Zellic

GoGoPool

Smart Contract Security Assessment

February 22, 2023

Prepared for:

Multisig Labs

Prepared by:

Katerina Belotskaia and Vlad Toie

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Multisig Labs from November 14th to 29th, 2022.

Our general overview of the code is that it was well-organized and structured. The code coverage is high, and tests are included for the majority of the functions. Some areas of the code have limited negative testing, which could be improved. The documentation was adequate, although it could be improved. The code was easy to comprehend, and in most cases, intuitive.

Zellic thoroughly reviewed the GoGoPool codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

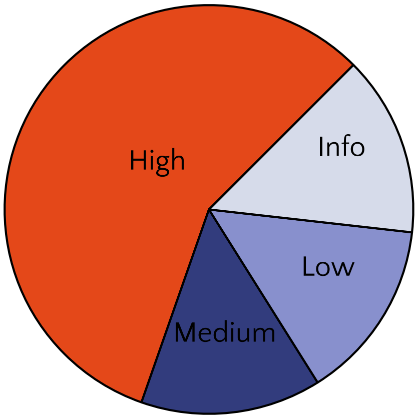
Specifically, taking into account GoGoPool's threat model, we focused heavily on issues that would break core invariants such as the management of the minipools, staking, withdrawing and minting shares, and the states of the Storage contract.

During our assessment on the scoped GoGoPool contracts, we discovered seven findings. Of the seven findings, four were of high severity, one was of medium severity, one was of low severity and the remaining finding was informational.

Additionally, Zellic recorded its notes and observations from the audit for Multisig Labs's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	4
Medium	1
Low	1
Informational	1



2 Introduction

2.1 About GoGoPool

GoGoPool allows Avalanche users to stake a minimum of 0.01 AVAX and operate a validator node with a minimum of 1000 AVAX, while providing instant liquidity and earning rewards for validating subnets. As an open protocol, any individual, business, or subnet can plug into the protocol without being charged platform fees.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of open-source tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so forth.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their importance may differ. This varies based on numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

GoGoPool Contracts

Repository	https://github.com/multisig-labs/gogopool-contracts
Versions	`7768287e94bff0f2e12f03427309777e82a6e2fc`
Contracts	• BaseAbstract.sol • RewardsPool.sol • BaseUpgradeable.sol • MultisigManager.sol • Oracle.sol • MinipoolManager.sol • Vault.sol • Storage.sol • Base.sol • ProtocolDAO.sol • Ocyticus.sol • tokens/TokenggAVAX.sol • tokens/TokenGGP.sol • tokens/upgradeable/ERC20Upgradeable.sol • tokens/upgradeable/ERC4626Upgradeable.sol • ClaimProtocolDAO.sol • ClaimNodeOp.sol • Staking.sol
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of four person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia, Engineer
kate@zellic.io

Vlad Toie, Engineer
vlad@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

November 14, 2022	Kick-off call
November 14, 2022	Start of primary review period
November 28, 2022	End of primary review period

3 Detailed Findings

3.1 The transferAVAX function allows arbitrary transfers

- **Target**: Vault.sol
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

The `transferAVAX` function is used to perform transfers of `avax` between two registered contracts.

```
function transferAVAX(
    string memory fromContractName,
    string memory toContractName,
    uint256 amount
) external onlyRegisteredNetworkContract {

    // Valid Amount?
    if (amount == 0) {
        revert InvalidAmount();
    }
    // Emit transfer event
    emit AVAXTransfer(fromContractName, toContractName, amount);

    // Make sure the contracts are valid, will revert if not
    getContractAddress(fromContractName);
    getContractAddress(toContractName);
    // Verify there are enough funds
    if (avaxBalances[fromContractName] < amount) {
        revert InsufficientContractBalance();
    }
    // Update balances
    avaxBalances[fromContractName] = avaxBalances[fromContractName]
    - amount;
    avaxBalances[toContractName] = avaxBalances[toContractName] + amount;
}
```

The current checks ensure that the `msg.sender` is a `registeredNetworkContract`; however, the function lacks a check on whether the `msg.sender` actually calls the function or not.

Impact

Due to the fact that `fromContractName` can be an arbitrary address, a presumably malicious `registeredNetwork` contract can drain the `avax` balances of all the other registered contracts.

Recommendations

We recommend removing the `fromContractName` parameter altogether and ensuring that the funds can only be transferred by the caller of the function, `msg.sender`.

```
function transferAVAX( // @audit-info doesn't exist in rocketvault
    string memory fromContractName,
    string memory toContractName,
    uint256 amount
) external onlyRegisteredNetworkContract {

    // Valid Amount?
    if (amount == 0) {
        revert InvalidAmount();
    }
    // Emit transfer event
    emit AVAXTransfer(msg.sender, toContractName, amount);

    // Make sure the contracts are valid, will revert if not
    getContractAddress(msg.sender);
    getContractAddress(toContractName);
    // Verify there are enough funds
    if (avaxBalances[msg.sender] < amount) {
        revert InsufficientContractBalance();
    }
    // Update balances
    avaxBalances[msg.sender] = avaxBalances[msg.sender] - amount;
    avaxBalances[toContractName] = avaxBalances[toContractName] + amount;
}
```

Remediation

The issue has been fixed by Multisig Labs in commit 84211f.

3.2 Ocyticus does not include the `Staking` pause

- **Target**: Ocyticus, Staking
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Description

The `pauseEverything` and `resumeEverything` functions are used to restrict access to important functions.

```
function pauseEverything() external onlyDefender {
    ProtocolDAO dao = ProtocolDAO(getContractAddress(“ProtocolDAO”));
    dao.pauseContract(“TokenggAVAX”);
    dao.pauseContract(“MinipoolManager”);
    disableAllMultisigs();
}

/// @notice Reestablish all contract's abilities
/// @dev Multisigs will need to be enabled seperately, we dont know which
    ones to enable
function resumeEverything() external onlyDefender {
    ProtocolDAO dao = ProtocolDAO(getContractAddress(“ProtocolDAO”));
    dao.resumeContract(“TokenggAVAX”);
    dao.resumeContract(“MinipoolManager”);
}
```

Apart from the `TokenGGAvax` and `MinipoolManager`, the `Staking` contract also makes use of the `whenNotPaused` modifier for its important functions. The `paused` state, will, however, not trigger at the same time with the `pauseEverything` call, since the `Staking` contract is omitted here, both for pausing and resuming.

Impact

Should an emergency arise, `pauseEverything` will be called. In this case, `Staking` will be omitted, which could put user funds in danger.

Recommendations

We recommend ensuring that the `Staking` contract is also paused in the `pauseEverything` function as well as un-paused in the `resumeEverything` function.

```
function pauseEverything() external onlyDefender {
    ProtocolDAO dao = ProtocolDAO(getContractAddress("ProtocolDAO"));
    dao.pauseContract("TokenggAVAX");
    dao.pauseContract("MinipoolManager");
    dao.pauseContract("Staking");
    disableAllMultisigs();
}

/// @notice Reestablish all contract's abilities
/// @dev Multisigs will need to be enabled seperately, we dont know which
    ones to enable
function resumeEverything() external onlyDefender {
    ProtocolDAO dao = ProtocolDAO(getContractAddress("ProtocolDAO"));
    dao.resumeContract("TokenggAVAX");
    dao.resumeContract("MinipoolManager");
    dao.resumeContract("Staking");
}
```

Remediation

The issue has been fixed by Multisig Labs in commit dbc499.

3.3 The reward amount manipulation

- **Target**: ClaimNodeOp.sol
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: High
- **Impact**: High

Descriptions

A staker is eligible for the upcoming rewards cycle if they have staked their tokens for a long enough period of time. The reward amount is distributed in proportion to the amount of funds staked by the user from the total amount of funds staked by all users who claim the reward. But since the `rewardsStartTime` is the time of creation of only the first pool, and during the reward calculations all staked funds are taken into account, even if they have not yet been blocked and can be withdrawn, the attack described below is possible.

The attack scenario:

1. An attacker stakes ggp tokens and creates a minipool with a minimum `avaxAssignmentRequest` value.
2. The multisig initiates the staking process by calling the `claimAndInitiateStaking` function.
3. Wait for the time of distribution of rewards.
4. Before the reward distribution process begins, the attacker creates a new minipool with the maximum `avaxAssignmentRequest` value.
5. Initiate the reward distribution process.
6. Immediately after that, the attacker cancels the minipool with `cancelMinipool` function before the `claimAndInitiateStaking` function call and returns most part of their staked funds.

Impact

The attacker can increase their reward portion without actually staking their own funds.

Recommendations

Take into account only the funds actually staked, or check that all minipools have been launched.

Remediation

The issue has been fixed by Multisig Labs in commits c90b2f and f49931.

3.4 Network registered contracts have absolute storage control

- **Target**: Project-wide
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: High
- **Impact**: High

Description

The network-registered contracts have absolute control over the storage that all the contracts are associated with through the `Storage` contract. This is inherent due to the overall design of the protocol, which makes use of a single `Storage` contract eliminating the need of local storage. For that reason any `registeredContract` can `update` **any** storage slot even if it "belongs" to another contract.

```
modifier onlyRegisteredNetworkContract() {
    if (booleanStorage[keccak256(abi.encodePacked(“contract.exists”,
    msg.sender))] == false && msg.sender != guardian) {
        revert InvalidOrOutdatedContract();
    }
    _;
}

// ...
function setAddress(bytes32 key, address value)
    external onlyRegisteredNetworkContract {
    addressStorage[key] = value;
}

function setBool(bytes32 key, bool value)
    external onlyRegisteredNetworkContract {
    booleanStorage[key] = value;
}

function setBytes(bytes32 key, bytes calldata value)
    external onlyRegisteredNetworkContract {
    bytesStorage[key] = value;
}
```

As an example, the setter functions inside the `Staking` contract have different restrictions for caller (e.g., the `setLastRewardsCycleCompleted` function can be called only by `ClaimNodeOp` contract), but actually the `setUint` function from it may be called by any

RegisteredNetworkContract.

Impact

We believe that in a highly unlikely case, a malicious networkRegistered contract could potentially alter the entire protocol Storage to their will. Additionally, if it were possible to setBool of an arbitrary address, then this scenario would be further exploitable by a malicious developer contract.

Recommendations

We recommend paying extra attention to the registration of networkContracts, as well as closely monitoring where and when the setBool function is used, since the network registration is based on a boolean value attributed to the contract address.

Remediation

The issue has ben acknowledged by the Multisig Labs. Their official reply is reproduced below:

> While it is true that any registered contract can write to Storage, we view all of the separate contracts comprising the Protocol as a single system. A single entity (either the Guardian Multisig or in future the ProtocolDAO) will be in control of all of the contracts. In this model, if an attacker can register a single malicious contract, then they are also in full control of the Protocol itself. Because all of the contracts are treated as a single entity, there is no additional security benefit to be gained by providing access controls between the various contract's storage slots. As a mitigation, the Protocol will operate several distributed Watchers that will continually scan the central Storage contract, and alert on any changes.

3.5 Oracle may reflect an outdated price

- **Target**: Oracle
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

Some functions at protocol-level make use of the `getGGPPriceInAvax`. This getter retrieves the `price`, which is set by the `Rialto` multisig.

```
/// @notice Get the price of GGP denominated in AVAX
/// @return price of ggp in AVAX
/// @return timestamp representing when it was updated
function getGGPPriceInAVAX() external view returns (uint256 price,
    uint256 timestamp) {
    price = getUint(keccak256(“Oracle.GGPPriceInAVAX”));
    if (price == 0) {
        revert InvalidGGPPrice();
    }
    timestamp = getUint(keccak256(“Oracle.GGPTimestamp”));
}
```

Due to the nature of on-chain price feeds, Oracles need to have an as-often-as-possible policy in regards to how often the price gets updated. For that reason, the reliance on the `Rialto` may be problematic should it fail to update the `price` often enough.

Impact

Should the price be erroneous, possible front-runs may happen at the protocol level, potentially leading to a loss of funds on the user-end side.

Recommendations

We recommend implementing a slippage check, which essentially does not allow a price to be used should it have been updated more than `x` blocks ago.

Remediation

The finding has been acknowledged by the Multisig Labs team. Their official reply is reproduced below:

> The price of GGP is used in the Protocol to determine collateralization ratios for minipools as well as slashing amounts. If the price of GGP is unknown or outdated, the protocol cannot operate. So our remediation for this will be to have a distributed set of Watchers that will Pause the Protocol if the GGP Price becomes outdated. At some point in the future the Protocol will use on-chain TWAP price oracles to set the GGP price.

3.6 Fields are not reset exactly after their usage

- **Target**: MinipoolManager
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

Due to the nature of the protocol, some fields are queried and used in one intermediary state of the application and then reset in the last state of the application. As an example, see the `avaxNodeOpRewardAmt` value, which is queried and used in `withdrawMinipoolFunds` (which can only be called in the `WITHDRAWABLE` stage)

```
function withdrawMinipoolFunds(address nodeID) external nonReentrant {
    int256 minipoolIndex = requireValidMinipool(nodeID);
    address owner = onlyOwner(minipoolIndex);
    requireValidStateTransition(minipoolIndex, MinipoolStatus.Finished);
    setUint(keccak256(abi.encodePacked(“minipool.item”, minipoolIndex,
    “.status”)), uint256(MinipoolStatus.Finished));

    uint256 avaxNodeOpAmt
    = getUint(keccak256(abi.encodePacked(“minipool.item”, minipoolIndex,
    “.avaxNodeOpAmt”)));

    uint256 avaxNodeOpRewardAmt
    = getUint(keccak256(abi.encodePacked(“minipool.item”, minipoolIndex,
    “.avaxNodeOpRewardAmt”)));

    uint256 totalAvaxAmt = avaxNodeOpAmt + avaxNodeOpRewardAmt;

    Staking staking = Staking(getContractAddress(“Staking”));
    staking.decreaseAVAXStake(owner, avaxNodeOpAmt);

    Vault vault = Vault(getContractAddress(“Vault”));
    vault.withdrawAVAX(totalAvaxAmt);
    owner.safeTransferETH(totalAvaxAmt);
}
```

and then either reset in the `recordStakingEnd` function, to the new rounds' `avaxNodeOpRewardAmt`, or set to `0` in `recordStakingError`.

The protocol's structure assumes that the way in which the states are transitioned through is consistent.

Impact

Should major changes occur in the future of the protocol, we suspect that some states that are presumably reset in an eventual state of the protocol may be omitted. This could in turn lead to unexpected consequences to the management of the minipool.

Recommendations

We highly recommend that once important storage states are used, they should also be reset. In this way, future versions of the protocol will have a solid way of transitioning without requiring additional synchronization of storage state.

Remediation

The issue has ben acknowledged by the Multisig Labs. Their official reply is reproduced below:

> The Protocol maintains some fields in Storage so that data such as avaxNodeOpRewardAmt can be displayed to the end user. The fields will be reset if the user relaunches a minipool with the same nodeID again in the future. This is by design.

3.7 Contracts can deposit arbitrary tokens in the Vault

- **Target**: Vault.sol
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Low
- **Impact**: Informational

Description

Multiple functions from the `Vault` contract allow arbitrary tokens to be deposited and withdrawn by `networkRegistered` contracts. For example, see the `depositToken` function:

```
function depositToken(string memory networkContractName,
    ERC20 tokenContract, uint256 amount
) external guardianOrRegisteredContracts {
    // Valid Amount?
    if (amount == 0) {
        revert InvalidAmount();
    }
    // Make sure the network contract is valid (will revert if not)
    getContractAddress(networkContractName);

    // Get contract key
    bytes32 contractKey = keccak256(abi.encodePacked(networkContractName,
    address(tokenContract)));
    // Emit token transfer event
    emit TokenDeposited(contractKey, address(tokenContract), amount);
    // Send tokens to this address now, safeTransfer will revert if it
    fails
    tokenContract.safeTransferFrom(msg.sender, address(this), amount);
    // Update balances
    tokenBalances[contractKey] = tokenBalances[contractKey] + amount;
}
```

Impact

As per the current implementation, there are no security implications. However, we consider that the `Vault` plays an essential role in the entire protocol, and thus we highly recommend fixing this issue for posterity.

Recommendations

Upon discussions with the Multisig Lab team, we settled that the best mitigation is `whitelisting` the `tokenContract` that are used in each function. This further allows flexibility and security in smoothly upgrading the `Vault` should it support more tokens. In that case, the mitigated version of the function could be:

```
function depositToken(string memory networkContractName,
    ERC20 tokenContract, uint256 amount
) external guardianOrRegisteredContracts {

    require(whitelisted[tokenContract], “tokenContract not whitelisted”);

    if (amount == 0) {
        revert InvalidAmount();
    }

    // ...
```

Remediation

The issue has been fixed by Multisig Labs in commit 644e8e.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 The `rewardsCycleEnd` calculation

The `rewardsCycleEnd` value from the `TokenggAVAX` contract should always be evenly divisible by `rewardsCycleLength`. This condition, however, is only met during the contract initialization, where the `rewardsCycleLength` is initially calculated. The `rewardsCycleLength` is eventually recalculated inside the `syncRewards` function, but this time, there is no check whether the value is evenly divisible or not.

Remediation

The issue has been fixed by Multisig Labs in commit 556ac4.

4.2 Lack of checks

1. The `calculateAndDistributeRewards` function from the `ClaimNodeOp` contract does not explicitly verify that the `stakerAddr` is a valid staker address.
2. Add a check that `rewardsPool.getRewardsCycleCount()` is not zero to the `calculateAndDistributeRewards` function from the `ClaimNodeOp` contract.
3. The `registerMultisig` function in the `MultisigManager` contract does not check that the `multisig.count` value has reached 10 to ensure that `There will never be more than 10 total multisigs`, which is a comment on the `requireNextActiveMultisig` function.
4. The `recordStakingStart` function in the `MinipoolManager` contract does not validate that the `startTime` value is not greater than the current time.
5. The `setRewardsStartTime` function in the `Staking` contract does not validate that the `time` value is not greater than the current time or that it can be only the current time or 0.
6. The `getInflationAmt` in the `RewardsPool` contract does not process the case when the max amount of tokens are released (22_500_000, the total minted amount).

Remediation

The issue has been fixed by Multisig Labs in commit 878b2e.

4.3 The process of distributing ggp rewards

In order to receive a reward the staker must be registered for the required amount of time. But the current implementation of the protocol allows users to stake most of the funds immediately before distribution of the reward. The `isEligible` function verifies that the staker should be registered at least `ProtocolDAO.RewardsEligibilityMinSeconds` amount of seconds before the rewards cycle starts (this happens after the first minipool is created), but this check takes into account only the first staking, and the first staked amount may be minimal. Therefore, users can use this possibility to their advantage.

Remediation

The discussion point has been acknowledged by the Multisig Labs team. Their official reply is reproduced below:

> We acknowledge that this attack is possible and is a side effect of the nature of our rewards protocol and the short duration of validating on Avalanche. There is some cost and difficulty to exploiting this. It depends on one getting a large amount of GGP before a rewards cycle. If GGP is only available on one AMM, this would greatly move the price with no CEX to arbitrage against. The end result would most likely not be profitable to the attacker if their intention was to dump.

4.4 Checks-effects-interactions pattern

We recommend following the checks-effects-interactions pattern during the `claimAndRestake` function in the `ClaimNodeOp` contract by moving the `staking.decreaseGGPRewards(msg.sender, ggpRewards);` line above the external calls.

Remediation

The issue has been fixed by Multisig Labs in commit 750812.

4.5 Missing status update

In MinipoolManager the _cancelMinipoolAndReturnFunds function should reset the rewardsStartTime if the .minipoolCount value for staker is zero.

Remediation

The discussion point has been acknowledged by the Multisig Labs team. Their official reply is reproduced below:

> We don't think that resetting rewardsStartTime is the fix because of the scenario below.
>
> - Day 1: NodeOp1 creates minipool 1, and it gets launched. Reward startTime set to Day 1.
> - Day 14: Minipool 1 ends. mpCount = 0. But rewards is still Day 1 so we can get paid on day 28.
> - Day 15: NodeOp1 creates minipool 2, mpCount = 1
> - Day 15: NodeOp1 cancels it before launch. mpCount = 0. We can't reset rewards time because we need to get paid on Day 28. We DO reset the AVAXAssignedHighWatermark, so the AVAX used for this cancelled minipool doesn't count.
>
> Instead we remediated by splitting up avaxAssignedHighWater and avaxAssigned in this PR. Now the AVAX value used for rewards (avaxAssignedHighWater), will only be increased when the node is started in recordStakingStart

The issue has been remediated by Multisig Labs in PR 181.

4.6 Unused variables

In Storage, the intStorage and bytesStorage mappings and related functions are not used and can be deleted.

Remediation

The issue has ben acknowledged by the Multisig Labs and they plan to use them in the future.

4.7 Contract upgrades

We recommend paying additional attention when upgrading the contracts. Should the same `Storage` be used, the contract itself might not be `re-initializable` since its storage would already be used by the previously initialized contract. For example, this could happen in the `RewardsPool` contract.

```
function initialize() external onlyGuardian {
    if (getBool(keccak256(“RewardsPool.initialized”))) {
```

Notice that the `RewardPool.initialized` will always be true after the first contract has been initialized.

Remediation

The issue has ben acknowledged by the Multisig Labs. Their official reply is reproduced below:

> This is by-design. This specific contract was built to ensure even if upgraded that the `InflationIntervalStartTime` and `RewardsCycleStartTime` values would not be overwritten.

4.8 IWithdrawer inheritance

In the `withdrawAVAX` function from Vault, it is assumed that msg.sender has inherited the IWithdrawer interface. We consider that there could be a check for this during the registration process, since in `Vault`, for example, `withdrawAVAX` cannot be used (it will revert) unless `msg.sender` has the IWithdrawer interface implemented beforehand.

Remediation

The discussion point has been acknowledged by the Multisig Labs team. Their official reply is reproduced below:

> We added methods to register, unregister and upgrade contracts to the Protocol Dao. We'll add a check to our deploy scripts to handle verifying that we inherit from IWithdrawer.

4.9 Protocol DAO setters range

In protocol DAO, setters that deal with rates should range from 0.0 - 1.0 ether. This is not directly enforced as of now. The same could be done for the rest of the setter functions in the contract.

Remediation

The issue has been fixed by Multisig Labs in commit f49931.

4.10 Leftover tokens in `RewardsPool`

In the `startRewardsCycle`, the allotment each party is supposed to receive is calculated; however, due to the nature of the arithmetics, some tokens might be left out due to rounding errors.

Remediation

The issue has ben acknowledged by the Multisig Labs and they have determined that the amounts would not be significant.

5 Threat Model

The purpose of this section is to provide a full threat model description for each function.

As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

5.1 File: `TokenggAVAX`

Function: `initialize()`

Intended behavior:

- Should initialize all state variables and function calls required for the contract to function.

Branches and code coverage:

Intended branches:

- Should be callable by anyone?
  - □ Test coverage
- Should be called after every upgrade.
  - □ Test coverage

Negative behavior:

- Shouldn't allow 2 x calling this.
  - ☑ Negative test?

Preconditions:

- Assumes it's not callable by anyone, or that there's no way someone can front-run this transaction
- Assumes that the `Storage` is adequately configured (should be fine, since `guardian` role is assigned in the constructor, for the `msg.sender`

Inputs:

- `asset`:
  - **Control**: full control
  - **Checks**: no checks
  - **Impact**: used as underlying asset for the vault
- `storageAddress`:
  - **Control**: full control
  - **Checks**: no checks
  - **Impact**: used as upgradeable storage contract.

Function: `receive()`

Intended behavior:

This function is used for receiving native tokens. It can be called only by the `asset` address.

Branches and code coverage:

Intended branches:

- Allow `asset` contract to send native tokens to contract.
  ☑ Test coverage

Negative behavior:

- It cannot be called from any other address.
  ☑ Negative test?

Preconditions:

- the `asset` should be set after `initialize` call

Inputs:

- `msg.value`:
  - **Control**: controllable
  - **Authorization**: no
  - **Impact**: -
- `msg.sender`:
  - **Control**: controllable
  - **Authorization**: `assert(msg.sender == address(asset));`
  - **Impact**: `only accept AVAX via fallback from the WAVAX contract.` Oth-

erwise, the balance information may be out of sync.

External call analysis

There are no external calls here.

Function: `syncRewards()`

Intended behavior:

- Should "distribute rewards" to **TokenggAVAX** holders. Anyone may call this.

`lastSync` - time of last successful call to this function

`rewardsCycleEnd` - the time when the total reward will be available;

`totalReleasedAssets` - the full amount of available tokens for withdrawal + the last reward value from the previous cycle. If the reward was not withdrawn immediately after the end of the cycle when the function `syncRewards` is called for the next cycle, `lastRewardsAmt` value will be added to the value `totalReleasedAssets`, and this reward still will be available for withdrawal.

Branches and code coverage:

Intended branches:

- `rewardsCycleEnd` = deadline for next `rewardsCycle`
  ☐ Test coverage
- `lastSync` = current timestamp
  ☐ Test coverage
- `lastRewardsAmt_` = to the amount that rewards will deplete from.
  ☑ Test coverage
- `totalReleasedAssets` is calculated correctly for the next cycle - not sure that it is calculated correctly because it happens differently during `initialize` call
  ☐ Test coverage
- `lastRewardsAmt` is calculated for the next cycle if the new reward was deposited.
  ☐ Test coverage
- if rewards didn't deposit, the `lastRewardsAmt` will equal 0 for the next cycle
  ☐ Test coverage
- `lastRewardsAmt` is calculated correctly and equals 0 for the first cycle
  ☐ Test coverage
- if nothing changed since the past cycle `lastRewardsAmt` is calculated correctly and equals 0 and `totalReleasedAssets` was increased by the previous `lastRewardsAmt`

  - ☐ Test coverage
- current `block.timestamp` should be less than `rewardsCycleEnd`
  - ☑ Test coverage

Negative behavior:

- It basically shouldn't update unless stuff unless it's really time to update stuff(see below)
  - ☐ Negative test?
- Shouldn't allow calling unless the `rewardsCycle` has passed the `block.timestamp`.
  - ☑ Negative test?

Preconditions:

- Assumes that the state variables(`lastRewardsAmt`, `lastSync` , `rewardsCycleEnd` and `totalReleasedAssets` are properly updated)
- Can be called by anyone.

Inputs:

Function call analysis

- `asset.balanceOf(address(this))`
  - **What is controllable?** The amount of returned value
  - **If return value controllable, how is it used and how can it go wrong?** It can grow if the asset is artificially pumped in the contract;
  - **What happens if it reverts, reenters, or does other unusual control flow?** Doesn't revert.

Function: `totalAssets()`

Intended behavior:

- This function returns the total amount of underlying assets held by the vault.

Branches and code coverage:

Intended branches:

- After the current cycle ends and the new one starts, the `totalAssets` amount will contain the past `lastRewardsAmt` value.
  - ☐ Test coverage
- `totalAssets` is calculated correctly if the current cycle is going.
  - ☐ Test coverage

- If the current cycle ends and the new one doesn't start, the `totalAssets` should be equal `totalReleasedAssets_ + lastRewardsAmt`
  - □ Test coverage

Negative behavior:

- There's multiple types of uints there, should ensure that there's no way that any of them can overflow and block the functionality of the contract.
  - □ Negative test?
- must not revert(as per eip4626)
  - □ Negative test?

Preconditions:

- Assumes `lastSync` is different than `0` (default value, which is never initialized)? this is missing
- assumes that `block.timestamp` is safecasted? just as in syncRewards(currently missing)

Inputs:

There aren't input values here.

Function call analysis

There aren't function calls here.

Function: `depositFromStaking()`

Intended behavior:

- Should allow converting `native` AVAX tokens to `wAVAX` (just like `wETH`)
- Allows to `MinipoolManager` contract return withdrawn funds and deposit reward.
- It is assumed that, at first will be called `MinipoolManager.sol`:`createMinipool` function, which call `depositAVAX` and after that caller will be able to call `withdrawForStaking` for previously deposited value over `MinipoolManager.sol`: `claimAndInitiateStaking` and only after that `depositFromStaking` can be called over `recordStakingEnd` or `recordStakingError` functions from `MinipoolManager.sol`

Branches and code coverage:

Intended branches:

- the `asset` balance of the current contract will increase by the `msg.value` after the

call
  - ☐ Test coverage
- `stakingTotalAssets` will decrease by the `baseAmt` value after the call
  - ☐ Test coverage
- `baseAmt` + `rewardAmt` should be equal `msg.value`
  - ☐ Test coverage

Negative behavior:

- Shouldn't be callable by anyone(there's a check put in place, such that only `onlySpecificRegisteredContract` can call the function.
  - ☑ Negative test?
- if `stakingTotalAssets` is less than `baseAmt` transaction will be rejected
  - ☑ Negative test?

Preconditions:

- `stakingTotalAssets` should contain a value more or equal to `baseAmt`. It means that this value should have been withdrawn over `withdrawForStaking` function
- `msg.sender` should be approved for a call

Inputs:

- `msg.value`:
  - **Control**: controlled, but actually, it is the value from `getUint(keccak256(abi.encodePacked("minipool.item", minipoolIndex, ".avaxLiquidStakerAmt")));`
  - **Checks**: should be equal `baseAmt + rewardAmt`
  - **Impact**: -
- `msg.sender`:
  - **Control**: only approved MinipoolManager contract
  - **Checks**: `onlySpecificRegisteredContract("MinipoolManager", msg.sender)`
  - **Impact**: function should be called only from trusted `MinipoolManager` contract
- uint256 `rewardAmt`:
  - **Control**: partly controlled
  - **Checks**: `msg.value == baseAmt + rewardAmt`
  - **Impact**: -
- uint256 `baseAmt`:
  - **Control**: partly controlled

    - **Checks**: `msg.value == baseAmt + rewardAmt`
    - **Impact**: -

Function call analysis

- `IWAVAX(address(asset)).deposit{value: totalAmt}();`
    - **What is controllable?** the `totalAmt` is basically `msg.value`
    - **If return value controllable, how is it used and how can it go wrong?** na
    - **What happens if it reverts, reenters, or does other unusual control flow?** na

Function: `withdrawForStaking()`

Intended behavior:

- Should perform the `withdrawal` from `wAVAX` , for the `MinipoolManager`

Branches and code coverage:

Intended branches:

- `wAVAX.balanceOf(address(this)) -= assets` and `balanceOf(msg.sender) += assets`
    - □ Test coverage

Negative behavior:

- Shouldn't allow unlimited amount to be withdrawn
    - ☑ Negative test?
- Shouldn't be callable when it's `paused`(has the `whenNotPaused`) modifier
    - ☑ Negative test?
- if `assets` more than the `amountAvailableForStaking` transaction will be rejected
    - ☑ Negative test?
- if `asset.balanceOf(address(this))` is less than `assets` transaction will be rejected
    - □ Negative test?
- if `msg.sender` is not approved transaction will be rejected
    - □ Negative test?

Preconditions:

- Assumes that there has been some `depositFromStaking` beforehand.
- Assumes that the same `MinipoolManager` deposited the amount. And that there

cannot be any issues should one deposit and someone else (with same role) withdraw.

Inputs:

- `assets`:
  - **Control**: full control
  - **Checks**: assets > amountAvailableForStaking()
  - **Impact**: arbitrary input for the amount of `assets` that are to be withdrawn from the `wAVAX`
- `msg.sender`:
  - **Control**: only approved MinipoolManager contract
  - **Checks**: `onlySpecificRegisteredContract("MinipoolManager", msg.sender)`
  - **Impact**: since the caller can withdraw any amount of funds through this function, it is critically important that it is called only by a trusted contract.

Function call analysis

- `withdrawer.receiveWithdrawalAVAX{value: assets}();`
  - **What is controllable?** the `assets`, `withdrawer`; it basically calls the `receiveWithdrawalAVAX` on the `msg.sender`!!! Really important
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** reenters: no problems because the contract being called is trusted. reverts: no problems
- `IWAVAX(address(asset)).withdraw(assets);`
  - **What is controllable?** the `assets` value; the `asset` address is state var
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: `depositAVAX` compare with `deposit()` from inherited

Intended behavior:

- Allows any user to deposit `AVAX` in exchange for `wAVAX`. It basically doesn't transfer the `wAVAX` back to the user, it keeps it and issues shares to the user.

Branches and code coverage:

Intended branches:

- `previewDeposit` should issue the amount of shares correctly!!
  ☑ Test coverage
- Should transfer the `wAVAX` back to the user.
  ☑ Test coverage
- Should exchange user's supplied `AVAX` into `wAVAX`
  ☑ Test coverage

Negative behavior:

- Shouldn't issue more or less shares than intended.
  ☑ Negative test?

Preconditions:

- Assumes users would use this function to deposit, rather than depositing on their own.
- Assumes `previewDeposit` calculates the amount of shares correctly.

Inputs:

- `IWAVAX(address(asset)).deposit()`
  - **What is controllable?** `assets` – the amount of deposited native tokens
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `afterDeposit()`
  - **What is controllable?** `assets` – the amount of deposited native tokens
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `_mint()`
  - **What is controllable?** `msg.sender` – is minted tokens receiver address
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

- `previewDeposit()` · `convertToShares()`
  - **What is controllable?** `assets` - the amount of deposited native tokens
  - **If return value controllable, how is it used and how can it go wrong?** if there are any mistakes during `shares` value calculations, then caller will get more or less shares than expected. If more then caller will be able to drain other users funds, if less then caller will withdraw less native tokens that was deposited.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: `withdrawAvax()` compare this with `withdraw()` from inherited

Intended behavior:

- Supposed to withdraw `wAVAX` on behalf of the `msg.sender`, and then transfer the native `AVAX` back to the `msg.sender`.

Branches and code coverage:

Intended branches:

- the `wavax` balance of the contract should decrease(by `assets`)
  ☑ Test coverage
- the `avax` balance of user should increase(by `assets`)
  □ Test coverage
- the `shares` of the user should decrease(by `shares`)
  ☑ Test coverage
- make sure that `preivewWithdraw` calculates the shares properly, in all market conditions
  □ Test coverage

Negative behavior:

- shouldn't allow withdrawing if `_burn` reverted
  □ Negative test?
- shouldn't allow burning on behalf of other users
  ☑ Negative test?

Preconditions:

- Assumes there are no rounding errors in `previewWithdraw` or other similar arithmetic issues.
- Assumes that user has enough `shares` to actually withdraw enough `wAVAX`

Inputs:

- `assets`:
  - **Control**: full control; the amount of assets that the user `intends` to withdraw.
  - **Checks**: there is no check here, however, it's assumed that `previewWithdraw` calculates the amount of shares properly, and then that `_burn` fails should the `msg.sender` not have enough shares to actually receive the amount of `assets`.
  - **Impact**: arbitrary input for the amount of `assets` that are to be withdrawn from the `wAVAX`
- `msg.sender`:
  - **Control**: any caller
  - **Checks**: must have the appropriate amount of shares
  - **Impact**: the caller will receive the appropriate amount of native tokens

Function call analysis

- `previewWithdraw(assets)`
  - **What is controllable?** the assets parameter;
  - **If return value controllable, how is it used and how can it go wrong?** return the amount of shares. In case of wrong calculations a caller can burn an excessive number of shares or, conversely, burn too few and receive disproportionately many native tokens.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `IWAVAX(address(asset)).withdraw(assets);`
  - **What is controllable?** the `assets` value; the `asset` address is state var
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: `redeemAVAX()` compare with `redeem()` from inherited

Intended behavior:

- Should redeem the `shares` for underlying native `avax`. Similar to how `withdraw` works.

Branches and code coverage:

- No test coverage

Intended branches:

- `assets` value is calculated correcly
  - □ Test coverage
- `totalReleasedAssets` is decreased by `assets` value
  - □ Test coverage
- `msg.sender` received the `assets` amount of native tokens
  - □ Test coverage
- token gg balance of `msg.sender` is decreased by `shares` value
  - □ Test coverage

Negative behavior:

- shouldn't allow withdrawing if `_burn` reverted
  - ☑ Negative test?
- shouldn't allow burning on behalf of other users
  - □ Negative test?
- revert if `contract.paused` is `True`
  - ☑ Negative test?

Preconditions:

- Assumes that user has enough `shares` to burn.

Inputs:

- `shares`:
  - **Control**: controlled
  - **Checks**: balance of `msg.sender` should be more or equal of `shares` amount
  - **Impact**: the number of gg tokens that the user can burn and receive a certain number of native tokens.
- `msg.sender`:
  - **Control**: any caller
  - **Checks**: must have the appropriate amount of shares
  - **Impact**: the caller will receive the appropriate amount of native tokens

Function call analysis

- `previewRedeem(shares)`

  - **What is controllable?** the shares parameter;
  - **If return value controllable, how is it used and how can it go wrong?** return the amount of assets. In case of wrong calculations a caller can receive a lot (thereby stealing other users funds) or, conversely, too few native tokens.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `IWAVAX(address(asset)).withdraw(assets);`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

5.2 File: `ClaimNodeOP`

Function: `calculateAndDistributeRewards()`

Intended behavior:

- Set the share of rewards that a `staker` is owed.(Fraction of 1 ether)

Branches and code coverage:

Lacks extensive testing.

Intended branches:

- Update the `rewardsCycleCount` of staker.
  - ☐ Test coverage
- Ensure calculations are properly performed.
  - ☐ Test coverage
- Increase the `ggpRewards` for the `stakerAddr` based on the input `totalEligibleGGPStaked`.
  - ☑ Test coverage

Negative behavior:

- Should fail if `stakerAddr` is not eligible for rewards.
  - ☑ Negative test?

Preconditions:

- Assumes `stakerAddr` is a valid one.

- Assumes that the caller has used the correct `totalEligibleGGPStaked` amount.

Inputs:

- `msg.sender`:
  - **Control**: -
  - **Checks**: onlyMultisig
  - **Impact**: the access to this function should be restricted because this function allows to assign any part of reward budget to any `stakerAddr`.
- `stakerAddr`:
  - **Control**: full control
  - **Checks**: no checks at this level; But will revert during the `increaseGGPRewards` function call.
  - **Impact**: the address of valid staker who can claim the reward.
- `totalEligibleGGPStaked`:
  - **Control**: full control
  - **Checks**: there aren't checks
  - **Impact**: the total amount of staked funds, from which the percentage of reward to `stakerAddr` will be calculated. So this value allow to control the reward part for `stakerAddr`.

Function call analysis

- `staking.getLastRewardsCycleCompleted(stakerAddr)`
  - **What is controllable?** `stakerAddr` is controllable
  - **If return value controllable, how is it used and how can it go wrong?** if someone will be able to manipulate `lastRewardsCycleCompleted` value, the `stakerAddr` will be able to double receive the reward.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `staking.getEffectiveGGPStaked(stakerAddr);`
  - **What is controllable?** `stakerAddr` is controllable
  - **If return value controllable, how is it used and how can it go wrong?** the amount of staked tokens is used to calculate the percentage of the total staked tokens.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problem
- `staking.setLastRewardsCycleCompleted(stakerAddr, rewardsPool.getRewardsCycleCount());`
  - **What is controllable?** `stakerAddr` is controllable

  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problem
- `staking.resetAVAXAssignedHighWater(stakerAddr);`
  - **What is controllable?** `stakerAddr` is controllable
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problem
- `staking.increaseGGPRewards(stakerAddr, rewardsAmt);`
  - **What is controllable?** `stakerAddr` is controllable
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problem
- `staking.setRewardsStartTime(stakerAddr, 0);`
  - **What is controllable?** `stakerAddr` is controllable
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problem

Function: `claimAndRestake()`

Intended behavior:

- Allows `msg.sender` to claim the `rewards` they were allocated.

Branches and code coverage:

Lacks extensive testing.

Intended branches:

- Should decrease rewards balance of `msg.sender`
  ☑ Test coverage
- Restake the amount of `ggpRewards - claimAmt`
  □ Test coverage

Negative behavior:

- Should not allow claiming more than `msg.sender` was owed

☑ Negative test?

Preconditions:

- Assumes `msg.sender` has some rewards
- Assume that the `vault` holds enough tokens to pay the rewards for `msg.sender`.

Inputs:

- `msg.sender`:
  - **Control**: -
  - **Checks**: if the `ggpRewards` value is zero, will revert.
  - **Impact**: the address who owns non zero reward value.
- `claimAmt`:
  - **Control**: full control
  - **Checks**: should not be more that the reward: claimAmt > ggpRewards
  - **Impact**: the amount of withdrawn funds, the surplus will be restake.

Function call analysis

- `vault.withdrawToken(address(this), ggp, restakeAmt)`
  - **What is controllable?** `restakeAmt` is controllable
  - **If return value controllable, how is it used and how can it go wrong?** there is no return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if there are not enough tokens.
- `staking.getGGPRewards(msg.sender)`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** return value is used for calculating the amount of rewards that `msg.sender` is owed.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

5.3 File: `ClaimProtocolDAO.sol`

Function: `spend()`

Intended behavior:

Allows to spend the ProtocolDAO's GGP rewards

Branches and code coverage:

Intended branches:

- The balance of `recipientAddress` is increased by `amount`; there is a revert put in place in case `transfer` fails.
  - ☑ Test coverage

Negative behavior:

- should be rejected if this contract has not enough ggp tokens in the `vault.tokenBalance`
  - ☑ Negative test?
- should reject if msg.sender isn't the guardian
  - ☑ Negative test?

Preconditions:

- msg.sender is the guardian
- tokens should be transferred to `ClaimProtocolDAO` contract over the `vault.transferToken` function

Inputs:

- `amount`:
  - **Control**: limited control
  - **Checks**: amount == 0 || amount > vault.balanceOfToken("ClaimProtocolDAO", ggpToken)
  - **Impact**:
- `recipientAddress`:
  - **Control**: controlled
  - **Checks**: there aren't checks here
  - **Impact**: since there are no address checks, in case of a mistake, tokens can be transferred to the wrong user.
- `invoiceID`:
  - **Control**: controlled
  - **Checks**: there aren't checks here
  - **Impact**: no impact
- `msg.sender`:
  - **Control**: -
  - **Checks**: onlyGuardian
  - **Impact**: it allows caller to withdraw the entire balance of ggpToken of this

contract from vault. The access to this function should be restricted.

Function call analysis

- `vault.withdrawToken()`
  - **What is controllable?** recipientAddress, amount
  - **If return value controllable, how is it used and how can it go wrong?** there is no return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if msg.sender doesn't have enough tokens

5.4 File: `BaseUpgradeable.sol`

The contract is inherited from `BaseAbstract.sol` and `Initializable.sol` (@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol);

Function: `__BaseUpgradeable_init()`

Allows to initialize the `gogoStorage` storage address. The function is internal and can be called only once due to `onlyInitializing` modifier.

5.5 File: `Base.sol`

The contract is inherited from `BaseAbstract.sol`; The contract contains only `constructor` with initialization of `gogoStorage` address.

5.6 File: `BaseAbstract.sol`

Function: `setters()`

Intended behavior:

Allows you to make changes to the data stored in the shared storage. All function is internal, therefore, they cannot be called directly. But they are called from various functions from inherited contracts.

5.7 File: `Storage.sol`

Function: `setGuardian()`

Intended behavior: Allow to reassign the guardian address. But to complete this process the new guardian should call `confirmGuardian` function.

Branches and code coverage:

Intended branches:

- After successful call the `guardian` address didn't change.
  - ☑ Test coverage

Negative behavior:

- Reject if `msg.sender` isn't the guardian; check put in place.
  - ☑ Negative test?

Preconditions:

msg.sender is current guardian.

Inputs:

- `msg.sender`:
  - **Control**: -
  - **Checks**: msg.sender != guardian
  - **Impact**: due to the guardian having a lot of control over the protocol, it's critically important that an untrusted caller doesn't have access to this function.

Function call analysis

There aren't external calls here.

Function: `confirmGuardian()`

Intended behavior: Allow to reassign the guardian address. But to complete this process the new guardian should call `confirmGuardian` function.

Branches and code coverage:

Intended branches:

- After successful call the `guardian` address is equal to `msg.sender` and `newGuardian`.

☑ Test coverage

Negative behavior:

- Reject if `msg.sender` isn't the newGuardian; check put in place.
  ☑ Negative test?

Preconditions:

The current guardian called the `setGuardian` function and `msg.sender` became the `newGuardian`.

Inputs:

- `msg.sender`:
  - **Control**: -
  - **Checks**: msg.sender != `newGuardian`
  - **Impact**: due to the guardian having a lot of control over the protocol, it's critically important that an untrusted caller doesn't have access to this function.

Function call analysis

There aren't external calls here.

Function: `setters()`

Intended behavior:

- Should be used among more contracts as a shared means of storage

Branches and code coverage:

Intended branches:

- Should update the {type} of value located at each particular key.
  ☐ Test coverage; Limited test coverage

Negative behavior:

- Network registered contracts shouldn't abuse the `booleanStorage[keccak256(abi.encodePacked("contract.exists", msg.sender))]` modifier. Basically once a contract is `whitelisted`, it can remove/register other contracts as `network registered`, or modify any other states altogether.
  ☐ Negative test?

Preconditions:

- Assumes that `msg.sender` handles the states properly, and doesn't have typos when reading / updating specific states. Basically all functions that interact with the getters/ setters/ deleters from other contracts should be extremely well tested.

5.8 File: `TokenGGP.sol`

The contract is standard `ERC20` from `@rari-capital/solmate/src/tokens/ERC20.sol`.

5.9 File: `Vault.sol`

Function: `depositAVAX()`

Intended behavior:

Allows registered contract to deposit avax.

Branches and code coverage:

Intended branches:

- `avaxBalances` of `msg.sender` increased by `msg.value`
  ☑ Test coverage

Negative behavior:

- if `msg.sender` is not `RegisteredNetworkContract` transaction will be reverted
  ☑ Negative test?
- if msg.value == 0, will be reverted
  ☑ Negative test?

Preconditions:

- msg.sender should be registered by the guardian

Inputs:

- `msg.sender`:
  - **Control**: -
  - **Checks**: onlyRegisteredNetworkContract
  - **Impact**: no impact

- `msg.value`:
  - **Control**: limited control
  - **Checks**: msg.value == 0
  - **Impact**: no impact

Function call analysis

There aren't external calls here.

Function: `withdrawAVAX()`

Intended behavior:

Allows registered contract to withdraw the deposited avax.

Branches and code coverage:

Intended branches:

- after the call `avaxBalances[msg.sender]` decreased by `amount`
  ☑ Test coverage

Negative behavior:

- if `msg.sender` is not `RegisteredNetworkContract` transaction will be reverted
  ☐ Negative test?
- if `avaxBalances[msg.sender] < amount`, transaction will be reverted
  ☑ Negative test?

Preconditions:

- avaxBalances of msg.sender · amount
- msg.sender should be registered contract by guardian

Inputs:

- `msg.sender`:
  - **Control**: -
  - **Checks**: onlyRegisteredNetworkContract
  - **Impact**: should has non zero balance for withdraw
- `amount`:
  - **Control**: controlled
  - **Checks**: avaxBalances[getContractName(msg.sender)] < amount
  - **Impact**: must withdraw only his tokens

Function call analysis

- `withdrawer.receiveWithdrawalAVAX()`
  - **What is controllable?** amount - partly controlled, the `avaxBalances[msg.sender] ≥ amount`
  - **If return value controllable, how is it used and how can it go wrong?** there isn't a return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** function is nonReentrant and state is updated before the external call.

Function: `transferAVAX()`

Intended behavior:

Allows transferring the balance from one registered contract to another.

Allows a transfer, not from the owner, and there is also no check for an allowance from the owner

Branches and code coverage:

Intended branches:

- `avaxBalances[toContractName]` is increased `amount`
  ☑ Test coverage
- `avaxBalances[fromContractName]` is decreased by `amount`
  ☑ Test coverage

Negative behavior:

- Should be rejected if `avaxBalances[fromContractName] < amount`
  ☑ Negative test?
- Should be rejected if `toContractName` and `fromContractName` is not added to gogoStorage
  ☑ Negative test?
- Should be rejected if msg.sender is not `fromContractName`
  □ Negative test?

Preconditions:

- `toContractName` and `fromContractName` is added to gogoStorage
- msg.sender is RegisteredNetworkContract
- `avaxBalances[fromContractName] · amount`

Inputs:

- `toContractName`:
  - **Control**: controlled
  - **Checks**: contract name should be saved inside gogoStorage
  - **Impact**: in the case of an incorrect recipient, funds may be lost.
- `fromContractName`:
  - **Control**: controlled
  - **Checks**: contract name should be saved inside gogoStorage
  - **Impact**: the contract which funds will be transferred, in this case the msg.sender has full control
- `msg.sender`:
  - **Control**: -
  - **Checks**: onlyRegisteredNetworkContract
  - **Impact**: -
- `amount`:
  - **Control**: controlled
  - **Checks**: `avaxBalances[fromContractName] < amount`
  - **Impact**: -

Function call analysis

There aren't external calls here.

Function: `depositToken()`

Intended behavior:

Allows registered contract to deposit any tokens

Branches and code coverage:

Intended branches:

- `tokenBalances` of networkContractName·contractKey is increased by `amount`
  ☑ Test coverage

Negative behavior:

- Should reject if `msg.sender` is not `guardianOrRegisteredContracts`
  ☑ Negative test?

Preconditions:

- msg.sender has enough tokens
- msg.sender is guardianOrRegisteredContracts

Inputs:

- `amount`:
  - **Control**: limited control
  - **Checks**: amount == 0
  - **Impact**: no problems
- `tokenContract`:
  - **Control**: full control
  - **Checks**: there isn't checks here
  - **Impact**: address of external contract to be called
- `networkContractName`:
  - **Control**: limited control
  - **Checks**: contract name should be saved inside gogoStorage
  - **Impact**: the recipient of tokens, in the case of an incorrect recipient, funds may be lost.

Function call analysis

- `tokenContract.safeTransferFrom()`
  - **What is controllable?** `amount`
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if `msg.sender` doesn't have enough tokens

Function: `withdrawToken()`

Intended behavior:

- Allow `registered` `msg.sender` to withdraw ERC20 tokens.

Branches and code coverage:

Intended branches:

- Check `withdrawalAddress`?
  - ☑ Test coverage

- Decrease `tokenBalance[paid(caller, token)]`
  ☑ Test coverage
- Validate the `tokenContract`, such that no arbitrary tokens can be used.
  □ Test coverage

Negative behavior:

- Shouldn't allow withdrawing more than `msg.sender` owns.
  ☑ Negative test?

Preconditions:

- Assumes `msg.sender` is registered;
- Assumes that the `tokenAddress` is legit and not some malicious token

Inputs:

- `withdrawalAddress`:
  - **Control**: full control
  - **Checks**: no checks
  - **Impact**: in the case of an incorrect recipient, funds may be lost.
- `tokenAddress`:
  - **Control**: full control
  - **Checks**: no checks
  - **Impact**: should allow to pass only trusted contracts.
- `amount`:
  - **Control**: limited control
  - **Checks**: check that it's ≠`0` and that user has more balance than it.
  - **Impact**: shouldn't allow to pass more tokens amount than caller owns.

Function call analysis

- `tokenContract.safeTransfer(withdrawalAddress, amount)`
  - **What is controllable?** `withdrawalAddress`, `amount`
  - **If return value controllable, how is it used and how can it go wrong?** no checks on `withdrawalAddress`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if `msg.sender` doesn't have enough tokens

Function: `transferToken()`

Intended behavior:

- Transfer token from one contract(msg.sender) to another

Branches and code coverage:

Intended branches:

- Validate the `tokenContract`, such that no arbitrary tokens can be used.
  - □ Test coverage
- Assure both contracts are registered.
  - ☑ Test coverage
- Compared to the `transferAVAX`, this function does not allow the transfer `from` arbitrary tokens, and only from `msg.sender`
  - □ Test coverage
- Increase `tokenBalances[to]` **AND** decrease `tokenBalances[from]` .
  - ☑ Test coverage

Negative behavior:

- Revert if `msg.sender` is not a registered contract.
  - x Test coverage
- Revert if `msg.sender` doesn't have enough tokens amount.
  - Test coverage

Preconditions:

- Assumes both contracts have been registered beforehand.

Inputs:

- `networkContractName`:
  - **Control**: full
  - **Checks**: check that it's registered
  - **Impact**: in the case of an incorrect recipient, funds may be lost.
- `tokenAddress`:
  - **Control**: full control
  - **Checks**: No checks! Any token
  - **Impact**: should allow to pass only trusted contract address.

Function call analysis

There aren't external calls here.

5.10 File: `MinipoolManager`

Function: `createMinipool()`

Intended behavior:

- Create a Minipool. Accepts `avax` native deposit(which have to be staked in) and it's open to public.
- Allows to any caller to recreate a minipool is current state is finished or canceled.

Branches and code coverage:

Intended branches:

- Ensure that the `msg.sender` is a registered staker(required checks are added in each underlying function)
  ☑ Test coverage
- Should ensure that the `avaxAssignmentRequest` can be fulfilled (or that it is at least achievable)
  ☐ Test coverage
- User's `avax` balance should deplete, and the contract's balance should increase.
  ☐ Test coverage
- After the call, the current state of the minipool is `Prelaunch`
  ☑ Test coverage
- native token balance of `assets` should increase by `msg.value`
  ☑ Test coverage
- `assets` balance of `vault` contract should increase by `msg.value`
  ☑ Test coverage
- if the pool for `nodeID` exists and the current state is Finished or Canceled, minipool data should be reset
  ☐ Test coverage
- create a new poll if the pool for `nodeID` did not exist before
  ☐ Test coverage
- `Staking.sol:getRewardsStartTime(msg.sender)` should be equal `block.timestamp` if `RewardsStartTime` was zero before the call
  ☑ Test coverage
- `Staking.sol:getMinipoolCount(msg.sender)` should increase by 1
  ☑ Test coverage
- `Staking.sol:getAVAXAssigned(msg.sender)` should increase by `avaxAssignmentRequest`
  ☑ Test coverage
- `Staking.sol:getAVAXStake(msg.sender)` should increase by `msg.value`

☑ Test coverage

Negative behavior:

- Shouldn't work when the contract is paused?/
  ☑ Negative test? There isn't test, but function has modifier `whenNotPaused`
- Should assure that the `nodeId` hasn't registered beforehand and is unique basically, so no overwrites can be made.
  □ Negative test?
- should revert if minipool for `nodeID` already exists and the currentStatus · Finished or currentStatus · Canceled
  □ Negative test?
- should revert if `msg.sender` invalid staker
  ☑ Negative test?

Preconditions:

- Assumes that the supplied `msg.value` surpasses the minimum staking amount.
- Assumes that the `multisig` that is to be assigned is `≠ 0`.
- Assumes that should the `miniPool` exist, it can only be overwritten if the node is either finished or cancelled.
- In the case that an already existing `miniPool`Id exists, it assumes that ALL PRIOR STATES HAVE BEEN RESET(FROM ALL CONTRACTS THAT WOULD HAVE INTERACTED WITH THIS ONE IN THE FIRST)
- `msg.sender` should be registered staker
- msg.sender should stake ggp over Staking.stakeGGP() function

Inputs:

- `msg.sender`:
  - **Control**: controlled
  - **Checks**: staking.increaseAVAXStake() · requireValidStaker() checks msg.sender address (should stake ggp over stakeGGP() function)
  - **Impact**: N/A
- `msg.value`:
  - **Control**: N/A
  - **Checks**: `msg.value should be equal avaxAssignmentRequest`
  - **Impact**: N/A
- `nodeId`:
  - **Control**: full control
  - **Checks**: there are some checks on whether the `nodeID` has been registered

before; need to look into this
  - **Impact**: could potentially be overwritten.
- `duration`:
  - **Control**: full control
  - **Checks**: There are no checks on the duration amount
  - **Impact**: N/A
- `delegationfee`:
  - **Control**: full
  - **Checks**: No checks
  - **Impact**: N/A
- `avaxAssignmentRequest`:
  - **Control**: full control; needs to match `msg.value` since it's the amount of requested `AVAX` TO BE MATCHED IN THE POOL.
  - **Checks**: there are checks on whether it matches `msg.value`
- there are also some checks on whether it matches the `dao details` ; assure that the data returned from there is not `0`?
  - **Impact**: N/A

Function call analysis

!!! Important functions(withdraw/ deposit/ etc) shouldn't work when the contract is `paused`.

- `vault.depositAVAX()`
  - **What is controllable?** msg.value
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `getCollateralizationRatio()`
  - **What is controllable?** msg.sender
  - **If return value controllable, how is it used and how can it go wrong?** The returns collateralization ratio also depends on how much msg.sender deposited ggp
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `increaseMinipoolCount()`
  - **What is controllable?** msg.sender (had to deposit ggp before)
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value

  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `increaseAVAXAssigned()`
  - **What is controllable?** msg.sender (had to deposit ggp before), avaxAssignmentRequest
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `increaseAVAXStake()`
  - **What is controllable?** msg.sender (had to deposit ggp before), msg.value
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: `cancelMinipool()`

Intended behavior:

- Allows owner to cancel existing minipool and get back the deposited funds.

Branches and code coverage:

Intended branches:

- Should update all details related to the specific `nodeId`. In such a way that one can then be re-used eventually(create with same `nodeId`)
  ☑ Test coverage
- Refund all invested funds to the `owner`(deployer)
  □ Test coverage
- Make sure that the minipool is `prelaunch` (NOT CHECKED); it's assured though in `requireValidStateTransition` basically, since it checks the current status against the wanted status update.
  □ Test coverage
- `Staking.sol:getAVAXAssigned(msg.sender)` should decrease by `avaxLiquidStakerAmt`
  ☑ Test coverage
- `Staking.sol:getAVAXStake(msg.sender)` should decrease by `avaxNodeOpAmt`
  ☑ Test coverage
- `Staking.sol:getMinipoolCount(msg.sender)` should decrease by 1
  ☑ Test coverage

- the native tokens balance of the caller should increase by the amount of funds previously deposited.
  ☑ Test coverage
- After the call, the current state of the minipool is `Canceled`
  ☑ Test coverage

Negative behavior:

- Shouldn't leave previously `set` fields to their value(eg. the `avaxLiquidStakerAmt`)
  ☑ Negative test?
- Shouldn't allow unauthorized access(`msg.sender` has to be the `owner`)
  ☑ Negative test?
- should revert if the current state of mini pool isn't `Prelaunch`
  ☑ Negative test?
- should revert if called non-owner of minipool
  ☑ Negative test?
- should revert if minipool for nodeID doesn't exist
  ☑ Negative test?

Preconditions:

- the minipool should be created over the `createMinipool` function
- the current state of the minipool should be `Prelaunch`
- Assumes that the `nodeId` has been created beforehand and that it's in the `prelaunch` stage
- Assumes that the `owner` of the `nodeID` calls it

Inputs:

- `nodeId`:
  - **Control**: full control
  - **Checks**: there's a check on whether the minipool is valid.
  - **Impact**: Id of minipool which will be canceled and funds will returned to owner.
- `msg.sender`:
  - **Control**: onlyOwner of minipool can call
  - **Checks**: `onlyOwner(index);`
  - **Impact**: only the owner should be able to call this function. otherwise, users will maliciously close other people's pools to get more rewards.

Function call analysis

- `_cancelMinipoolAndReturnFunds()`
  - **What is controllable?** the `nodeID` is controllable.
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** can be reverted if there aren't enough native tokens for withdraw.
- `owner.safeTransferETH()`
  - **What is controllable?** nothing controllable
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `vault.withdrawAVAX()`
  - **What is controllable?** nothing controllable
  - **If return value controllable, how is it used and how can it go wrong?** there isn't a return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if contract has not enough shares

Function: `_cancelMinipoolAndReturnFunds()`

Intended behavior:

- Internal function.
- Main logic of cancelling a minipool and returning the funds that were initially attributed to it.

Branches and code coverage:

Intended branches:

- Ensure that all states are reset after a `Minipool` has been cancelled and that `owner` no longer has access to it.
  ☐ Test coverage
- Ensure that current state allows `cancellation`.
  ☑ Test coverage
- Ensure that `avaxNodeOpAmt` is decreased.
  ☑ Test coverage
- Ensure that `avaxLiquidStakerAmt` is decreased
  ☑ Test coverage

Negative behavior:

- Shouldn't allow `cancellation` if the current state ≠ `prelaunch`
  ☑ Negative test?
- Shouldn't allow cancellation on behalf of `msg.sender` ≠ `owner`
  ☑ Negative test?

Preconditions:

- Assumes that the function has been called from a privileged one(i.e one that has a check that `msg.sender == owner of market`)

Inputs:

- `nodeID`:
  - **Control**: full control
  - **Checks**: no checks at this level
  - **Impact**: nothing is done on the `nodeId` at this level, so not that important
- `index`:
  - **Control**: full control(it's generated in previous function)
  - **Checks**: no checks
  - **Impact**: important, as it allows altering states of the `minipool`

Function call analysis

- `decreaseAVAXStake()`
  - **What is controllable?** the `owner` (who's supposed to be the caller of the function)
- it basically decreases the `avaxNodeOpAmt` value which is originally `increased` in the pool creation! The detail here is that it uses `.avaxNodeOpAmount` to store the `amount`, while it decreases the `avaxNodeOpAmt`
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?**
- if it reverts it could affect cancelling the pool. (that's why it's better to only use one type of amount ˆ)
- `decreaseAVAXAssigned()`
  - **What is controllable?** nothing, the values are taken from the storage.
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?**

    if current `avaxAssigned` is not enough function will be reverted

- `resetAVAXAssignedHighWater()`
  - **What is controllable?** nothing, the value is taken from the storage.
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** allows to set the `avaxAssignedHighWater` to the previous value, so that the current value is not used when calculating the reward.
- `decreaseMinipoolCount()`
  - **What is controllable?** nothing, the value is taken from the storage.
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** reduces the number of pools, if it is reset to zero, this staker will not be taken into account when calculating the reward.

Function: `withdrawMinipoolFunds()`

Intended behavior:

- Node operator can claim all `avax` they are due.(staked + rewards if any)

Branches and code coverage:

LIMITED TESTING

Intended branches:

- Should decrease `msg.sender` stake in the `minipool` by `avaxNodeOpAmt`

  ☐ Test coverage
- the native tokens balance of minipool owner should increase by `totalAvaxAmt` value (deposited amount + reward)

  ☑ Test coverage

Negative behavior:

- Shouldn't be callable by any `msg.sender` or on any `nodeId`

  ☑ Negative test?
- should revert if the owner calls it a second time after the successful first execution

  ☐ Negative test?
- should revert if the current state of mini pool isn't `Withdrawable` or `Error`

  ☐ Negative test?

- should revert if called non-owner of minipool
  - ☑ Negative test? There isn't test, but there is a check `onlyOwner` inside the function
- should revert if minipoll for nodeID doesn't exist
  - ☑ Negative test? There isn't test, but there is a check `requireValidMinipool` inside the function

Preconditions:

- The minipool should be created over the `createMinipool` function.
- Assumes that the state can transition to `finished`, and that the current state of the minipool should be `Withdrawable` (after `recordStakingEnd` call) or `Error`.

Inputs:

- `msg.sender`:
  - **Control**: -
  - **Checks**: there is a check that msg.sender is owner of minipool
  - **Impact**: allows to owner of minipool withdraw funds when staking finished
- `nodeID`:
  - **Control**: controlled
  - **Checks**: there are a check of the status of the minipool and a check of the owner
  - **Impact**: allows to return the funds to the owner of minipool if staking was finished

Function call analysis

- `owner.safeTransferETH()`
  - **What is controllable?** nothing controllable
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `vault.withdrawAVAX()`
  - **What is controllable?** nothing controllable
  - **If return value controllable, how is it used and how can it go wrong?** there isn't a return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if contract has not enough shares

Function: `claimAndInitiateStaking()`

Intended behavior:

- Remove the `minipool's` avax from the protocol and stake it on avalanche, register node as validator.

Branches and code coverage:

Intended branches:

- Ensure only `multisig rialto` can call this.
  ☑ Test coverage
- Should ensure the status of the `minipool` is such that it can be launched
  ☑ Test coverage
- Should decrease the `avax` associated to the pool(something with `.avaxLiquidStakerAmt`)
  ☐ Test coverage

Negative behavior:

- transaction should be rejected if current status · `Prelaunch`
  ☑ Negative test?
- transaction should be rejected if msg.sender isn't approved address
  ☑ Negative test?

Preconditions:

- Assumes that contract has enough `wavax` staked that can be withdrawable.

Inputs:

- `msg.sender`:
  - **Control**: -
  - **Checks**: `onlyValidMultisig(nodeID) : msg.sender == assignedMultisig`
  - **Impact**: only valid multisig can call this function, because the all deposit funds will be transferred to caller.
- `nodeID`:
  - **Control**: full control
  - **Checks**: `requireValidMinipool(nodeID)`
  - **Impact**: no impact

Function call analysis

- `msg.sender.safeTransferETH()`
  - **What is controllable?** msg.sender is controlled
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert in case of error
- `vault.withdrawAVAX()`
  - **What is controllable?** nothing is controlled
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** allows to withdraw `avaxNodeOpAmt` from vault and transfer this funds to caller
- `ggAVAX.withdrawForStaking()`
  - **What is controllable?** nothing is controlled
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** allows to withdraw `avaxLiquidStakerAmt` from vault and transfer this funds to caller

Function: `recordStakingStart()`

Intended behavior:

- Rialto calls after `claimAndInitiateStaking` succeeded.

Branches and code coverage:

Intended branches:

- Changes the `starttime`. Make sure it's not in past or future?
  - ☐ Test coverage
- Should transition a `nodeID` into "staking" period.
  - ☑ Test coverage

Negative behavior:

- Anyone other than `rialto` shouldn't be able to call this.
  - ☑ Negative test?
- transaction should be rejected if current status · `Launched`

- ☑ Negative test?

Preconditions:

- Has to assure that enough values are in the `minipool`

Inputs:

- `startTime`:
  - **Control**: controllable
  - **Checks**: there isn't check
  - **Impact**: if the value is far in the future it will be impossible to complete the stacking successfully only with error state
- `txID`:
  - **Control**: controllable
  - **Checks**: there isn't check
  - **Impact**: n/a
- `nodeID`:
  - **Control**: partly controllable
  - **Checks**: `requireValidMinipool(nodeID)`
  - **Impact**: n/a
- `msg.sender`:
  - **Control**: -
  - **Checks**: `onlyValidMultisig(nodeID) : msg.sender == assignedMultisig`
  - **Impact**: if a malicious user is able to call the function, he will be able to set startTime value, at which it will be impossible to successfully complete the stacking with only an error state

Function call analysis

There aren't external function calls here.

Function: `recordStakingEnd()`

Intended behavior:

- Finish the `validation` period of the `staking` for the `nodeid`.

Branches and code coverage:

Intended branches:

- Should update all states accordingly after the transfers occur.
  - ☑ Test coverage
- End time should be in the future(`starttime` and not in past compared to `block.timestamp`?)
  - ☐ Test coverage
- Should only be callable when the `endtime` is reached.
  - ☑ Test coverage

Negative behavior:

- Shouldn't be callable twice or in any other circumstance other than the transition to `withdrawable`
  - ☑ Negative test?
- transaction should be rejected if msg.value is not enought
  - ☑ Negative test?
- transaction should be rejected if msg.sender isn't approved address
  - ☑ Negative test?
- transaction should be rejected if current status · `Staking`
  - ☑ Negative test?

Preconditions:

- the current state of the minipool should be `Staking`.

Inputs:

- `msg.value`:
  - **Control**: -
  - **Checks**: `msg.value` should be equal `totalAvaxAmt + avaxTotalRewardAmt`
  - **Impact**:
- `avaxTotalRewardAmt`:
  - **Control**: full control
  - **Checks**: `msg.value` should be equal `totalAvaxAmt + avaxTotalRewardAmt`
  - **Impact**: the value completely controls how much reward the owner of the pool will receive.
- `endTime`:
  - **Control**: controllable
  - **Checks**: should be more than the startTime and more than current time
  - **Impact**: no impact
- `nodeID`:
  - **Control**: partly controllable

  - **Checks**: `requireValidMinipool(nodeID)`
  - **Impact**: no impact
- `msg.sender`:
  - **Control**: -
  - **Checks**: `onlyValidMultisig(nodeID) : msg.sender == assignedMultisig`
  - **Impact**: only valid multisig can control when staking will be finished

Function call analysis

- `slash()`
  - **What is controllable?** minipoolIndex
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** can be reverted if
- `ggAVAX.depositFromStaking`
  - **What is controllable?** avaxLiquidStakerRewardAmt - partly controlled
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** revert if `stakingTotalAssets` value is less than `avaxLiquidStakerAmt`
- `vault.depositAVAX()`
  - **What is controllable?** avaxNodeOpRewardAmt - partly controlled
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** revert if `previewDeposit` returns 0.

Function: `recordStakingError()`

Intended behavior:

A staking error occurred while registering the node as a validator.

Can be called after `claimAndInitiateStaking` or `recordStakingStart`

Branches and code coverage:

Intended branches:

- After the call the new status is `Error`
  - ☑ Test coverage

Negative behavior:

- transaction should be rejected if current status · `Staking` or `Launched`
  - ☑ Negative test?

Preconditions:

- current status should be `Launched` or `Staking`

Inputs:

- `msg.value`:
  - **Control**: -
  - **Checks**: `msg.value` should be equal `avaxNodeOpAmt + avaxLiquidStakerAmt` - the withdrawn funds
  - **Impact**: amount of returned to staker funds. must not be less than the funds taken.
- `errorCode`:
  - **Control**: controlled
  - **Checks**: there isn't check here
  - **Impact**: no problems
- `nodeID`:
  - **Control**: controlled
  - **Checks**: check that minipool exists
  - **Impact**: the ID of the minipool that will be completed with an error without issuing a reward.

Function call analysis

- `ggAVAX.depositFromStaking()`
  - **What is controllable?** nothing is controlled
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if `stakingTotalAssets` is less than `avaxLiquidStakerAmt`
- `vault.depositAVAX()`
  - **What is controllable?** avaxNodeOpRewardAmt - partly controlled
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** revert if `previewDeposit` returns 0.

5.11 File: `MultisigManager`

Function: `registerMultisig()`

Intended behavior:

- Register a multisig. Defaults to disabled when first registered. The index where the `multisig` is to be added should be the previously increased `multisig.count`

Branches and code coverage:

Intended branches:

- "There will never be more than 10 total multisigs" There should be a check that 10 total multisigs can be registered (index · 9) and no more
  ☐ Test coverage
- Should register the `addr` as a new multisig, only if it doesn't exist already.
  ☑ Test coverage

Negative behavior:

- Shouldn't allow anyone else other than the guardian to call it
  ☑ Negative test?
- Shouldn't overwrite already existing multisig
  ☑ Negative test?
- Shouldn't also enable the multisig
  ☑ Negative test?

Preconditions:

- Assumes `getIndexOf` calculates the `index` properly and that two addresses cannot point to same `index`.
- Assumes there's a way to de-register a `Multisig`? Currently, there's none; there's only a way to disable them.

Inputs:

Function call analysis

Function: `enableMultisig()`

Intended behavior:

- Should enable a registered multisig.

Branches and code coverage:

Intended branches:

- The “enabled” of the `index` should be set to `true` .
  ☑ Test coverage

Negative behavior:

- Shouldn’t update the index of another multisig.
  ☑ Negative test?
- Shouldn’t be callable by anyone.
  □ Negative test? Not directly, but the `registerMultisig` which has the same modifier is tested when `msg.sender` ≠ `guardian`
- Shouldn’t enable a multisig that doesn’t exist.
  ☑ Negative test? Not tested, there is a check in the code that prevents this from happening.

Preconditions:

- Assumes that the `multisig` has been created beforehand.

Inputs:

Function call analysis

Function: `disableMultisig()`

Intended behavior:

- Should disable a registered multisig.

Branches and code coverage:

Intended branches:

- The “enabled” of the `index` should be set to `false`.
  ☑ Test coverage

Negative behavior:

- Shouldn’t be callable by any `msg.sender`
  ☑ Negative test?
- Shouldn’t update an non-existing `index`
  ☑ Negative test? Not tested, there is a check in the code that prevents this from happening.

Preconditions:

- Assumes that it can be called under any circumstances. What if it's called during a transaction where it needs to approve it?

Inputs:

Function call analysis

5.12 File: `Ocyticus`

Function: `addDefender()`, `removeDefender()`

Intended behavior:

- Allow guardian to `add` or `remove` defenders.

Branches and code coverage:

Lacks testing

Intended branches:

- Should update the `defenders` states properly.
  - ☐ Test coverage

Negative behavior:

- Should only be callable by `guardian`; covered by `onlyGuardian` modifier.
  - ☑ Negative test?

Preconditions:

- Assumes they are called by external accounts.

Inputs:

n/a

Function call analysis

Function: `pauseEverything()`

Intended behavior:

- Allows the `defender` to `pause` every contract that can be paused.

Branches and code coverage:

Intended branches:

- Pause `TokenGGAVAX`
  - ☑ Test coverage
- Pause `MinipoolManager`
  - ☑ Test coverage
- Pause `Staking` (MISSING!) - added as remediation
  - ☐ Test coverage

Negative behavior:

Preconditions:

- Assumes that the contracts can be `paused`.
- Assumes that when paused, no important functions from these contracts can be called! Double check this

Inputs:

n/a

Function call analysis

n/a

Function: `resumeEverything()`

Intended behavior:

Branches and code coverage:

Intended branches:

- Unpause `TokenGGAVAX`
  - ☑ Test coverage
- Unpause `MinipoolManager`
  - ☑ Test coverage
- Unpause `Staking` - added as remediation
  - ☐ Test coverage

Negative behavior:

Preconditions:

- Assumes that some other function will reenable all `multisigs`? That's not covered in this contract

Inputs:

n/a

Function call analysis

n/a

5.13 File: `Oracle`

Function: `setGGPPriceInAVAX()`, `getGGPPriceInAVAXFromOneInch`, `getGGPPriceInAVAX`

Intended behavior:

- Interface for off-chain aggregated data, used for pricing the tokens and calculating amounts. The `getGGPPriceInAVAXFromOneInch` should never be used on-chain.

Branches and code coverage:

Lacks testing.

Intended branches:

- The functions/ contracts that make use of the `GetGGPPriceInAvax` SHOULD have some slippage check in regards to the timestamp when the price has been updated: eg. If the price update happened more than 5 blocks away, revert the transaction.
  - ☐ Test coverage

Negative behavior:

- Shouldn't be callable by anyone. Only Multisig modifier put in place.
  - ☑ Negative test?

Preconditions:

- `getGGPPriceInAVAXFromOneInch` should only be called off-chain; it's not reliable enough to be called on chain directly.

- Assumes the Multisig update the `getGGPPRiceInAvax` quite often and that they are trustworthy.

Inputs:

There aren't input values here.

Function: `setOneInch()`

Intended behavior:

- Allows to guardian to set the address of the One Inch price aggregator contract

Branches and code coverage:

Intended branches:

- after the call `Oracle.OneInch` is updated to new address
  - □ Test coverage

Negative behavior:

- Revert if caller is not Guardian.
  - ☑ Negative test?

Preconditions:

- `msg.sender` is Guardian

Inputs:

- `addr`:
  - **Control**: controlled
  - **Checks**: There isn't check here.
  - **Impact**: The contract address which will be called inside view `getGGPPriceInAVAXFromOneInch` function

Function call analysis

There aren't external calls here.

Function: `setGGPPriceInAVAX()`

Intended behavior:

- The function is used by the Multisig to update the `on-chain` prices, with presumably the data retrieved off-chain from `OneInch`.

Branches and code coverage:

Intended branches:

- Should update the `GGPTimestamp`
  - □ Test coverage
- Should update the `GGPPriceInAvax`
  - □ Test coverage

Negative behavior:

- Revert if caller is not Multisig
  - □ Negative test?

Preconditions:

- `msg.sender` is Multisig

Inputs:

- `price`:
  - **Control**: controlled
  - **Checks**: price != 0
  - **Impact**: the price value is used during `calculateGGPSlashAmt` call
- `timestamp`:
  - **Control**: controlled
  - **Checks**: `timestamp` should be `>=` `lastTimestamp` or `timestamp` should be `<=` `block.timestamp`
  - **Impact**: n/a

Function call analysis

There aren't external calls here.

5.14 File: `ProtocolDAO`

Function: `initialize()`

Intended behavior:

- Initialize the contract
- Total `GGPCirculatingSupply = 18.000.000` but total `TokenGGP` supply `= 22.500.000`

Branches and code coverage:

- Not tested in the case of a re-deployment(or upgrade, as discussed with the team).

Intended branches:

- All `set` parameters should have a getter.
  - ☑ Test coverage; not test covered, but verified in the code.

Negative behavior:

- Setters that deal with rates should range from `0.0 - 1.0 ether`. This is not directly enforced; The same should be done for the rest of the `setter` functions from the contract. This was mitigated.
  - □ Negative test?

Preconditions:

- Assumes that it can only be called once, and that is through the `onlyGuardian`
- Assumes it will be called BEFORE any other functions that would use the initialized variables will be called. Maybe assure in important functions that
  - `getBool(keccak256("ProtocolDAO.initialized"))`is TRUE

Inputs:

Function call analysis

5.15 File: `RewardsPool`

Function: `initialize()`

Intended behavior:

- Re-initialize all `RewardsPool` variables for a new `RewardsPool`; This is upgradeable

Branches and code coverage:

- Not tested in the case of a re-deployment(or upgrade, as discussed with the team).

Intended branches:

- Should set the `RewardsPool` variables to their initial values.
  - □ Test coverage

Negative behavior:

Preconditions:

- Assumes it's the first the this type of contract has been deployed.

Inputs:

There aren't input values here.

Function call analysis

There aren't external calls here.

Function: `inflate()`

Intended behavior:

- Called to release more `GGP` from the total supply.
- says "mint" new tokens, but all of them are already minted.

Branches and code coverage:

Intended branches:

- Should update the rewardsCycle total amount.
  - □ Test coverage
- Should update the `inflationIntervalElapsedSeconds`
  - □ Test coverage
- Should increase circulating supply of tokens.
  - □ Test coverage

Preconditions:

- Assumes it won't be called that often

Inputs:

There aren't input values here.

Function call analysis

- `dao.setTotalGGPCirculatingSupply(newTotalSupply)`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: `startRewardsCycle()`

Intended behavior:

- Runs a `ggp` rewards cycle if possible.

Branches and code coverage:

- More extensive testing required.

Intended branches:

- `if dao allotment` exists · transfer `daoAllotment` to DAO · its balance should increase
  - ☑ Test coverage
- `if nop allotment` exists · transfer `nopAllotment` to NOP · its balance should increase
  - ☑ Test coverage
- `if multisig allotment`exists · transfer `multisigAllotment` to MULTISIG · its balance should increase
  - ☑ Test coverage
- Make sure allotments add up to 100%(the percentages)
  - ☐ Test coverage

Negative behavior:

- Shouldn't be callable whenever(`rewardscycle` should be scheduled)
  - ☑ Negative test?

Preconditions:

- Assumes that the `rewardsCycle` is startable.
- Also assumes that each allotment is `>0` . works even if that's not the case.

Inputs:

There aren't input values here.

Function call analysis

- `nopClaim.setRewardsCycleTotal(nopClaimContractAllotment)`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `vault.transferToken()`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value here
  - **What happens if it reverts, reenters, or does other unusual control flow?** revert if `tokenBalance` is less than `amount` value, or if `amount` is zero

Function: `distributeMultisigAllotment()`

Intended behavior:

- Should distribute the `ggp` to the multisigs.

Branches and code coverage:

Intended branches:

- Should only be called with legitimate `ggp` tokens.
  ☑ Test coverage

Negative behavior:

Lacks negative testing

- Should not distribute rewards to deactivated `multisigs`.
  □ Test coverage

Preconditions:

- Assumes there aren't that many multisigs
- Assumes that if multisigs gets deleted, they won't be eligible for `rewards`.

Inputs:

- allotment:
  - **Control**: value is calculated inside getClaimingContractDistribution(“ClaimMultisig”)
  - **Checks**: no checks at this function level, however, there may be some leftover tokens due to rounding errors; assure that these are sent somewhere after all allotments? (in startRewardsCycle)
  - **Impact**: determines the total amount of tokens that will be sent to multisigs.
- vault:
  - **Control**: address is taken from Vault(getContractAddress(“Vault”))
  - **Checks**: passed from previous function; same as ggp parameter.
  - **Impact**: n/a
- ggp:
  - **Control**: address is taken from TokenGGP(getContractAddress(“TokenGGP”))
  - **Checks**: full control; it’s passed from the previous function; ENSURE that it’s never called somewhere else or with a different GGP than here
  - **Impact**: n/a

Function call analysis

- mm.getCount();
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** out of gas inside the for loop if count value is too big
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- mm.getMultisig(i)
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** returns address and status of multisig, if enabled then this address will receive ggp tokens.
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- vault.withdrawToken(enabledMultisigs[i], ggp, tokensPerMultisig)
  - **What is controllable?** since this is an internal call, all input values are taken from storage.
  - **If return value controllable, how is it used and how can it go wrong?** there isn’t return value here.

- **What happens if it reverts, reenters, or does other unusual control flow?** will revert if `safeTransfer` call reverts and if `tokenBalances` less than `amount` value

5.16 File: `Staking`

Function: `GGP staking components`

Intended behavior:

- Limited negative testing

- `getGGPStake` = view current stake
- `increaseGGPStake` = increase `.ggpStaked`
- `decreaseGGPStake` = decrease `.ggpStaked`

Branches and code coverage:

Intended branches:

- Should retrieve / increase / decrease the `ggpStaked`.
  - □ Test coverage

Negative behavior:

- Shouldn't update an unregistered `stakerIndex`.
  - □ Negative test?

Preconditions:

- `increase` assumes that user has `deposited` the `ggp` and that the contract's balance has/ will increase
- `decrease` assumes that the user has `withdrawn` and that the `ggp` balance of the contract will `decrease` + `ggp` balance of user will `increase`.

Where are the functions used:

- `increaseGGPSTake`: Used in `_stakeGGP`
- `decreaseGGPStake`: Used in `slashGGP`, `withdrawGGP`

Function: `increaseAVAXStake()`

Intended behavior:

Increase the amount of AVAX for stakerAddr.

The function is called only from `MinipoolManager.createMinipool`.

Branches and code coverage:

Intended branches:

- After the function call the `getAVAXStake` for `stakerAddr` increased by the `amount` value
  ☑ Test coverage

Negative behavior:

- The function will revert if `stakerAddr` is not valid staker
  ☐ Negative test?
- The function will revert if msg.sender is not `MinipoolManager` contract
  ☐ Negative test?

Preconditions:

- `stakerAddr` called `stakeGGP` and was registered as a staker.

Inputs:

- `msg.sender`:
  - **Control**: -
  - **Checks**: `onlySpecificRegisteredContract("MinipoolManager", msg.sender)`
  - **Impact**: access to the function by untrusted addresses will allow manipulating the number of tokens staked.
- `amount`:
  - **Control**: `msg.value` is passed from the function `MinipoolManager.createMinipool` to ths function. limited control.
  - **Checks**: there are no checks.
  - **Impact**: this value reflects the number of stacked tokens. manipulating this value will allow an attacker to specify the number of tokens that have not actually been deposited.
- `stakerAddr`:
  - **Control**: `msg.sender` from `MinipoolManager.createMinipool`. not controlled.
  - **Checks**: the `requireValidStaker` function checks the address. If this address isn't staker, will revert.

- **Impact**: in case of full access it will allow any user to increase the number of tokens deposited.

Function call analysis

- `requireValidStaker()`
  - **What is controllable?** `stakerAddr`
  - **If return value controllable, how is it used and how can it go wrong?** return the `stakerIndex` corresponding to the `stakerAddr`. The Index must be unique, otherwise will be possible to lose funds.
  - **What happens if it reverts, reenters, or does other unusual control flow?** will be reverted if `stakerAddr` is not a valid staker.
- `addUint()`
  - **What is controllable? amount**
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** it can be reverted in overflow case,

Function: `decreaseAVAXStake()`

Intended behavior:

Decrease the amount of AVAX for stakerAddr.

The function is called from `MinipoolManager.withdrawMinipoolFunds` and `MinipoolManager._cancelMinipoolAndReturnFunds`.

Branches and code coverage:

Intended branches:

- After the function call the `getAVAXStake` for `stakerAddr` decreased by the `amount` value
  ☑ Test coverage

Negative behavior:

- The function will revert if `stakerAddr` is not valid staker
  □ Negative test?
- The function will revert if the `avaxStaked` for the `stakerAddr` is less than `amount`
  □ Negative test?
- The function will revert if msg.sender is not `MinipoolManager` contract
  □ Negative test?

Preconditions:

- `stakerAddr` have called `stakeGGP` and was registered as a staker.
- `stakerAddr` has non zero `avaxStaked` value

Inputs:

- `msg.sender`:
  - **Control**: -
  - **Checks**: `onlySpecificRegisteredContract("MinipoolManager", msg.sender)`
  - **Impact**: access to the function by untrusted addresses will allow manipulating the number of tokens staked
- `amount`:
  - **Control**: `getUint(keccak256(abi.encodePacked("minipool.item", minipoolIndex, ".avaxNodeOpAmt")))` value from `gogoStorage`, limited control.
  - **Checks**: this value cannot be more than current the `avaxStaked` value
  - **Impact**: this value reflects the number of stacked tokens. manipulating this value will allow an attacker to specify the number of tokens that have not actually been withdrawn.
- `stakerAddr`:
  - **Control**: owner of minipool. not controlled.
  - **Checks**: the `requireValidStaker` function checks the address. If this address isn't staker, will revert.
  - **Impact**: in case of full access it will allow any user to decrease the number of tokens deposited.

Function call analysis

- `subUint()`
  - **What is controllable?** amount
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if `avaxStaked` less than `amount`.
- `requireValidStaker()`
  - **What is controllable?** `stakerAddr`
  - **If return value controllable, how is it used and how can it go wrong?** return the `stakerIndex` corresponding to the `stakerAddr`. The Index must be unique, otherwise will be possible to lose funds.
  - **What happens if it reverts, reenters, or does other unusual control flow?**

will be reverted if stakerAddr is not a valid staker.

Function: increaseAVAXAssigned()

Intended behavior:

Increase the amount of AVAX a given staker is assigned by the protocol

The function is called only from MinipoolManager.createMinipool.

Branches and code coverage:

Intended branches:

- After the function call the getAVAXAssigned for stakerAddr increased by the amount value
  ☑ Test coverage

Negative behavior:

- The function will revert if stakerAddr is not valid staker
  □ Negative test?
- The function will revert if msg.sender is not MinipoolManager contract
  □ Negative test?

Preconditions:

- stakerAddr have called stakeGGP and was registered as a staker.

Inputs:

- amount:
  - **Control**: avaxAssignmentRequest is passed from the function MinipoolManager.createMinipool to ths function and should be equal the msg.sender value. limited control.
  - **Checks**: there are no checks
  - **Impact**: this value reflects the number of assigned tokens. Manipulating this value will allow an attacker to specify the number of tokens that have not actually been assigned.
- msg.sender:
  - **Control**: -
  - **Checks**: onlySpecificRegisteredContract(“MinipoolManager”, msg.sender)
  - **Impact**: access to the function by untrusted addresses will allow manipu-

lating the number of tokens assigned.

- stakerAddr:
  - **Control**: msg.sender from MinipoolManager.createMinipool. not controlled.
  - **Checks**: the requireValidStaker function checks the address. If this address isn't staker, will revert.
  - **Impact**: in case of full access it will allow any user to increase the number of tokens assign.

Function call analysis

- setUint(... ".avaxAssignedHighWater")
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- addUint(... ".avaxAssigned")
  - **What is controllable?** amount
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: decreaseAVAXAssigned()

Intended behavior:

Allows to decrease the amount of AVAX a given staker is assigned by the protocol

The function is called from MinipoolManager.recordStakingEnd and MinipoolManager.recordStakingError and MinipoolManager._cancelMinipoolAndReturnFunds.

Branches and code coverage:

Intended branches:

- After the function call the getAVAXAssigned for stakerAddr decreased by the amount value
  ☑ Test coverage

Negative behavior:

- The function will revert if stakerAddr is not valid staker
  ☐ Negative test?

- The function will revert if the `avaxAssigned` for the `stakerAddr` is less than `amount`
    - □ Negative test?
- The function will revert if msg.sender is not `MinipoolManager` contract
    - □ Negative test?

Preconditions:

- `stakerAddr` have called `stakeGGP` and was registered as a staker.
- `stakerAddr` has non zero `avaxAssigned` value

Inputs:

- `msg.sender`:
    - **Control**: -
    - **Checks**: `onlySpecificRegisteredContract("MinipoolManager", msg.sender)`
    - **Impact**: access to the function by untrusted addresses will allow manipulating the number of tokens assign.
- `amount`:
    - **Control**: `getUint(keccak256(abi.encodePacked("minipool.item", minipoolIndex, ".avaxLiquidStakerAmt")))` value from `gogoStorage`, limited control.
    - **Checks**: this value cannot be more than current the `avaxAssigned` value
    - **Impact**: this value reflects the number of staked tokens. Manipulating this value will allow an attacker to specify the number of tokens that have not actually been deposited.
- `stakerAddr`:
    - **Control**: owner of minipool. not controlled.
    - **Checks**: the `requireValidStaker` function checks the address. If this address isn't staker, will revert.
    - **Impact**: in case of full access it will allow any user to decreased the number of tokens assigned.

Function call analysis

- `subUint()`
    - **What is controllable?** amount
    - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
    - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if `avaxAssigned` less than `amount`.

- `requireValidStaker()`
  - **What is controllable?** `stakerAddr`
  - **If return value controllable, how is it used and how can it go wrong?** return the `stakerIndex` corresponding to the `stakerAddr`. The Index must be unique, otherwise will be possible to lost funds.
  - **What happens if it reverts, reenters, or does other unusual control flow?** will be reverted if `stakerAddr` is not a valid staker.

Function: `setRewardsStartTime`

Intended behavior:

- Rewards start time refers to the `timestamp` when the staker registered for `GGPrewards`

Branches and code coverage:

Intended branches:

- Ensure that `time` is in the future?
  - □ Test coverage
- Should allow `setting` the `rewardStartTime`
  - □ Test coverage

Negative behavior:

- Also, assuming that `onlyRegisteredNetworkContract` calls it. Also I think they whitelist their own `Staking` contract(basically `address(this)`
  - □ Negative test?

Preconditions:

- Assumes that it's called from `onlySpecificRegisteredContract("ClaimNodeOp", msg.sender)`

Inputs:

- `time`:
  - **Control**: full control
  - **Checks**: there's no check on whether the `time` is in the future or not
  - **Impact**: the value is used during reward distribution, if zero, the staker will not receive reward

Function call analysis

There aren't external calls here.

Where are the functions used:

- `setRewardsStartTime`: used in `MinipoolManager` and `ClaimNodeOp`

Function: `GGP Rewards()`

Intended behavior:

- Should `get`, `increase`, `decrease` the `GGPRewards` assigned to a staker.

Branches and code coverage:

Intended branches:

- These should update whenever the staker claims / is issued rewards.
  - □ Test coverage
- Should retrieve/ increase/ decrease the amount of `GGPrewards` a staker has **earned** and **not claimed yet.**
  - □ Test coverage

Negative behavior:

- Should revert if anyone other than the `ClaimNodeOp` contract calls them.
  - □ Negative test?

Preconditions:

- Assumes that the calling contract holds the correct accounting for how the `ggp` rewards are issued and maintained.

Function call analysis

There aren't external calls here.

Where are the functions used:

- `increaseGGPRewards`: used in `ClaimNodeOP`
- `decreaseGGPrewards`: used in `ClaimNodeOP`

Function: `increaseMinipoolCount()`

Intended behavior:

The function is called from `MinipoolManager.createMinipool`

Increase the number of minipools the given staker has

Branches and code coverage:

Intended branches:

- After the function call the `.minipoolCount` increased by 1
  - ☑ Test coverage

Negative behavior:

- The function will revert if the `.minipoolCount` is zero
  - □ Negative test?
- The function will revert if msg.sender is not `MinipoolManager` contract
  - □ Negative test?

Preconditions:

- `stakerAddr` have called `stakeGGP` and was registered as a staker.

Inputs:

- `stakerAddr`:
  - **Control**: owner of minipool. not controlled.
  - **Checks**: the `requireValidStaker` function checks the address. If this address isn't staker, will revert.
  - **Impact**: in case of full access it will allow any user to increase the amount of minipools
- `msg.sender`:
  - **Control**: -
  - **Checks**: `onlySpecificRegisteredContract("MinipoolManager", msg.sender)`
  - **Impact**: access to the function by untrusted addresses will allow manipulating the number of the given staker minipools. The `setRewardsStartTime` value depends of the amount of minipools, if minipoolCount = 0 `RewardsStartTime` will be reset. If `RewardsStartTime == 0` then `RewardsStartTime` will be set during minipool creation. And if `RewardsStartTime == 0` then owner of minipool doesn't get the GGP rewards

Function call analysis

- `addUint()`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `requireValidStaker()`
  - **What is controllable?** `stakerAddr`
  - **If return value controllable, how is it used and how can it go wrong?** return the `stakerIndex` corresponding to the `stakerAddr`. The Index must be unique, otherwise will be possible to lost funds.
  - **What happens if it reverts, reenters, or does other unusual control flow?** will be reverted if `stakerAddr` is not a valid staker.

Function: `decreaseMinipoolCount()`

Intended behavior:

Decrease the number of minipools the given staker has

The function is called from `MinipoolManager.recordStakingEnd` and `MinipoolManager._cancelMinipoolAndReturnFunds`

Branches and code coverage:

Intended branches:

- After the function call the `.minipoolCount` decreased by 1
  ☑ Test coverage

Negative behavior:

- The function will revert if `stakerAddr` is not valid staker
  □ Negative test?
- The function will revert if the `.minipoolCount` is zero
  □ Negative test?
- The function will revert if msg.sender is not `MinipoolManager` contract
  □ Negative test?

Preconditions:

- `stakerAddr` have called `stakeGGP` and was registered as a staker.

- The `.minipoolCount` is not zero

Inputs:

- `stakerAddr`:
  - **Control**: owner of minipool. not controlled.
  - **Checks**: the `requireValidStaker` function checks the address. If this address isn't staker, will revert.
  - **Impact**: in case of full access it will allow any user to decrease the amount of minipools
- `msg.sender`:
  - **Control**: -
  - **Checks**: `onlySpecificRegisteredContract("MinipoolManager", msg.sender)`
  - **Impact**: access to the function by untrusted addresses will allow manipulating the number of the given staker minipools. The `setRewardsStartTime` value depends of the amount of minipools, if minipoolCount = 0 `RewardsStartTime` will be reset. if `RewardsStartTime == 0` then `RewardsStartTime` will be set during minipool creation. And if `RewardsStartTime == 0` then owner of minipoll doesn't get the GGP rewards

Function call analysis

- `subUint()`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if `.minipoolCount` is 0.
- `requireValidStaker()`
  - **What is controllable?** `stakerAddr`
  - **If return value controllable, how is it used and how can it go wrong?** return the `stakerIndex` corresponding to the `stakerAddr`. The Index must be unique, otherwise will be possible to lost funds.
  - **What happens if it reverts, reenters, or does other unusual control flow?** will be reverted if `stakerAddr` is not a valid staker.

Function: `setRewardsStartTime()`

Intended behavior:

Set the timestamp when the staker registered for GGP rewards.

The `setRewardsStartTime` value depends of the amount of minipools, if minipoolCount = 0 `RewardsStartTime` will be reset inside the `calculateAndDistributeRewards()` function, which called from `processGGPRewards` if `isEligible` true (is not true if `RewardsStartTime == 0`). if `RewardsStartTime == 0` then `RewardsStartTime` will be set during minipool creation.

Branches and code coverage:

Intended branches:

- After the function call the `.rewardsStartTime` is equal to `time`
  ☑ Test coverage

Negative behavior:

- The function will revert if `stakerAddr` is not valid staker
  □ Negative test?
- The function will revert if `msg.sender` is not `RegisteredNetworkContract`
  □ Negative test?

Preconditions:

- `stakerAddr` have called `stakeGGP` and was registered as a staker.

Inputs:

- `time`:
  - **Control**: partly controlled: during minipool creation `block.timestamp` is passed
  - **Checks**: there aren't any checks
  - **Impact**: if set to 0 than owner of minipool cannot get the GGP rewards and if non zero will be able to get (`isEligible()`: if `(block.timestamp - rewardsStartTime) · dao.getRewardsEligibilityMinSeconds()`)
- `stakerAddr`:
  - **Control**: owner of minipool. not controlled.
  - **Checks**: the `requireValidStaker` function checks the address. If this address isn't staker, will revert.
  - **Impact**: in case of full access it will allow any user to set the `RewardsStartTime` and bypass the `isEligible` check.
- `msg.sender`:
  - **Control**: -
  - **Checks**: `onlyRegisteredNetworkContract`
  - **Impact**: access to the function by untrusted addresses will allow manipu-

lating the `RewardsStartTime` value. If `RewardsStartTime` != 0 then owner of minipool will be able to get the GGP rewards

Function call analysis

- `setUint()`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems
- `requireValidStaker()`
  - **What is controllable?** `stakerAddr`
  - **If return value controllable, how is it used and how can it go wrong?** return the `stakerIndex` corresponding to the `stakerAddr`. The Index must be unique, otherwise will be possible to lost funds.
  - **What happens if it reverts, reenters, or does other unusual control flow?** will be reverted if `stakerAddr` is not a valid staker.

Function: `increaseGGPRewards()`

Intended behavior:

Increase the amount of GGP rewards the staker has earned and not claimed

The function is called from `ClaimNodeOp.calculateAndDistributeRewards`

Branches and code coverage:

Intended branches:

- After the call the `.ggpRewards` amount will be increased by `amount`
  ☑ Test coverage

Negative behavior:

- The function will revert if `stakerAddr` is not valid staker
  ☐ Negative test?
- The function will revert if `msg.sender` is not `ClaimNodeOp` contract
  ☐ Negative test?

Preconditions:

- `stakerAddr` have called `stakeGGP` and was registered as a staker.

Inputs:

- `amount`:
  - **Control**:
  - **Checks**: there aren't checks
  - **Impact**: The value determines how much the user will be able to receive rewards. In case of full access to the function, users will be able to steal all funds from the vault.
- `stakerAddr`:
  - **Control**: owner of minipool. not controlled.
  - **Checks**: the `requireValidStaker` function checks the address. If this address isn't staker, will revert.
  - **Impact**: in case of full access it will allow any user to increase the `.ggpRewards`
- `msg.sender`:
  - **Control**: -
  - **Checks**: `onlySpecificRegisteredContract("ClaimNodeOp", msg.sender)`
  - **Impact**: access to the function by untrusted addresses will allow manipulating the `.ggpRewards` value.

Function call analysis

- `requireValidStaker()`
  - **What is controllable?** `stakerAddr`
  - **If return value controllable, how is it used and how can it go wrong?** return the `stakerIndex` corresponding to the `stakerAddr`. The Index must be unique, otherwise will be possible to lost funds.
  - **What happens if it reverts, reenters, or does other unusual control flow?** will be reverted if `stakerAddr` is not a valid staker.
- `addUint()`
  - **What is controllable?** amount
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: `decreaseGGPRewards()`

Intended behavior:

Decrease the amount of GGP rewards the staker has earned and not claimed.

The function is called from `ClaimNodeOp.claimAndRestake`

Branches and code coverage:

Intended branches:

- After the call the `.ggpRewards` is decreased by the `amount` value.
  ☑ Test coverage

Negative behavior:

- The function will revert if `stakerAddr` is not valid staker
  □ Negative test?
- The function will revert if the `.ggpRewards` is less than `amount`
  □ Negative test?
- The function will revert if msg.sender is not `ClaimNodeOp` contract
  □ Negative test?

Preconditions:

- `stakerAddr` have called `stakeGGP` and was registered as a staker.
- The `.ggpRewards` is set by the `ClaimNodeOp.calculateAndDistributeRewards` function call

Inputs:

- `amount`:
  - **Control**: not controlled
  - **Checks**: there aren't checks
  - **Impact**: in case of an untrusted caller, the `.ggpRewards` can be reset and owner of pool will not be able to get reward
- `stakerAddr`:
  - **Control**: owner of minipool. not controlled.
  - **Checks**: the `requireValidStaker` function checks the address. If this address isn't staker, will revert.
  - **Impact**: in case of full access it will allow any user to decrease the `.ggpRewards`
- `msg.sender`:
  - **Control**: -
  - **Checks**: `onlySpecificRegisteredContract("ClaimNodeOp", msg.sender)`
  - **Impact**: access to the function by untrusted addresses will allow manipulating the `.ggpRewards` value.

Function call analysis

- requireValidStaker()
  - **What is controllable?** stakerAddr
  - **If return value controllable, how is it used and how can it go wrong?** return the stakerIndex corresponding to the stakerAddr. The Index must be unique, otherwise will be possible to lost funds.
  - **What happens if it reverts, reenters, or does other unusual control flow?** will be reverted if stakerAddr is not a valid staker.
- subUint()
  - **What is controllable?** amount
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if .ggpRewards is less than amount.

Function: setLastRewardsCycleCompleted()

Intended behavior:

Set the most recent reward cycle number that the staker has been paid out for.

The function is called from ClaimNodeOp.calculateAndDistributeRewards

Branches and code coverage:

Intended branches:

- After the call the .lastRewardsCycleCompleted is equal to the cycleNumber value
  - □ Test coverage

Negative behavior:

- The function will revert if stakerAddr is not valid staker
  - □ Negative test?
- The function will revert if msg.sender is not ClaimNodeOp contract
  - □ Negative test?

Preconditions:

- stakerAddr have called stakeGGP and was registered as a staker.

Inputs:

- cycleNumber:

    - **Control**: the value from the `rewardsPool.getRewardsCycleCount()` function call
    - **Checks**: there aren't checks
    - **Impact**: prevents re-receiving the reward in the same cycle.
- `stakerAddr`:
    - **Control**: owner of minipool. not controlled.
    - **Checks**: the `requireValidStaker` function checks the address. If this address isn't staker, will revert.
    - **Impact**: in case of full access it will allow any user to decrease the `.ggpRewards`
- `msg.sender`:
    - **Control**: -
    - **Checks**: `onlySpecificRegisteredContract("ClaimNodeOp", msg.sender)`
    - **Impact**: access to the function by untrusted addresses will allow manipulating the `.lastRewardsCycleCompleted` value.

Function call analysis

- `requireValidStaker()`
    - **What is controllable?** `stakerAddr`
    - **If return value controllable, how is it used and how can it go wrong?** return the `stakerIndex` corresponding to the `stakerAddr`. The Index must be unique, otherwise will be possible to lost funds.
    - **What happens if it reverts, reenters, or does other unusual control flow?** will be reverted if `stakerAddr` is not a valid staker.
- `setUint()`
    - **What is controllable?** `cycleNumber`
    - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
    - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: `getMinimumGGPStake()`

Intended behavior:

- Retrieve staker's minimum `GGP` stake, based on current `GGP` price.

Branches and code coverage:

Intended branches:

- Ensure that `stakerAddr` is valid; currently not checked
  - ☐ Test coverage

Preconditions:

- Assumes that the `stakerAddr` has some `avaxAssigned` to them.

Function call analysis

- `(uint256 ggpPriceInAvax, ) = oracle.getGGPPriceInAVAX();`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** Part of the return value is ignored(that refers to the block.timestamp when the price has been updated) Maybe it's a good idea to also return that? The price could be really outdated; Add something like a max amount of blocks that go without update?
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if `price` is zero

Function: `getCollateralizationRatio()`

Intended behavior:

- Return collateralization ratio based on current `GGP` price.

Branches and code coverage:

Intended branches:

- Ensure that `stakerAddr` is valid; currently not checked
  - ☐ Test coverage

Preconditions:

- Assumes that the `stakerAddr` has some `avaxAssigned` to them.

Function call analysis

- `(uint256 ggpPriceInAvax, ) = oracle.getGGPPriceInAVAX();`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** Part of the return value is ignored(that refers to the block.timestamp when the price has been updated) Maybe it's a good idea to also return that? The price could be really outdated; Add something like a max amount of blocks

    that go without update?
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if `price` is zero

Where is the function used:

- `MinipoolManager`:
- `Staking`:

Function: `getEffectiveRewardsRatio()`

Intended behavior:

- return effective collateralization ratio used to pay rewards based on `GGP` price and `AVAX` high water.

Branches and code coverage:

Intended branches:

- Ensure that `stakerAddr` is valid; currently not checked
  - ☐ Test coverage

Preconditions:

- Assumes that the `stakerAddr` has some `GGPstaked` already.

Function call analysis

- `(uint256 ggpPriceInAvax, ) = oracle.getGGPPriceInAVAX();`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** Part of the return value is ignored(that refers to the block.timestamp when the price has been updated) Maybe it's a good idea to also return that? The price could be really outdated; Add something like a max amount of blocks that go without update?
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if `price` is zero
- `dao.getMaxCollateralizationRatio();`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** return the max collateralization ratio of GGP to Assigned AVAX eligible for rewards. This value is used for `EffectiveGGPStaked` value calculations for

reward distribution process

- **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: `getEffectiveGGPStaked()`

Intended behavior:

- Get amount of `ggp` that will count towards the `rewards` cycle.

Branches and code coverage:

Intended branches:

- Ensure that `stakerAddr` is valid; currently not checked
  - □ Test coverage

Preconditions:

- the `price` value is set inside Oracle contract

Function call analysis

- `(uint256 ggpPriceInAvax, ) = oracle.getGGPPriceInAVAX();`
  - **What is controllable?** -
  - **If return value controllable, how is it used and how can it go wrong?** Part of the return value is ignored(that refers to the block.timestamp when the price has been updated) Maybe it's a good idea to also return that? The price could be really outdated; Add something like a max amount of blocks that go without update?
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if `price` is zero

Where is the function used:

- `ClaimNodeOp`:

Function: `stakeGGP()` and `_stakeGGP`

Intended behavior:

- Should allow any user to `stake` GGP into the contract.

Branches and code coverage:

Intended branches:

- Should revert if `msg.sender` transferred less than `amount` tokens.
  ☑ Test coverage
- The `ggp` balance of the `msg.sender` should deplete by `amount`, whilst the `contract` should have enough to `deposit` into the `vault(like a middleman)`
  ☑ Test coverage
- The `GGPStake` of the user should be increased by the `staked` amount.
  ☑ Test coverage

Negative behavior:

- Limited negative testing
- Shouldn't allow transferring arbitrary tokens
  ☑ Negative test?

Preconditions:

- Assumes `msg.sender` is registered as a staker in the contract; however, if that's not the case, it creates an index for a new staker:
- Assumes that `msg.sender` has previously approved the amount that is to be transferred by `stakeGGP`.

Inputs:

- `amount`:
  - **Control**: full control
  - **Checks**: there are no 0 checks, however, they do `safeTransferFrom` user with the `amount`
  - **Impact**: n/a

Function call analysis

- `ggp.safeTransferFrom()`
  - **What is controllable?** amount
  - **If return value controllable, how is it used and how can it go wrong?** there ins't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if msg.sender doesn't have enough ggp tokens.
- `_stakeGGP()`
  - **What is controllable?** amount

  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: `restakeGGP()`

Intended behavior:

- allow restaking for claimedGGP rewards

Branches and code coverage:

Intended branches:

- after the call the `.ggpStaked` value of `stakerAddr` will be increased by `amount` value
  - ☑ Test coverage

Negative behavior:

Limited negative testing

- if msg.sender doesn't have enough ggp tokens, transaction will be reverted
  - □ Negative test?
- if msg.sender is not trusted ClaimNodeOp contract, transaction will be reverted
  - □ Negative test?

Preconditions:

- Assumes `msg.sender` is `ClaimNodeOp`
- msg.sender must have at least the amount value of ggp tokens

Inputs:

- `amount`:
  - **Control**: limited control
  - **Checks**: safeTransferFrom will revert if msg.sender balance less than `amount`
  - **Impact**: -
- `stakerAddr`:
  - **Control**: full control
  - **Checks**: there aren't any checks
  - **Impact**: -

- `msg.sender`:
  - **Control**: -
  - **Checks**: onlySpecificRegisteredContract("ClaimNodeOp", msg.sender)
  - **Impact**: the function allows caller to increase `.ggpStaked` value for any user. but caller should send this value of ggp tokens to contract

Function call analysis

- `ggp.safeTransferFrom()`
  - **What is controllable?** amount
  - **If return value controllable, how is it used and how can it go wrong?** there ins't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** will revert if msg.sender doesn't have enough ggp tokens.
- `_stakeGGP()`
  - **What is controllable?** amount
  - **If return value controllable, how is it used and how can it go wrong?** there isn't return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** no problems

Function: `withdrawGGP()`

Intended behavior:

- Allows withdrawing `GGP` tokens.

Branches and code coverage:

Intended branches:

- Should ensure that the `.ggpStaked` decreases.
  - □ Test coverage

Negative behavior:

- Should never lock-up `ggp` ; this could happen in a scenario where the `msg.sender` is never over 150% collateralization
  - □ Negative test?

Preconditions:

- Assumes that the user is over `150%` in collateralization ratio.

- Assure that `maxCollateralizationRatio` is synced up! Maybe check the last block and compare it with the last block from `getCollateralizationRatio` as well?! a de-sync could lead to lower threshold of withdrawals. Any huge fluctuations would greatly affect this.

Inputs:

- `amount`:
  - **Control**: full controll
  - **Checks**: checks that `amount > getGGPStake` and check that `getCollateralizationRatio(msg.sender)` at least 150 after withdraw
  - **Impact**: could lead to loss of funds if not depleted properly.

Function: `slashGGP()`

Intended behavior:

- Should be used by the `MinipoolManager` in case that a `minipool` has ended; this happen

Branches and code coverage:

Intended branches:

- Decrease the `ggpStake` of the `staker` (assuming `staker` has some left)
  ☑ Test coverage
- `StakerAddr` must be registered.
  □ Test coverage

Negative behavior:

- Only allow `minipoolmanager` to call this.
  □ Negative test?

Preconditions:

- Assumes that `decreaseGGPSTake` can be called on the `stakerAddr`(this implies that `stakerAddr` has been registered beforehand)

Inputs:

- `ggpAmt`:
  - **Control**: full control
  - **Checks**: assumes that `decreaseGGPStake` properly decreases the amount

  that the `stakerAddr` has

- **Impact**: n/a

6 Audit Results

At the time of our audit, the code was not deployed to mainnet Avalanche.

During our assessment on the scoped GoGoPool contracts, we discovered seven findings. Of the seven findings, four were of high severity, one was of medium severity, one was of low severity and the remaining finding was informational. Multisig Labs acknowledged all findings and implemented fixes.

6.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.