts]
- GitSwarmToken.sol [https://github.com/GitSwarmOrg/contracts]
- Proposal.sol [https://github.com/GitSwarmOrg/contracts]
- UpgradableToken.sol [https://github.com/GitSwarmOrg/contracts]
- Parameters.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Recommendations

Remediation: Consider updating all `uint` declarations to `uint256`.

Resolution: The `uint` values were replaced with `uint256` in the aforementioned assets. The issue was fixed in the commit `ec88426`.

F-2024-2260 - Solidity style guide violation - Info

Description: The contract ERC20Base declares a `constant` state variable as `_decimals`. However, according to the Solidity style guide, constants should be named with `CAPITAL_LETTERS`.

Assets:

- ERC20Base.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Recommendations

Remediation: It is recommended to update the name of the constant variable `_decimals` to comply with the Solidity style guide.

Resolution: The variable `_decimals` was replaced with `DECIMALS` in the commit `f830c0c`.

F-2024-2263 - Missing minimum amount check results in waste of Gas - Info

Description:

The function `proposeCreateTokens` does not check that the input `amount > 0`, allowing the creation of a proposal that would result in a waste of Gas.

Similarly, the function `proposeTransferToGasAddress` allows the processing of a proposal of value `0`.

The aforementioned functions can be seen in the code snippets below.

`proposeCreateTokens()`

```
/**
* @dev Proposes the creation of new tokens.
* @param amount The amount of new tokens to create.
*/
function proposeCreateTokens(uint amount) external {
require(!createMoreTokensDisabled, "Increasing token supply is perma
nently disabled");
createTokensProposals[proposalContract.nextProposalId(projectId)].am
ount = amount;
proposalContract.createProposal(projectId, CREATE_TOKENS, msg.sender
);
}
```

and `proposeTransferToGasAddress()`

```
/**
* @dev Proposes a transfer of ETH to a specified gas address.
* @param amount The amount of ETH to transfer.
* @param to The recipient address of the ETH.
* @notice The proposal is recorded and will be executed upon approva
l.
*/
function proposeTransferToGasAddress(uint amount, address to) extern
al {
transferToGasAddressProposals[proposalContract.nextProposalId(0)].am
ount = amount;
transferToGasAddressProposals[proposalContract.nextProposalId(0)].to
= to;
proposalContract.createProposal(0, TRANSFER_TO_GAS_ADDRESS, msg.send
er);
}
```

Assets:

- ExpandableSupplyTokenBase.sol [https://github.com/GitSwarmOrg/contracts]
- GasStation.sol [https://github.com/GitSwarmOrg/contracts]

Status: Accepted

Recommendations

Remediation: It is recommended to check that the input `amount > 0` or that surpasses a minimal amount.

F-2024-2268 - Residual proposal data is not deleted in proposal execution - Info

Description: In the function `executeProposal()`, the created proposal in the `proposalContract` is not deleted via `deleteProposal()` when `typeOfProposal == DISABLE_CREATE_MORE_TOKENS`, resulting in residual data that can confuse.

```
function executeProposal(uint proposalId) external {
...
if (typeOfProposal == CREATE_TOKENS) {
..
delete createTokensProposals[proposalId];
proposalContract.deleteProposal(projectId, proposalId);
createTokens(amount);
} else if (typeOfProposal == DISABLE_CREATE_MORE_TOKENS) {
createMoreTokensDisabled = true;
emit CreateMoreTokensDisabledEvent();
} else {
revert('Unexpected proposal type');
}
emit ExecuteProposal(projectId, proposalId);
}
```

Assets:

- ExpandableSupplyTokenBase.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Recommendations

Remediation: Consider deleting the created proposal via `deleteProposal()`.

Resolution: The `DISABLE_CREATE_MORE_TOKENS` proposal is now deleted in the `ExpandableSupplyTokenBase::executeProposal()`. It was fixed in the commit `ec88426`.

F-2024-2280 - Redundant call - Info

Description: The function `executeProposal` will delete the processed proposal via `deleteProposal` in `CHANGE_TRUSTED_ADDRESS` type. However, this call is duplicated since it be performed later in the same function:

```
function executeProposal(uint projectId, uint proposalId) external {
...
if (typeOfProposal == CHANGE_TRUSTED_ADDRESS) {
...
proposalContract.deleteProposal(projectId, proposalId);
} else if (typeOfProposal == CHANGE_PARAMETER) {
...
} else {
...
}
proposalContract.deleteProposal(projectId, proposalId);
emit ExecuteProposal(projectId, proposalId);
}
```

Assets:

- Parameters.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Recommendations

Remediation: It is recommended to delete redundant code.

Resolution: The redundant call was deleted from the `Parameters::executeProposal()` in the commit `ec88426`.

F-2024-2292 - Cache array length in loops to save gas - Info

Description: Several loops calculate the length of `storage` arrays instead of caching such length into a `memory` variable. As a consequence, the whole array will be accessed as a `storage` reading for each iteration, spending a big amount of gas unnecessarily.

Additionally, this extra amount of gas will contribute to the likelihood of reaching the block gas limit for such operations, as reported in the `Risk` section of this report.

An example can be found below:

```
function isTrustedAddress(uint projectId, address trustedAddress) vi
ew public returns (bool) {
...
for (uint i = 0; i < trustedAddresses[projectId].length; i++) {
if (trustedAddresses[projectId][i] == trustedAddress) {
return true;
}
}
return false;
}
```

Affected functions are:

- `Parameters:: isTrustedAddress()`
- `Delegates:: undelegateAddress(), undelegateAllFromAddress()`
- `ContractsManager:: burnedTokens(), executeProposal()`
- `Proposal:: getVoteCount(), removeSpamVoters()`

Assets:

- ContractsManager.sol [https://github.com/GitSwarmOrg/contracts]
- Delegates.sol [https://github.com/GitSwarmOrg/contracts]
- Proposal.sol [https://github.com/GitSwarmOrg/contracts]
- Parameters.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Recommendations

Remediation: Cache the array length into a `memory` variable to be iterated through.

Resolution: Fixed in commit `d17439d`.

F-2024-2302 - Code does not comply with NatSpec requirements - Info

Description:

According to the constructors' NatSpec, the parameter `creatorSupply` must be non-zero. However, this is not enforced. The aforementioned contracts can be found below.

`ExpandableSupplyToken.sol`

```
contract ExpandableSupplyToken is ExpandableSupplyTokenBase {
/**
* @param prjId The project ID for the new token.
* @param supply The total fixed supply of the token.
* @param creatorSupply The portion of the supply allocated to the cr
eator.
* @param contractsManagerAddress Address of the Contracts Manager.
* @param fundsManagerContractAddress Address of the Funds Manager Co
ntract.
* @param proposalContractAddress Address of the Proposal Contract.
* @param tokenName The name of the token.
* @param tokenSymbol The symbol of the token.
*
* Requirements:
* - `creatorSupply` must be non-zero.
* - Contract must have no pre-existing supply (`__totalSupply` == 0)
.
*/
constructor(
string memory prjId,
uint supply,
uint creatorSupply,
address contractsManagerAddress,
address fundsManagerContractAddress,
address proposalContractAddress,
address parametersContractAddress,
string memory tokenName,
string memory tokenSymbol
) {
name = tokenName;
symbol = tokenSymbol;
_init(address(0), fundsManagerContractAddress, parametersContractAdd
ress, proposalContractAddress, address(0), contractsManagerAddress);
contractsManagerContract.createProject(prjId, address(this), false);
createInitialTokens(supply, creatorSupply);
}
}
```

and `FixedSupplyToken.sol`

```
contract FixedSupplyToken is ERC20Base {
uint immutable public projectId;
/**
* @param prjId The project ID for the new token.
* @param supply The total fixed supply of the token.
* @param creatorSupply The portion of the supply allocated to the cr
eator.
* @param contractsManagerAddress Address of the Contracts Manager.
* @param fundsManagerContractAddress Address of the Funds Manager Co
ntract.
* @param proposalContractAddress Address of the Proposal Contract.
* @param tokenName The name of the token.
* @param tokenSymbol The symbol of the token.
*
* Requirements:
* - `creatorSupply` must be non-zero.
* - Contract must have no pre-existing supply (`__totalSupply` == 0)
.
*/
constructor(
```

```
string memory prjId,
uint supply,
uint creatorSupply,
address contractsManagerAddress,
address fundsManagerContractAddress,
address proposalContractAddress,
address parametersContractAddress,
string memory tokenName,
string memory tokenSymbol
) {
name = tokenName;
symbol = tokenSymbol;
_init(address(0), fundsManagerContractAddress, parametersContractAdd
ress, proposalContractAddress, address(0), contractsManagerAddress);
contractsManagerContract.createProject(prjId, address(this), false);
projectId = contractsManagerContract.nextProjectId() - 1;
__totalSupply = supply + creatorSupply;
__balanceOf[msg.sender] = creatorSupply;
__balanceOf[address(fundsManagerContract)] = supply;
fundsManagerContract.updateBalance(projectId, address(this), supply)
;
}
}
```

Assets:

- ExpandableSupplyToken.sol [https://github.com/GitSwarmOrg/contracts]
- FixedSupplyToken.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Recommendations

Remediation: Consider adding a check to make sure `creatorSupply > 0`.

Resolution: The NatSpec comments were removed. The mismatch between NatSpec and the implementation was fixed in the commit `ec88426`.

F-2024-2303 - Missing check to ensure the next project Id is correct - Info

Description: New projects are created in `ContractsManager.sol` using `createProject()`, where the `projectId` is defined as `dbProjectId`. However, there is no check to enforce that the parameter corresponds to `nextProjectId`. There is a code snippet below which demonstrates this problem.

```
function createProject(string memory dbProjectId, address tokenContr
actAddress, bool checkErc20) public {
require(tokenContractAddress != address(0), "Contract address can't
be 0x0");
if (checkErc20) {
require(isERC20Token(tokenContractAddress), "Address is not an ERC20
token contract");
}
parametersContract.initializeParameters(nextProjectId);
burnAddresses[nextProjectId].push(BURN_ADDRESS);
votingTokenContracts[nextProjectId] = ERC20interface(tokenContractAd
dress);
emit CreateProject(nextProjectId, dbProjectId, tokenContractAddress)
;
nextProjectId++;
}
```

Assets:

- ContractsManager.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Recommendations

Remediation: Consider adding a check to make sure `dbProjectId` matches `nextProjectId`.

Resolution: The issue was considered fixed after the following explanation from the GitSwarm team:

> The emitted `dbProjectId` is a string and only used in our server backend code, unrelated to the uint `projectId` and `nextProjectId` variables from the contracts.

F-2024-2304 - Missing check if project exists - Info

Description: In the `ContractsManager::proposeTransaction()` there is no check to ensure the corresponding project exists, resulting in a waste of gas and unexpected results.

```
function proposeTransaction(
uint projectId,
address[] memory tokenContractAddress,
uint[] memory amount,
address[] memory to,
uint[] memory depositToProjectId) external {

require(amount.length > 0, "Amount can't be an empty list.");
require(amount.length == to.length && amount.length == depositToProj
ectId.length &&
amount.length == tokenContractAddress.length,
"'amount', 'to' and 'depositToProjectId' arrays must have equal leng
th");
for (uint i = 0; i < amount.length; i++) {
require(amount[i] > 0, "Amount must be greater than 0.");
}

uint nextProposalId = proposalContract.nextProposalId(projectId);
TransactionProposal storage p = transactionProposals[projectId][next
ProposalId];
p.token = tokenContractAddress;
p.amount = amount;
p.depositToProjectId = depositToProjectId;
p.to = to;
proposalContract.createProposal(projectId, TRANSACTION, msg.sender);
}
```

Assets:

- ContractsManager.sol [https://github.com/GitSwarmOrg/contracts]

Status: Accepted

Recommendations

Remediation: Consider adding a check to make sure the project exists.

F-2024-2352 - GitSwarm address can be removed from voters - Info

Description: The functions `removeSpamVoters()` and `getSpamVoters()` identify the voters that have not enough voting power and should/will be removed from the voting.

However, it is not checked whether the `GitSwarm` address is deleted. This allows this privileged, tie-breaker address to be erased, whilst playing an important role in the voting system.

```
function removeSpamVoters(uint projectId, uint proposalId, uint[] me
mory indexes) external {
uint minimum_amount = contractsManagerContract.votingTokenCirculatin
gSupply(projectId) / parametersContract.parameters(projectId, keccak
256("MaxNrOfVoters"));
emit RemovedSpamVoters(projectId, proposalId, indexes);
ProposalData storage p = proposals[projectId][proposalId];
for (uint64 index = uint64(indexes.length); index > 0; index--) {
// avoiding underflow when decrementing, that would have happened fo
r value 0
uint64 i = uint64(indexes[index - 1]);
require(i < p.nrOfVoters, "Index out of bounds");
if (!delegatesContract.checkVotingPower(projectId, p.voters[i], mini
mum_amount)) {
p.nrOfVoters--;
delete p.votes[p.voters[i]];
p.voters[i] = p.voters[p.nrOfVoters];
delete p.voters[p.nrOfVoters];
}
}
}
```

Assets:

- Proposal.sol [https://github.com/GitSwarmOrg/contracts]

Status: Fixed

Recommendations

Remediation: Consider protecting the `GitSwarm` address from being removed from voters.

Resolution: Fixed in commit **d17439d**. The `gitswarmAddress` address is now skipped in the `Proposal::removeSpamVoters()` and `Proposal::getSpamVoters()`.

F-2024-2353 - The isERC20Token() function does not completely verify the adherence to the ERC20 standard - Info

Description:

The `ContractsManager::isERC20Token()` function does not completely verify the adherence to the ERC20 standard, which creates a security risk of adding not fully erc20 compliant tokens. The function can be found in the code snippet below.

```
/**
* @notice Verifies if a given address is an ERC20 token contract.
* @dev Attempts to call ERC20-specific functions to confirm complian
ce.
* @param _addr The address to be verified.
* @return True if the address is an ERC20 token contract, false othe
rwise.
*/
function isERC20Token(address _addr) public view returns (bool) {
address dummyAddress = 0x0000000000000000000000000000000000000000;


if (_addr.code.length == 0) {
return false;
}


try ERC20interface(_addr).name() {} catch {return false;}
try ERC20interface(_addr).symbol() {} catch {return false;}
try ERC20interface(_addr).decimals() {} catch {return false;}
try ERC20interface(_addr).totalSupply() {} catch {return false;}
try ERC20interface(_addr).balanceOf(dummyAddress) {} catch {return f
alse;}
try ERC20interface(_addr).allowance(dummyAddress, dummyAddress) {} c
atch {return false;}


return true;
}
```

The function `ContractsManager::isERC20Token()` does not take into account the complete list of ERC20 methods: `transfer()`, `approve()` and `transferFrom()`, which creates a risk of adding a not fully ERC20 compliant voting token.

Assets:

- ContractsManager.sol [https://github.com/GitSwarmOrg/contracts]

Status: Mitigated

Recommendations

Remediation: It is recommended to add the necessary checks to ensure that the adherence to the ERC20 standard is verified completely.

Resolution: The finding was considered Mitigated given the following explanation from the GitSwarm team:

The purpose of this method is to catch user error. Creating a project with a non functioning voting token will lead to that project being unusable but it should not affect other projects.

F-2024-2355 - Users can create projects without ERC20 compliant voting tokens - Info

Description:

Users can create new projects using the function `createProject`, where they input the `tokenContractAddress` as the token to be used for voting.

This token should be ERC20 compliant, but the users can choose not to check that compliance when calling this function. As a result, users can use addresses that are not compliant with this requirement, resulting in a malfunction.

```
function createProject(string memory dbProjectId, address tokenContr
actAddress, bool checkErc20) public {
require(tokenContractAddress != address(0), "Contract address can't
be 0x0");
if (checkErc20) {
require(isERC20Token(tokenContractAddress), "Address is not an ERC20
token contract");
}
parametersContract.initializeParameters(nextProjectId);
burnAddresses[nextProjectId].push(BURN_ADDRESS);
votingTokenContracts[nextProjectId] = ERC20interface(tokenContractAd
dress);
emit CreateProject(nextProjectId, dbProjectId, tokenContractAddress)
;
nextProjectId++;
}
```

Assets:

- ContractsManager.sol [https://github.com/GitSwarmOrg/contracts]

Status: Accepted

Recommendations

Remediation: It is recommended to always check wether the input token is ERC20 compliant.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

[hknio/severity-formula]

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details

Repository	https://github.com/GitSwarmOrg/contracts
Commit	a07999c
Remediation commit	8e707a9
Whitepaper	-
Requirements	NatSpec
Technical Requirements	NatSpec

Contracts in Scope

./contracts/prod/1.1/base/Common.sol

./contracts/prod/1.1/base/Constants.sol

./contracts/prod/1.1/base/ERC20Base.sol

./contracts/prod/1.1/base/ERC20interface.sol

./contracts/prod/1.1/base/ExpandableSupplyTokenBase.sol

./contracts/prod/1.1/base/Interfaces.sol

./contracts/prod/1.1/base/MyTransparentUpgradeableProxy.sol

./contracts/prod/1.1/base/SelfAdminTransparentUpgradeableProxy.sol

./contracts/prod/1.1/ContractsManager.sol

./contracts/prod/1.1/Delegates.sol

./contracts/prod/1.1/ExpandableSupplyToken.sol

./contracts/prod/1.1/FixedSupplyToken.sol

./contracts/prod/1.1/FundsManager.sol

./contracts/prod/1.1/GasStation.sol

./contracts/prod/1.1/Parameters.sol

./contracts/prod/1.1/Proposal.sol

Contracts in Scope

./contracts/prod/1.1/UpgradableToken.sol