Arcade.xyz
Findings & Analysis Report

2023-10-25

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 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 outlined in this document, C4 conducted an analysis of the Arcade.xyz smart contract system written in Solidity. The audit took place between July 21 — July 28 2023.

Wardens

64 Wardens contributed reports to the Arcade.xyz:

1. DadeKuma
2. bart1e
3. ladboy233
4. 0x3b
5. osmanozdemir1

6. T1MOH
7. bin2chen
8. caventa
9. 0xWaitress
10. Juntao
11. Topmark
12. Jiamin
13. crunch
14. Anirruth
15. K42
16. ktg
17. 0xnev
18. MohammedRizwan
19. Matin
20. giovannidisiena
21. rvierdiiev
22. Udsen
23. Sathish9098
24. UniversalCrypto (amaechieth and tettehnetworks)
25. vangrim
26. zaevlad
27. auditsea
28. circlelooper
29. lanrebayode77
30. oakcobalt
31. kutugu
32. peanuts
33. 3agle
34. foxb868

35. c3phas
36. excalibor
37. dharma09
38. kaveyjoe
39. Viktor_Cortess
40. 0xmuxyz
41. 0xbranded
42. sces60107
43. BenRai
44. 0xastronatey
45. LaScaloneta (nicobevi, 0x4non and juancito)
46. BugBusters (nirlin and 0xepley)
47. QiuhaoLi
48. 0xComfyCat
49. zhaojie
50. squeaky_cactus
51. SM3_SS
52. Aymen0909
53. Raihan
54. jeffy
55. ak1
56. 0xDING99YA
57. immeas
58. matrix_0wl
59. koxuan
60. ABA

This audit was judged by 0xean.

Final report assembled by thebrittfactor.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 20 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 10 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 Arcade.xyz repository, and is composed of 12 smart contracts written in the Solidity programming language and includes 979 lines of Solidity code.

In addition to the known issues identified by the project team, a Code4rena bot race was conducted at the start of the audit. The winning bot, **c4lanky** from warden favelanky, generated the Automated Findings report and all findings therein were classified as out of scope.

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.

Medium Risk Findings (8)

[M-01] `_syncVotingPower` can revert in underflow in `NFTBoostVault.sol` and `ARCDVestingVault.sol`

Submitted by ladboy233, *also found by* Topmark

Lines of code:

https://github.com/code-423n4/2023-07-arcade/blob/f8ac4e7c4fdea559b73d9dd5606f618d4e6c73cd/contracts/NFTBoostVault.sol#L593

https://github.com/code-423n4/2023-07-arcade/blob/f8ac4e7c4fdea559b73d9dd5606f618d4e6c73cd/contracts/ARCDVestingVault.sol#L352

Proof of Concept

Note this function:

```
    function _syncVotingPower(address who, NFTBoostVaultStorage.R
        History.HistoricalBalances memory votingPower = _votingP
        uint256 delegateeVotes = votingPower.loadTop(registratio

        uint256 newVotingPower = _currentVotingPower(registratio
        // get the change in voting power. Negative if the votin

        int256 change = int256(newVotingPower) - int256(uint256(

        // do nothing if there is no change
        if (change == 0) return;
        if (change > 0) {
            votingPower.push(registration.delegatee, delegateeVo
        } else {
            // if the change is negative, we multiply by -1 to a
            votingPower.push(registration.delegatee, delegateeVo
        }
```

```
            registration.latestVotingPower = uint128(newVotingPower)

            emit VoteChange(who, registration.delegatee, change);
        }
```

We need to pay special attention to this line of code:

```
votingPower.push(registration.delegatee, delegateeVotes - uint
```

It is possible that `delegateeVotes -` changes can revert in underflow. This only happens when we try to reduce the voting power.

In `_withdrawNFT`, if an NFT with a high multipler is removed in `NFTBoostVault.sol` it can cause the [voting power to decrease](#).

```
// update the delegatee's voting power based on multiplier remov
_syncVotingPower(msg.sender, registration);
```

Or, in `ARCDVestingVault.sol` when an admin tries to [revoke a user's grant](#).

```
// update the grant's withdrawn amount
if (amount == withdrawable) {
        grant.withdrawn += uint128(withdrawable);
} else {
        grant.withdrawn += uint128(amount);
        withdrawable = amount;
}

// update the user's voting power
_syncVotingPower(msg.sender, grant);
```

Recommended Mitigation Steps

Before calling `delegateeVotes - uint256(change * -1)`, make sure `delegateeVotes > uint256(change * -1)`, otherwise just set to 0.

PowVT (Arcade.xyz) confirmed and commented via duplicate issue #540:

> Will look into underflow scenarios here where there is a multiplier boost.

[M-02] Proposal vote power can be easily manipulated

Submitted by DadeKuma

Lines of code

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/ArcadeGSCCoreVoting.sol#L32

Impact

`ArcadeGSCCoreVoting` can be a target of vote manipulation. An attacker might be able to take a huge loan (even uncollateralized) for a single block before creating a proposal.

When voting, only this single block is checked when calculating the `votingPower`. This may lead to an attacker being able to execute arbitrary proposals with minimal risks involved.

Proof of Concept

When a proposal is created, the timestamp registered is the block before the creation. This mitigates flash loan attacks, but it's still possible to manipulate the vote with a normal loan:

```
    proposals[proposalCount] = Proposal(
        proposalHash,
        // Note we use blocknumber - 1 here as a flash loan mitigati
        uint128(block.number - 1), //@audit created
        uint128(block.number + lockDuration),
        uint128(block.number + lockDuration + extraVoteTime),
        uint128(quorum),
        proposals[proposalCount].votingPower,
        uint128(lastCall)
    );
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/external/council/CoreVoting.sol#L172-L181

During a `vote`, the `msg.sender` voting power is queried and it will use the previous `created` field:

```
    for (uint256 i = 0; i < votingVaults.length; i++) {
        // ensure there are no voting vault duplicates
        for (uint256 j = i + 1; j < votingVaults.length; j++) {
            require(votingVaults[i] != votingVaults[j], "duplicate v
        }
        require(approvedVaults[votingVaults[i]], "unverified vault")
        votingPower += uint128(
            IVotingVault(votingVaults[i]).queryVotePower(
                msg.sender,
                proposals[proposalId].created,
                extraVaultData[i]
            )
        );
    }
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/external/council/CoreVoting.sol#L234-L238

If the attacker took a huge loan (even uncollateralized) for that single block, they would be able to manipulate the vote with minimal slippage, as it's only one block.

If this happens, they would be able to execute any arbitrary code, if `queryVotePower` depends on the amount of tokens held by the attacker.

Note about severity

By reading the comments in `ArcadeGSCCoreVoting` it seems that the voting vault used will be the `ArcadeGSCVault`:

```
 * The Arcade GSC Core Voting contract allows members of the GSC
 * in an instance of governance separate from general governance
```

In this case, this issue can't occur, as the `votingPower` does not depend on the amount held by the attacker:

```
// If the address queried is the owner they get a huge number of
// This allows the primary governance timelock to take any actic
// can make or block any action the GSC can make. But takes as n
// a protocol upgrade.
if (who == owner) {
    return 100000;
}
// If the who has been in the GSC longer than idleDuration
// return 1 and otherwise return 0.
if (
    members[who].joined > 0 &&
    (members[who].joined + idleDuration) <= block.timestamp
) {
    return 1;
} else {
    return 0;
}
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/external/council/vaults/GSCVault.sol#L145-L167

However, it's worth noting that this situation might change in the future, as `ArcadeGSCCoreVoting` approved vaults can be multiple, and they are not immutable:

```
/// @notice Updates the status of a voting vault.
/// @param vault Address of the voting vault.
/// @param isValid True to be valid, false otherwise.
function changeVaultStatus(address vault, bool isValid) external
    approvedVaults[vault] = isValid;
}
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/external/council/CoreVoting.sol#L333-L338

As there are other vaults that use tokens as voting power (e.g. LockingVault), there is a real possibility that this might occur in the future, so I'm flagging it as high severity.

Recommended Mitigation Steps

Consider using a `TWAP` to check the voting power of a proposal, instead of checking only the block before the proposal was created.

Assessed type

Timing

PowVT (Arcade.xyz) confirmed, disagreed with severity and commented:

> Should make adding approved vaults to the `ArcadeGSC` contract revert. Will only use the `GSCVault` here and can eliminate any potential for the scenario described above. I would mark it as medium, as this is not the intended usage of the `GSCCoreVoting` contract and there are high custom quorums for sensitive actions like approving a new vault.

0xean (judge) decreased severity to Medium and commented:

> Will only use the GSCVault here and can eliminate any potential for the scenario described above.

> Based on this being a vote among the steering council I think M is the correct severity here.

- The Arcade GSC Core Voting contract allows members of the GSC vault to vote on and execute proposals.
- In an instance of governance separate from general governance votes.

T1MOH (warden) commented:

> I would argue this issue is medium. The report says an attacker can take a loan in certain block to take most of the voting power. I argue that if attackers can lend this amount, they already own this amount, and `TWAP` won't save protocol; it just requires the funds to be frozen in this governance token for several blocks to become available in voting. But it doesn't mitigate the described attack.

> The lending market, especially in blockchain, is overcollateralized; you can't lend more than deposited. I don't see the difference between the described attack vector and the 51% attack which is obviously out of scope. This issue is more suitable in an analysis report.

DadeKuma (warden) commented:

> The lending market, especially in blockchain, is overcollateralized; you can't lend more than deposited.

> This is false, undercollateralized lending **exists**.

> `TWAP` itself doesn't make the protocol immune to multi-block attacks, but it definitely helps mitigate the issue by making them exponentially difficult to perform in comparison to a single-block check.

> The solution proposed by the sponsor mitigates this issue if multiple vaults are not required:

> Should make adding approved vaults to the `ArcadeGSC` contract revert. Will only use the `GSCVault` here and can eliminate any potential for the scenario described above. I would mark it as medium, as this is not the intended usage of the `GSCCoreVoting` contract and there are high custom quorums for sensitive actions like approving a new vault.

T1MOH (warden) commented:

1. From an economic view, there is no undercollateralized lending. What you refer to is more similar to real life lending, where you provide collateral; not immediately, but undertake to repay it in the future. In other words, you can't lend more than you will have to at the end of repayment; no one will give you money for free. Otherwise it is just crowdfunding for a 51% attack.
2. Compound-like governance framework and `ERC20Votes` is at least battle tested and they use similar approach in counting votes. Can it be medium vulnerability?

> As I understand (but can be wrong), the proposed solution by the sponsor blocks the usage of vaults with voting tokens. But I argue that these vaults are not vulnerable.

PowVT (Arcade.xyz) commented:

> To add another note here, since there is no way to override the `changeVaultStatus` in CoreVoting because it is not virtual, the other easier fix would be to set the `customQuorum` for interacting with this function unattainably high. This is more than likely the direction we will pursue, since no contract changes are needed.

[M-03] If someone becomes a GSC member, they may become un-kickable forever

Submitted by bart1e

Lines of code

https://github.com/code-423n4/2023-07-arcade/blob/f8ac4e7c4fdea559b73d9dd5606f618d4e6c73cd/contracts/BaseVotingVault.sol#L96-L102
https://github.com/code-423n4/2023-07-arcade/blob/f8ac4e7c4fdea559b73d9dd5606f618d4e6c73cd/contracts/external/council/vaults/GSCVault.sol#L123
https://github.com/code-423n4/2023-07-arcade/blob/f8ac4e7c4fdea559b73d9dd5606f618d4e6c73cd/contracts/external/council/libraries/History.sol#L198-L199
https://github.com/code-423n4/2023-07-arcade/blob/f8ac4e7c4fdea559b73d9dd5606f618d4e6c73cd/contracts/ArcadeGSCVault.sol#L25

Vulnerability details

Note: Some of the contracts mentioned are out of scope, but the vulnerability exists in the `BaseVotingVault`, which is in scope, so I argue that the finding is in scope.

In the Arcade ecosystem, there is a GSC group which has some extra privileges, like spending some token amount from the treasury or creating new proposals in the core voting contract.

In order to become a member of this group, a user has to have a high enough voting power (combined from several voting vaults) and call `proveMembership`. When user's voting power drops beneath a certain threshold, they may be kicked out of the GSC.

`proveMembership` contains the following code:

```
    for (uint256 i = 0; i < votingVaults.length; i++) {
                // Call the vault to check last block's voting power
                // Last block to ensure there's no flash loan or oth
                // intra contract interaction
                uint256 votes =
                    IVotingVault(votingVaults[i]).queryVotePower(
                        msg.sender,
                        block.number - 1,
                        extraData[i]
                    );
                // Add up the votes
                totalVotes += votes;
            }
```

So, it basically iterates over all voting vaults that a user specifies, sums up their voting power, and if it's enough, it grants that user a place in GSC.

In order to kick a user out from the GSC, the `kick` function may be used and it will iterate over all vaults that were supplied by a user when they call `proveMembership` and if their voting power drops beneath the threshold, they will be removed from the GSC. `kick` contains the following code:

```
            for (uint256 i = 0; i < votingVaults.length; i++) {
                // If the vault is not approved we don't count its v
                if (coreVoting.approvedVaults(votingVaults[i])) {
                    // Call the vault to check last block's voting p
                    // Last block to ensure there's no flash loan or
                    // intra contract interaction
                    uint256 votes =
                        IVotingVault(votingVaults[i]).queryVotePower
                            who,
                            block.number - 1,
                            extraData[i]
```

```
                );
                // Add up the votes
                totalVotes += votes;
            }
        }
```

As we see, `queryVotePower` will be called again on each vault. Let's see how `queryVotePower` is implemented in `BaseVotingVault`, which is used as a base contract for some voting contracts:

```
    function queryVotePower(address user, uint256 blockNumber, b
        // Get our reference to historical data
        History.HistoricalBalances memory votingPower = _votingP


        // Find the historical data and clear everything more th
        return votingPower.findAndClear(user, blockNumber, block
    }
```

As we see, it will always call the `findAndClear` function that will return the most recent voting power and will attempt to erase some older entries. New entries are added for a user when their voting power changes and no more than `1` entry is added to each block (if several changes happen in one block, values are just updated).

Now, the attacker (Bob) may perform the following attack:

1. Bob acquires enough votes to become a GSC member (they can either just buy enough tokens or deploy a smart contract that will offer a high yield for users who stake their vault tokens there).
2. Bob calls `proveMembership` and specifies `NFTBoostVault` as a proof. They will be accepted.
3. Bob "poisons" their voting power history by delegating from and redelegating to themself some dust token amount in several thousand different blocks (possible to do in less than `12h` on Ethereum).
4. Bob withdraws all their tokens from `NFTBoostVault`.

5. Alice spots that Bob doesn't have enough voting power anymore and will attempt to kick them, but since `findAndClear` will try to erase several thousand entries, it will exceed the Ethereum block limit for gas and the transaction will revert with Out Of Gas exception (`kick` will iterate over all vaults supplied by Bob, so Alice is forced to call `queryVotePower` on the vault that Bob used for the attack).
6. Bob can now send tokens to their other account, repeat the attack and they can do this until they have `>50%` in the GSC.

A similar exploit was presented by myself in a different submission, but this one is different; since the previous one focuses on a different aspect of that DoS attack - changing voting outcome. Here, I'm showing how somebody can permanently become a GSC member.

As reported in that different submission, the cost of performing the attack once is `~14ETH` , so it's not that much (currently `14ETH = $1880 * 14 = $26320`), but it may be worth it to perform this attack in order to be able to get `>50%` of GSC.

Impact

An attacker is able to become a permanent GSC member, even if they don't stake any tokens; which shouldn't be allowed and already has a huge impact on the protocol.

Even worse, they may try to acquire `> 50%` of voting power (GSC shouldn't have too many members - probably about `10` or something like this). Still, the GSC owner has `100,000` votes, but it is a time-lock contract, so might not be able to react fast enough to veto malicious proposals. Even if it is, this attack will destroy the entire GSC (since, from now on, the owner will decide about everything making GSC members useless), which is an important concept in the Arcade protocol.

Hence, the impact is huge (and assets can be lost since GSC is able to spend some tokens from the treasury) and there aren't any external factors allowed. So, I'm submitting this issue as High.

Proof of Concept

Several modifications are necessary in order to run the test. They are only introduced to make testing easier and don't have anything in common with the attack itself. First of all, please change `Authorizable::setOwner` as follows:

```
    function setOwner(address who) public /*onlyOwner()*/ {
```

Then, please change `NFTBoostVault` so that withdrawals are possible:

```
constructor(
        IERC20 token,
        uint256 staleBlockLag,
        address timelock,
        address manager
    ) BaseVotingVault(token, staleBlockLag) {
        if (timelock == address(0)) revert NBV_ZeroAddress("time
        if (manager == address(0)) revert NBV_ZeroAddress("manag

        Storage.set(Storage.uint256Ptr("initialized"), 1);
        Storage.set(Storage.addressPtr("timelock"), timelock);
        Storage.set(Storage.addressPtr("manager"), manager);
        Storage.set(Storage.uint256Ptr("entered"), 1);
        Storage.set(Storage.uint256Ptr("locked"), 0); // line ch
    }
```

Then, please add a basic `ERC20` token implementation to a `TestERC20.sol` file in the `contracts/test` directory (it's only used to mint some tokens to the users):

```
// SPDX-License-Identifier: MIT

pragma solidity 0.8.18;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";

contract TestERC20 is ERC20
{
    constructor() ERC20("TestERC20", "TERC20") {

    }
```

```
    function mint(address to, uint256 amount) external {
        _mint(to, amount);
    }
}
```

Finally, please put the following test inside `ArcadeGscVault.ts` (`import { mine } from "@nomicfoundation/hardhat-network-helpers";` will have to be also inserted there):

```
describe("Unkickable from GSC vault", async () => {
        it("Unkickable from GSC vault", async () => {
            const { coreVoting, arcadeGSCVault } = ctxGovernance
            const signers = await ethers.getSigners();
            const owner = signers[0];
            const Alice = signers[1];
            const Bob = signers[2];

            // balance of each user in TestERC20 custom token
            const ALICES_BALANCE = ethers.utils.parseEther("100
            const BOBS_BALANCE = ethers.utils.parseEther("100");

            const TestERC20Factory = await ethers.getContractFac
            const TestERC20 = await TestERC20Factory.deploy();

            // mine some block in the future to resemble mainnet
            await mine(1_000_000);

            // deploy NFTBoostVault with custom token (TestERC20
            // balance to users so that they can stake their tok
            const NFTBoostVaultFactory = await ethers.getContrac
            const NFTBoostVault = await NFTBoostVaultFactory.dep

            // set owner just to be able to call `changeVaultSta
            await coreVoting.connect(owner).setOwner(owner.addre
            await coreVoting.connect(owner).changeVaultStatus(NF

            // mint TestERC20 to users so that they can stake th
            await TestERC20.connect(Alice).mint(Alice.address, A
            await TestERC20.connect(Bob).mint(Bob.address, BOBS_

            // everyone approves TestERC20, so that they can sta
            await TestERC20.connect(Alice).approve(NFTBoostVault
            await TestERC20.connect(Bob).approve(NFTBoostVault.a
```

```
// Alice and Bob add some tokens and delegate
await NFTBoostVault.connect(Alice).delegate(Alice.ac
await NFTBoostVault.connect(Alice).addTokens(ALICES_

await NFTBoostVault.connect(Bob).delegate(Bob.addres
await NFTBoostVault.connect(Bob).addTokens(BOBS_BALA

// Alice becomes GSC member since she has enough vot
expect(await arcadeGSCVault.members(Alice.address)).
await arcadeGSCVault.connect(Alice).proveMembership(
expect(await arcadeGSCVault.members(Alice.address)).

// Bob also becomes GSC member, but when he unstakes
await arcadeGSCVault.connect(Bob).proveMembership([N
expect(await arcadeGSCVault.members(Bob.address)).nc

await NFTBoostVault.connect(Bob).withdraw(BOBS_BALAN
await arcadeGSCVault.connect(Alice).kick(Bob.address
expect(await arcadeGSCVault.members(Bob.address)).to
// kicking out Bob succeeds

// Bob adds tokens again and becomes GSC member, but
await TestERC20.connect(Bob).approve(NFTBoostVault.a
await NFTBoostVault.connect(Bob).delegate(Bob.addres
await NFTBoostVault.connect(Bob).addTokens(BOBS_BALA
await arcadeGSCVault.connect(Bob).proveMembership([N

// attack
// Bob performs it on himself
var gasUsed = 0;
for (var i = 0; i < 3500; i++)
{
    const tx1 = await NFTBoostVault.connect(Bob).del
    // impossible to change current delegatee to the
    const tx2 = await NFTBoostVault.connect(Bob).del
    const r1 = await tx1.wait();
    const r2 = await tx2.wait();
    gasUsed += r1.cumulativeGasUsed.toNumber();
    gasUsed += r2.cumulativeGasUsed.toNumber();
}
console.log(`Gas used by the attacker: ${gasUsed}`);

// Bob withdraws his tokens
await NFTBoostVault.connect(Bob).withdraw(BOBS_BALAN
```

```
                // Alice cannot kick out Bob
                await expect(arcadeGSCVault.connect(Alice).kick(Bob.

                // Bob is still GSC member; he can now transfer all
                // the attack again until he controls > 50% of GSC
                expect(await arcadeGSCVault.members(Bob.address)).nc

            }).timeout(400000);
        });
```

Tools Used

VS Code, hardhat

Recommended Mitigation Steps

Change `BaseVotingVault::queryVotePower` implementation so that it calls `find` instead of `findAndClear` (as in the `queryVotePowerView`).

Assessed type

DoS

PowVT (Arcade.xyz) confirmed and commented:

> Need to add a check to the `NFTBoostVault delegate` function to check that a registration exists for the user calling it. This is a valid submission and will be addressed.

Oxean (judge) decreased severity to Medium and commented:

> @PowVT - consider the C4 criteria, I don't see how this results in a direct loss of funds; therefore, should probably be Medium.

> What are your thoughts?

> For reference - https://docs.code4rena.com/awarding/judging-criteria/severity-categorization#estimating-risk

> Going to downgrade to Medium for now.

PowVT (Arcade.xyz) commented:

> The vulnerability they mention about being 'unkickable' is a side effect of the `NFTBoostVault` not being used properly. The user should not be allowed to call `delegate` without an existing registration. I would agree with medium.

PowVT (Arcade.xyz) commented:

> The vulnerability they mention about being 'unkickable' is a side effect of the NFTBoostVault not being used properly. The user should not be allowed to call `delegate` without an existing registration. I would agree with medium.

> Update here, I was incorrect in saying this solves the 'unkickable' issue. A check will be added so that the voting history does not exceed a certain length and does not cause out of gas reverts. Instead of pushing to the voting power array every time on delegating, we will add checks to confirm the length does not exceed X value (10 maybe).

0xnev (warden) commented:

> Hi @PowVT - When I was investigating this issue, I found out that actually this is a known issue in the docs of elementals council here, section 4 point 2. The root cause of the problem is the small staleblocklag set when deploying the `NFTBoostVault` / `VestingVault`.

> https://docs.element.fi/governance-council/council-protocol-smart-contracts/voting-vaults/governance-steering-council-gsc-vault

> I don't think this invalidates the finding since you guys seem to not know about this, but the elemental council sets a sufficiently high staleblocklag of `200000` that can be seen here. But if you do know, then this falls under admin input issues and could be invalidated under C4 rules; which I have tested against this submission and helps prevent the OOG problem stated in this submission. Hope this helps!

[M-04] User voting power is not synchronized after multiplier changes

Submitted by ktg, *also found by* Viktor_Cortess, oakcobalt, Oxmuxyz, Oxbranded, 0x3b, caventa, sces60107, 0xnev, BenRai, *and* 0xastronatey

Impact

- User voting power is not synchronized after multiplier changes.
- If the manager decreases the multiplier for a token, users' voting power will not be affected.

Proof of Concept

Currently the function `setMultiplier` of contract `NFTBoostVault` is implemented as follows:

```
function setMultiplier(address tokenAddress, uint128 tokenIc
    if (multiplierValue > MAX_MULTIPLIER) revert NBV_Multipl

    NFTBoostVaultStorage.AddressUintUint storage multiplierI
    // set multiplier value
    multiplierData.multiplier = multiplierValue;

    emit MultiplierSet(tokenAddress, tokenId, multiplierValu
}
```

The problem with this code is that it does not update the user's voting power, even though their multiplier has changed. The contract did provide a function called `updateVotingPower` to update users' voting power as follows:

```
function updateVotingPower(address[] calldata userAddresses)
    if (userAddresses.length > 50) revert NBV_ArrayTooManyEl

    for (uint256 i = 0; i < userAddresses.length; ++i) {
        NFTBoostVaultStorage.Registration storage registrati
        _syncVotingPower(userAddresses[i], registration);
    }
}
```

We can argue that the manager can call this function to update users voting power after they change the multiplier; however, there are multiple problems with this

solution:

- The input of `updateVotingPower` is a list of user addresses, so the manager must have a map of the token id to a list of users using that token ID as their multiplier. However, this data is not saved in the contract. When users register, their registrations are tracked using a variable of type `map(address => NFTBoostVaultStorage.Registration)`. I've looked into the contracts and found no way to retrieve a list of users associated with a token ID, even the function `_registerAndDelegate` does not emit an event showing which `ERC1155` token ID the users used at registration time.
- The function `updateVotingPower` only allows updating 50 addresses at a time, so if there are many users associated with a particular token ID, they have more time to try front running the manager.

Below is a POC for this issue. Place this file under the `test` folder with the name `VotingPowerNotUpdatedAfterMultiplierChange.ts` and run it using the command `npx hardhat test test/VotingPowerNotUpdatedAfterMultiplierChange.ts`.

The flow of this POC is as follows:

- Alice is minted a reputation badge.
- The reputation badge is then set as a multiplier by nft boost vault manager.
- Alice uses this reputation badge to boost their voting power.
- After the manager changes Alice's badge multiplier, their voting power is unchanged.

▶ Details

Recommended Mitigation Steps

I recommend creating a map of token IDs to a list of users, using that token as a multiplier, and synchronizing their voting power in the function `setMultiplier`.

PowVT (Arcade.xyz) disagreed with severity, acknowledged and commented via duplicate issue #431:

> Should be medium severity. This is a constraint with the current design of the multiplier mechanism. This is the intended functionality. Multiplier voting power is snapshotted at the time of proposal creation, even if a multiplier updates after it was current at the time of proposal creation.

0xean (judge) commented via duplicate issue #431:

> Agree on Medium.

0xepley (warden) commented:

> This is intentional and by design. I noticed the same issue and then read the following on the details page of code4rena:

> **Due to gas constraints, this will not immediately update the voting power of users who are using this NFT for a boost. However, any user's voting power can be updated by any other user via the `updateVotingPower` function - this value will look up the current multiplier of the user's registered NFT and recalculate the boosted voting power. This can be used in cases where obsolete boosts may be influencing the outcome of a vote.**

> Given that, it's unfair to other wardens. Better be Low, in my opinion.

Viktor_Cortess (warden) commented:

> @0xepley - https://github.com/code-423n4/2023-07-arcade-findings/issues/431.

> There is a discussion of this issue. One of the problems with the update mechanism is that nobody knows the addresses of the users whose votes should be updated. So if the user's multiplier decreases, they can avoid updating their voting power.

0xean (judge) commented:

> Given that it's unfair to other wardens. Better be Low, in my opinion.

> This isn't a consideration in the severity of an issue and I think it's worth highlighting the current design flaw that I think qualifies as Medium. This could be thought of as a "leak of value" since votes have some value, and this allows for a user to have more votes than expected in some scenarios.

[M-05] Users who have claimed before the update on the merkle tree will have no shares of the rewards afterward

Submitted by **0x3b**, *also found by* **osmanozdemir1** *and* **T1MOH**

Users who have claimed before an update on the merkle tree in **ArcadeAirdrop** are not going to be able to claim the extra rewards. The reward tokens will be left stuck in the contract, waiting for the manager to reclaim them.

Proof of Concept

Due to the nature of how `ArcadeAirdrop` is made and **the expectation** that there might be a change in the merkle tree in the near future, users will expect to be able to claim the extra rewards (if they are increased on merkle root change). However, because there is a problematic **if stamen** (that is inherited in `ArcadeAirdrop` by **ArcadeMerkleRewards**) users that have already claimed their rewards will have no share of the future ones.

Example:

1. Alice and Bob both have **100** tokens set to be claimed on the Merkle tree.
2. Alice claims their **100** tokens.
3. The owner sees that there is a huge activity on the platform and wants to incentivize users, so they increase rewards (by changing the merkle root) to **120** tokens.
4. Bob claims their **120** tokens
5. Alice sees that and tries to claim the remainder of their **120** tokens (**20**), but the claim reverts because of this **if**

```
    function _validateWithdraw(uint256 totalGrant, bytes32[] mem
        ...
        //@audit if the user has already claimed, he is not able
```

```
        if (claimed[msg.sender] != 0) revert AA_AlreadyClaimed()
        claimed[msg.sender] = totalGrant;
    }
```

Recommended Mitigation Steps

Change the claim implementation the way how mint in `ReputationBadge` is done. They have the total amount in the merkle tree and track claimed amount in a mapping.

Assessed type

DoS

__141345__ (lookout) commented:

> Invalid if the airdrop is one time.

> Admin's mistake. If there's a change in airdrop amount and merkle root, the user can not claim the difference.

PowVT (Arcade.xyz) confirmed

[M-06] `ArcadeTreasury.sol` allowance may be overriden

Submitted by **bin2chen**, *also found by* **ladboy233**, **caventa**, *and* **0xWaitress**

Direct use of `IERC20(token).approve(spender, amount);` causes the same `spender` allowances to be overridden by each other.

Proof of Concept

In the `gscApprove()` method, it is possible to give `spender` a certain allowance.

The code is as follows:

```
    function gscApprove(
        address token,
```

```
        address spender,
        uint256 amount
    ) external onlyRole(GSC_CORE_VOTING_ROLE) nonReentrant {
        if (spender == address(0)) revert T_ZeroAddress("spender
        if (amount == 0) revert T_ZeroAmount();

        // Will underflow if amount is greater than remaining al
@>      gscAllowance[token] -= amount;

        _approve(token, spender, amount, spendThresholds[token].
    }

    function _approve(address token, address spender, uint256 am
        // check that after processing this we will not have spe
        uint256 spentThisBlock = blockExpenditure[block.number];
        if (amount + spentThisBlock > limit) revert T_BlockSpenc
        blockExpenditure[block.number] = amount + spentThisBlock

        // approve tokens
@>      IERC20(token).approve(spender, amount);

        emit TreasuryApproval(token, spender, amount);
    }
```

From the above code, we can see that when executed, `gscApprove` consumes `gscAllowance[]` and ultimately uses `IERC20(token).approve();` to give the `spender` allowance. Since the direct use is `IERC20.approve(spender, amount)`, the amount of the allowance is overwritten, whichever comes last.

In the other method,s `approveSmallSpend`, `approveMediumSpend` and `approveLargeSpend` also use `IERC20(token).approve();`, which causes them to override each other if targeting the same `spender`.

Even if there is a malicious `GSC_CORE_VOTING_ROLE`, it is possible to execute `gscApprove(amount=1 wei)` after `approveLargeSpend()` to reset to an allowance of only `1 wei`.

The recommendation is to use accumulation to avoid, intentionally or unintentionally, overwriting each other.

Recommended Mitigation Steps

```
    function _approve(address token, address spender, uint256 am
        // check that after processing this we will not have spe
        uint256 spentThisBlock = blockExpenditure[block.number];
        if (amount + spentThisBlock > limit) revert T_BlockSpenc
        blockExpenditure[block.number] = amount + spentThisBlock

        // approve tokens
-       IERC20(token).approve(spender, amount);
+       uint256 old = IERC20(token).allowance(address(this),spenc
+       IERC20(token).approve(spender,  old + amount);

        emit TreasuryApproval(token, spender, amount);
    }
```

Assessed type

Context

__141345__ (lookout) commented:

> https://github.com/code-423n4/2023-07-arcade-findings/issues/58 talks about the issue with internal accounting of `gscAllowance`. Another aspect of the issue.

> This report shows some unexpected results due to the external `erc20` allowance overwrite.

PowVT (Arcade.xyz) confirmed

0xepley (warden) commented:

> Doesn't it fall under the admin mistake? The admin would be aware and can change these to desired values anytime.

0xean (judge) commented:

> Doesn't it fall under the admin mistake? The admin would be aware and can change these to desired values anytime.

> I don't believe this should be viewed as input sanitization, as the core functionality is written in a way that has unexpected consequences that could affect the functionality of the protocol. I can see arguing for QA, I welcome more comments from other wardens or sponsors.

0xnev (warden) commented:

> @0xean - correct me if I'm wrong, but I think it is intended for allowance to be overwritten for the spender when it is decreased, given the changing of approval needs to go through a GSC proposal.

> But since this report also highlights the potential inability to reduce allowance due to underflow in the `gscAllowance` mapping, it should be duplicated with #58.

> Again correct me if I'm wrong, If the above reasoning is correct, #55, #146, and #535 should be invalidated since they do not mention the possible DoS when decreasing allowance.

[M-07] `DAY_IN_BLOCKS` is incorrect

Submitted by Anirruth, *also found by* DadeKuma, bart1e, ladboy233, Matin, giovannidisiena, rvierdiiev, *and* MohammedRizwan

The day in blocks is calculated with the block time as 13.3 seconds in CoreVoting.sol.: `uint256 public constant DAY_IN_BLOCKS = 6496;` , but since moving to proof of stake, block times are fixed to 12 seconds per block. Reference.

This results in an incorrect calculation of the `lockDuration` and `extraVoteTime` , which is used in setting the total duration of a proposal should be active and also the max vote time.

The time difference can be calculated:

`3*24*60*60 / 13.3` = 19488.721804511 (`lockDuration` with 13.3 seconds).
`3*24*60*60 / 12` = 21600 (`lockDuration` with 12 seconds).
21600 - 19488.7 = 2111.3
`2111.3*12 / (60*60)` = 7.03 (difference in hours for `lockDuration`).

`5*24*60*60 / 13.3` = 32481.203007519 (`extraVoteTime` with 13.3 seconds).

`5*24*60*60 / 12` = 36000 (`extraVoteTime` with 12 seconds).

36000 - 32481.2 = 3518.8

`3518.8*12 / (60*60)` = 11.72 (difference in hours `extraVoteTime`).

By using block time as 13.3 seconds, the `lockDuration` expires 7 hours earlier and the `extraVoteTime` expires 11.72 hours earlier. Since it is a significant time and affects the proposal and voting duration, I consider medium severity to be fair.

Tools Used

VS code

Recommended Mitigation Steps

`86400 / 12` = 7200

Change the `DAY_IN_BLOCKS` to 7200:

```
uint256 public constant DAY_IN_BLOCKS = 7200;
```

Assessed type

Error

PowVT (Arcade.xyz) disagreed with severity, confirmed and commented via duplicate issue #56:

> QA - Having a shorter `staleBlockLag` doesn't affect the contract's behavior at all really. Just makes things a bit more confusing for those reading the code. We will accept this since it is a very straightforward change and helps the readability of the contract.

0xean (judge) decreased severity to QA via duplicate issue #56

MohammedRizwan (warden) commented via duplicate issue #56:

> @0xean @PowVT - With due respect to the sponsor and judge, I would like to add below additional context:

> This should be considered a valid Medium. It's not about the readability of the code, but about the design consideration which is currently deviating. 13.3 seconds was for proof of work and the proposed 12 seconds is for the proof of stake which happened after the Ethereum merge. The overall impact has been described in the report and it's duplicate.

> Let's consider a case related to voting and the number of blocks considered in the current implementation:

```
uint256 public constant DAY_IN_BLOCKS = 6496;
```

> The lock duration is 3 days by default:

```
uint256 public lockDuration = DAY_IN_BLOCKS * 3;
```

> Lock duration in number of blocks = `19_488` .

> Now, as I said, Ethereum produces blocks at 12 seconds. Ethereum official reference and chart.

> Therefore, the lock duration will expire in 2.7 days instead of 3 days.

> How? Let's see below:

> Lock duration in number of blocks = `19_488` blocks
> Block formation period = 12 seconds
> Lock duration will expire in = `19_488 X 12 = 2,33,856 seconds` ~ 2.7 days

> Such `blockduration` issues have been judged as Medium/High in past Code4rena audits. I am putting below the reference report link from backstage as the report is not public yet. This reference report was judged by Trust.

> The in-scope Issue 2 about `staleBlockLag` is to be considered as `between 1 and 2 months (equivalent in blocks)` per sponsor response in a discord chat.

> Let's consider 1 month for `staleBlockLag` :

> With current implementation: `staleBlockLag` with 1 month(equivilent in blocks) = `1_94_887` .

> With proposed recommendation: `staleBlockLag` with 1 month(equivilent in blocks) = `2_16_000` .

> Total difference = `21_113` blocks.

> It means `staleBlockLag` will expire before 2.93 days i.e. ~ 27 days instead of 30 days (1 month as designed for).

> This is a major change in arcade protocol design reducing the block duration time to 12 seconds instead of 13.3 seconds. This will affect not only contracts in scope but it will also the out-of-scope contracts relating to voting calculations.

> I believe this should be considered a Medium.

Oxean (judge) increased severity to Medium and commented via duplicate issue #56:

> 2 — Med: 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

> After reviewing this all further, and re-reading the C4 documentation I think this does qualify as Medium. While there are setters for some of these variables that could be modified to end up with the correct values, this issue *does* impact the functionality of the protocol.

> I will go ahead and upgrade all of these to Medium.

[M-08] Approved `gscApprove` allowance to an address may not able to be decreased

Submitted by Juntao, also found by Juntao, UniversalCrypto, vangrim, Jiamin (1, 2), crunch (1, 2), zaevlad, auditsea, circlelooper, and lanrebayode77

Proof of Concept

`GSC_CORE_VOTING_ROLE` calls the `gscApprove(...)` function to approve tokens to be pulled from the treasury:

```
        function gscApprove(
            address token,
            address spender,
            uint256 amount
        ) external onlyRole(GSC_CORE_VOTING_ROLE) nonReentrant {
            if (spender == address(0)) revert T_ZeroAddress("spender
            if (amount == 0) revert T_ZeroAmount();


            // Will underflow if amount is greater than remaining al
            gscAllowance[token] -= amount;


            _approve(token, spender, amount, spendThresholds[token].
        }
```

Each approval will decrease the `gscAllowance[token]` by the approved allowance amount, this is problematic as `GSC_CORE_VOTING_ROLE` may not able to decrease the approved allowance.

Consider the following scenario:

1. `gscAllowance[token]` is 100.
2. `GSC_CORE_VOTING_ROLE` calls `gscApprove(...)` to give 60 allowance to a third party.
3. `gscAllowance[token]` is 40 now;
4. Later, `GSC_CORE_VOTING_ROLE` finds the approved allowance is a bit too high and wants to decrease the allowance to 50.
5. `gscApprove(...)` is called again, but the transaction reverts due to an underflow error (`gscAllowance[token]` is less than 50).

Please see the tests:

```
contract ArcadeTreasuryTest is Test {
    address timelock = address(1);
    address coreVoting = address(2);
    address gscCoreVoting = address(3);
    address other = address(4);

    ArcadeTreasury treasury;
    MockToken mockToken;

    function setUp() public {
        treasury = new ArcadeTreasury(timelock);

        vm.startPrank(timelock);
        treasury.grantRole(treasury.CORE_VOTING_ROLE(), coreVoti
        treasury.grantRole(treasury.GSC_CORE_VOTING_ROLE(), gscC

        mockToken = new MockToken("Mock Token", "MT");
        IArcadeTreasury.SpendThreshold memory threshold = IArcac
        treasury.setThreshold(address(mockToken), threshold);
        vm.stopPrank();
    }

    function testGscApprove() public {
        vm.warp(7 days);

        vm.prank(timelock);
        treasury.setGSCAllowance(address(mockToken), 100 ether);

        vm.startPrank(gscCoreVoting);
        treasury.gscApprove(address(mockToken), address(4), 60 e
        vm.expectRevert(stdError.arithmeticError);
        treasury.gscApprove(address(mockToken), address(4), 50 e
        vm.stopPrank();
    }
}

contract MockToken is ERC20 {
    constructor(string memory name_, string memory symbol_) ERC2

    function mint(address account, uint256 amount) public {
        _mint(account, amount);
    }
}
```

Recommended Mitigation Steps

When approving a particular address, compare the new allowance with the current allowance (IERC20.allowance(...)):

1. If the current allowance is equal to the new allowance, do nothing.
2. If the current allowance is less than the new allowance, `gscAllowance[token] -= (new allowance - current allowance)`.
3. If the current allowance is larger than the new allowance, `gscAllowance[token] += (current allowance - new allowance)`.

Assessed type

Access Control

__141345__ (lookout) commented:

> An already approved allowance can not be decreased. Might be an expected behavior.

> Because in this report, the focus is inner accounting `gscAllowance`, not the external ERC20 allowance Issue #85. Here, the approved allowance seems different from the "ERC20 approved allowance". Here, it can be seen as an account payable, the amount is spent, but the spender has not taken it yet.

> Or, if the designed behavior is to also provide a way to revoke the allowance, this inner accounting needs some improvement.

> To my understanding, the issue is rooted in the un-sync of internal and external allowance accounting.

> Issue #85 also talks about the allowance issue. It also focuses more on ERC20 approved allowance override, which seems better pointing out the potential impact of with this mechanism. Some way to revoke the allowance, if the allowance is overwritten.

PowVT (Arcade.xyz) disagreed with severity and confirmed

Oxean (judge) decreased severity to Medium and commented:

> Based on my current understanding, I think Medium is the correct severity here.

M-09 Approve zero first

Submitted by c4lanky

Note: this finding was reported via the winning Automated Findings report*. It was declared out of scope for the audit, but is being included here for completeness.*

Some ERC20 tokens (like USDT) do not work when changing the allowance from an existing non-zero allowance value. For example, Tether (USDT)'s `approve()` function will revert if the current approval is not zero, to protect against front-running changes of approvals.

There are 1 instances of this issue: File: contracts/ArcadeTreasury.sol

```
384:     function _approve(address token, address spender, uint2
385:         // check that after processing this we will not hav
386:         uint256 spentThisBlock = blockExpenditure[block.num
387:         if (amount + spentThisBlock > limit) revert T_Block
388:         blockExpenditure[block.number] = amount + spentThis
389:
390:         // approve tokens
391:         IERC20(token).approve(spender, amount);
392:
393:         emit TreasuryApproval(token, spender, amount);
394:     }
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/ArcadeTreasury.sol#L384-L394

PowVT (Arcade) confirmed

M-10 The surplus ether is not returned

Submitted by c4lanky

Note: this finding was reported via the winning Automated Findings report. It was declared out of scope for the audit, but is being included here for completeness.

The function is marked as payable, but the surplus ether is not returned. This may lead to the loss of ether.

The condition should be changed to check for equality, or the code should refund the excess.

There are 1 instances of this issue: File: contracts/nft/ReputationBadge.sol

```
98:     function mint(
99:         address recipient,
100:         uint256 tokenId,
101:         uint256 amount,
102:         uint256 totalClaimable,
103:         bytes32[] calldata merkleProof
104:     ) external payable {
105:         uint256 mintPrice = mintPrices[tokenId] * amount;
106:         uint48 claimExpiration = claimExpirations[tokenId];
107: 
108:         if (block.timestamp > claimExpiration) revert RB_Cl
109:         if (msg.value < mintPrice) revert RB_InvalidMintFee
110:         if (!_verifyClaim(recipient, tokenId, totalClaimabl
111:         if (amountClaimed[recipient][tokenId] + amount > to
112:             revert RB_InvalidClaimAmount(amount, totalClaim
113:         }
114: 
115:         // increment amount claimed
116:         amountClaimed[recipient][tokenId] += amount;
117: 
118:         // mint to recipient
119:         _mint(recipient, tokenId, amount, "");
120:     }
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/nft/ReputationBadge.sol#L98-L120

PowVT (Arcade) confirmed

M-11 Use `_safeMint` instead of `_mint` for ERC721

Submitted by c4lanky

Note: this finding was reported via the winning Automated Findings report*. It was declared out of scope for the audit, but is being included here for completeness.*

`_mint()` is discouraged in favor of `_safeMint()`, which ensures that the recipient is either an EOA or implements IERC721Receiver. Both OpenZeppelin and solmate have versions of this function.

There are 1 instances of this issue: File: contracts/nft/ReputationBadge.sol

```
119:         _mint(recipient, tokenId, amount, "");
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/nft/ReputationBadge.sol#L119

PowVT (Arcade) confirmed

M-12 Unsafe ERC20 operations

Submitted by c4lanky

Note: this finding was reported via the winning Automated Findings report*. It was declared out of scope for the audit, but is being included here for completeness.*

Certain tokens may not adhere to the ERC20 standard completely, yet they are generally accepted by most code designed for ERC20 tokens. An instance of this is Tether (USDT), where the `transfer()` and `transferFrom()` functions on L1 do not conform to the specification's requirement of returning booleans; instead, they have no return value.

Consequently, when such tokens are cast to IERC20, their function signatures do not align, leading to reverted calls (refer to this link for a test case). To mitigate this

issue, it is recommended to utilize OpenZeppelin's SafeERC20's `safeTransfer()` and `safeTransferFrom()` functions instead.

There are 2 instances of this issue: File: contracts/ARCDVestingVault.sol

```
201:         token.transferFrom(msg.sender, address(this), amour
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/ARCDVestingVault.sol#L201

File: contracts/NFTBoostVault.sol

```
657:         token.transferFrom(from, address(this), amount);
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/NFTBoostVault.sol#L657

PowVT (Arcade) confirmed

Low Risk and Non-Critical Issues

For this audit, 20 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **LaScaloneta** received the top score from the judge.

The following wardens also submitted reports: QiuhaoLi, BugBusters, DadeKuma, 0xComfyCat, Udsen, bart1e, zhaojie, ak1, ladboy233, 0xDING99YA, immeas, oakcobalt, squeaky_cactus, matrix_0wl, koxuan, Sathish9098, ABA, 0xnev, *and* MohammedRizwan.

[L-01] The merkle root won't revert if is set to `0x00`

If merkle root is set to `0x00` it could leave the system in an insecure state where the system will accept any message that it has never seen before and process it as if it were genuine.

This has happened before in the [nomad bridge incident](#).

> The Nomad team initialized the trusted root to be `0x00`. To be clear, using zero values as initialization values is a common practice.

Instances:

- [ArcadeMerkleRewards.sol#L62](#)
- [MerkleRewards.sol#L28](#)
- [ArcadeAirdrop.sol#L76](#)

[L-02] `delegate()` can be called with no voting power

The `delegate()` functions in `ARCDVestingVault` and `NFTBoostVault` can be called by anyone, despite them not having any voting power.

This leads to changes in the storage variables, like pushing empty entries to the `votingPower`, and setting a `delegatee`, despite not delegating anything.

As it will set a `registration.delegatee`, it will bypass the validations `if (registration.delegatee == address(0)) revert NBV_NoRegistration();` on `addTokens()`, and `updateNft()`. Making it possible to call those functions without "initializing" the corresponding registration storage via `addNftAndDelegate()`.

This can lead to unexpected effects.

Instances:

- [NFTBoostVault.sol#L182-L212](#)
- [ARCDVestingVault.sol#L260-L282](#)

Recommended Mitigation Steps

Verify that `_currentVotingPower(registration) > 0`.

[L-03] Use a two-step transfer function for the Minter role in `ArcadeToken`

The `ArcadeToken` has a very important Minter role which can be changed via a `setMinter()` function.

In order to prevent the role from mistakenly being transferred to an address that cannot handle it (e.g. due to a typo in the address), require that the recipient of the new minter role actively accepts via a contract call of its own.

Instances:

- ArcadeToken.sol#L132-L137

[L-04] `NFTBoostVault::updateNft()` allows to deposit any NFT

The function `addNftAndDelegate` reverts when depositing any random NFT because of the `if (multiplier == 0) revert NBV_NoMultiplierSet();` in `_registerAndDelegate()`.

But `updateNft()` does not have that check, which allows anyone to update and deposit any NFT, leading to a multiplier of 0; and thus a voting power of 0, despite the user having deposited any other tokens.

This might also be combined with other issues for a bigger severity issue. We recommend that the same check to revert on `multiplier == 0` is applied on `updateNft()`.

Instances:

- NFTBoostVault.sol#L305-L330
- NFTBoostVault.sol#L477

[L-05] If `token` has hooks, the grant owner can revert when admin try to revoke the role

An evil granted owner can avoid being `revoke`. They can revert triggering a `DOS` to avoid a manager to revoke the role.

[L-06] If `token` has hooks, the manager could `reenter` to `revokeGrant` draining the contract

A token with a hook could leave a reentrancy issue, letting the manager drain funds.

[L-07] The storage slots/hashes have known pre-images.

Knowledge of pre-images might allow manipulation of specific mapping parameters by crafting special keys. This is possible if a mapping parameter's storage slot lands on a specific slot.

It might be best to decrement them by "one" so that finding the pre-image would be harder.

Instances:

- Storage.sol

Example:

```
bytes32 typehash = keccak256("mapping(address => AddressUint)");
bytes32 offset = keccak256(abi.encodePacked(typehash, name));
assembly {
    data.slot := sub(offset, 1) // keccak256(NAME) - 1, to avoic
}
```

Also please review this eip-1967

[L-08] Type of `tokenId` in `ReputationBadge` and `NftBoostVault` do not match and can lead to tokens not being usable as multipliers

`ReputationBadge` tokens are meant to be used as multipliers of voting power in `NftBoostVault`.

Users can mint badges with `tokenId` up to `uint256`, but `NftBoostVault` only allows to set `tokenId` up to `uint128` to work as multipliers.

Tokens with `tokenId > type(uint128).max` won't be able to be used as multipliers.

`tokenId` in `ReputationBadge` (uint256):

```
        function mint(
            address recipient,
    @>      uint256 tokenId,
            uint256 amount,
            uint256 totalClaimable,
            bytes32[] calldata merkleProof
        ) external payable {
```

ReputationBadge.sol#L100

`tokenId` in `NftBoostVault` (uint128):

```
    @>  function setMultiplier(address tokenAddress, uint128 tokenIc
```

NFTBoostVault.sol#L363

Consider setting the same `tokenId` type for the expected NFTs to be used in the contract.

[L-09] You can bypass the ERC1155 check and deposit anything if `tokenId == 0`

The check on NFTBoostVault.sol#L472 can be easily bypassed if the ERC1155 token id is `0`:

```
if (_tokenAddress != address(0) && _tokenId != 0) {
```

Proof of Concept

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.13;
import "forge-std/Test.sol";
import "../contracts/NFTBoostVault.sol";
import "../contracts/token/ArcadeToken.sol";

contract ExampleTest is Test {
    NFTBoostVault public boostVault;
    ArcadeToken public token;

    function setUp() public {
        vm.roll(10);
        token = new ArcadeToken(address(this), address(this));
        boostVault = new NFTBoostVault(IERC20(address(token)), 1
    }

    function testDepositAnyERC1155() public {
        // create a fake erc1155 (could also happen with a )
        FakeERC1155 f = new FakeERC1155();

        token.approve(address(boostVault),100);
        boostVault.addNftAndDelegate(
            1,
            0, // uint128 tokenId,
            address(f), //address tokenAddress,
            address(0xdead) // delegate
        );

        token.transfer(address(0x02), 1);
        vm.prank(address(0x02));
        token.approve(address(boostVault),100);

        vm.prank(address(0x02));
        boostVault.addNftAndDelegate(
            1,
            0, // uint128 tokenId,
            address(f), //address tokenAddress,
            address(0xdead01)
        );

        vm.prank(address(0xdead));
        boostVault.delegate(address(0x01));

        vm.prank(address(0xdead01));
        boostVault.delegate(address(0x01));
```

```
        }
    }

    contract FakeERC1155 {
        fallback() external {}
    }
```

[L-10] Approve zero first

Some ERC20 tokens (like USDT) do not work when changing the allowance from an existing non-zero allowance value. For example, Tether, (USDT)'s `approve()` function, will revert if the current approval is not zero to protect against front-running changes of approvals.

Instances:

- ArcadeMerkleRewards.sol#L86

Non-Critical Findings

[N-01] `NFTBoostVault` sets an unused `initialized` value in storage

The `"initialized"` storage variable is set on the constructor, but not used by the `NFTBoostVault`, or another contract, and can't be accessed by any public or external method.

Recommendation

Verify if it is actually needed, or remove it in case it is not.

```
    Storage.set(Storage.uint256Ptr("initialized"), 1); // @audit che
```

NFTBoostVault.sol#L91

[N-02] `ARCDVestingVault` inherits twice from `HashedStorageReentrancyBlock`

`ARCDVestingVault` inherits both from `HashedStorageReentrancyBlock` and `BaseVotingVault`. But `BaseVotingVault` is already inheriting from `HashedStorageReentrancyBlock`.

Remove the unnecessary inheritance from `HashedStorageReentrancyBlock` on `ARCDVestingVault`.

Instances:

- ARCDVestingVault.sol#L48

PowVT (Arcade.xyz) confirmed and commented:

> There are some good valid fixes in this report that we will address.

0xean (judge) commented:

> L03 - is largely already covered in the bot report. This is slightly different only its location in the codebase so doesn't hurt to include it in the report. L10 - is in the bot report already.

> Otherwise severities look reasonable to me.

Gas Optimizations

For this audit, 10 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **K42** received the top score from the judge.

The following wardens also submitted reports: c3phas, excalibor, dharma09, kaveyjoe, SM3_SS, Aymen0909, Raihan, Sathish9098, *and* jeffy.

Possible Optimizations in ArcadeTreasury.sol

General Optimization

Reducing the number of state variable updates can save gas. In Ethereum, every storage operation costs a significant amount of gas. Therefore, optimizing the contract to minimize storage operations can lead to substantial gas savings.

Possible Optimization 1

In the _spend and _approve functions, the blockExpenditure mapping is updated twice. This can be optimized to a single update, which would save gas.

Here is the optimized code snippet:

```
function _spend(address token, uint256 amount, address destinati
    uint256 spentThisBlock = blockExpenditure[block.number] + am
    if (spentThisBlock > limit) revert T_BlockSpendLimit();
    blockExpenditure[block.number] = spentThisBlock;
    if (address(token) == ETH_CONSTANT) {
        payable(destination).transfer(amount);
    } else {
        IERC20(token).safeTransfer(destination, amount);
    }
    emit TreasuryTransfer(token, destination, amount);
}

function _approve(address token, address spender, uint256 amount
    uint256 spentThisBlock = blockExpenditure[block.number] + am
    if (spentThisBlock > limit) revert T_BlockSpendLimit();
    blockExpenditure[block.number] = spentThisBlock;
    IERC20(token).approve(spender, amount);
    emit TreasuryApproval(token, spender, amount);
}
```

Estimated gas saved:
This optimization reduces the number of storage operations from 2 to 1 in each of the _spend and _approve functions. Given that a storage operation costs 20,000 gas, this optimization could save approximately 20,000 gas per call to these functions.

Possible Optimization 2

The batchCalls function can be optimized by removing the check if (spendThresholds[targets[i]].small != 0) revert T_InvalidTarget(targets[i]);. This

check is not necessary because if the target contract does not exist or does not have a function that matches the provided `calldata`, the call will fail anyway. Removing this check can save gas.

Here is the optimized code:

```
function batchCalls(
    address[] memory targets,
    bytes[] calldata calldatas
) external onlyRole(ADMIN_ROLE) nonReentrant {
    if (targets.length != calldatas.length) revert T_ArrayLength
    for (uint256 i = 0; i < targets.length; ++i) {
        (bool success, ) = targets[i].call(calldatas[i]);
        if (!success) revert T_CallFailed();
    }
}
```

Estimated gas saved:
This optimization removes a storage read operation from the `batchCalls` function. Given that a storage read operation costs 200 gas, this optimization could save approximately 200 gas per call to this function. However, the actual gas savings would be higher when considering the gas cost of the conditional check and the potential revert operation.

Possible Optimizations in BaseVotingVault.sol

General Optimization

The contract uses the `onlyManager` and `onlyTimelock` modifiers in multiple functions. These modifiers check if the caller is the `manager` or the `timelock`. However, these checks could be optimized by creating a new modifier that directly checks the `msg.sender` against the `manager` or `timelock`, saving the gas used by the function call.

Possible Optimization 1

The queryVotePower function calls the findAndClear function, which iterates over the votingPower mapping and clears entries that are older than `staleBlockLag`.

This could be optimized by only clearing entries when the number of entries exceeds a certain limit. This would reduce the number of storage operations, which are expensive in terms of gas.

After Optimization:

```
function queryVotePower(address user, uint256 blockNumber, bytes
    // Get our reference to historical data
    History.HistoricalBalances memory votingPower = _votingPower

    // Find the historical data and clear everything more than '
    // only if the number of entries exceeds a certain limit (e.
    if (votingPower.numEntries(user) > 100) {
        return votingPower.findAndClear(user, blockNumber, block
    } else {
        return votingPower.find(user, blockNumber);
    }
}
```

Estimated gas saved:
This optimization reduces the number of storage operations, which costs 20,000 gas each. The actual gas saved would depend on the number of entries in the `votingPower` mapping for a user. If there are more than 100 entries, this optimization could save approximately 20,000 gas per extra entry.

Possible Optimization 2

The setTimelock and setManager functions update the timelock and manager addresses. These functions could be optimized by checking if the new address is different from the current one before updating. This would save gas when the new address is the same as the current one.

After Optimization:

```
function setTimelock(address timelock_) external onlyTimelock {
    if (timelock_ == address(0)) revert BVV_ZeroAddress("timeloc
    if (timelock_ != timelock()) {
        Storage.set(Storage.addressPtr("timelock"), timelock_);
    }
}
```

```
function setManager(address manager_) external onlyTimelock {
    if (manager_ == address(0)) revert BVV_ZeroAddress("manager'
    if (manager_ != manager()) {
        Storage.set(Storage.addressPtr("manager"), manager_);
    }
}
```

Estimated gas saved:
This optimization could save around 5,000 to 10,000 gas per transaction, depending on the gas cost of the storage operation.

Possible Optimizations in ARCDVestingVault.sol

Possible Optimization 1

In the addGrantAndDelegate function, there are multiple calls to Storage.Uint256 storage unassigned = _unassigned(); and ARCDVestingVaultStorage.Grant storage grant = _grants()[who];. These calls could be optimized by declaring them once at the beginning of the function and reusing the reference. This would save the gas used by the function call.

After Optimization:

```
function addGrantAndDelegate(
    address who,
    uint128 amount,
    uint128 cliffAmount,
    uint128 startTime,
    uint128 expiration,
    uint128 cliff,
    address delegatee
) external onlyManager {
    Storage.Uint256 storage unassigned = _unassigned();
    ARCDVestingVaultStorage.Grant storage grant = _grants()[who]

    // input validation
    if (who == address(0)) revert AVV_ZeroAddress("who");
    if (amount == 0) revert AVV_InvalidAmount();

    // if no custom start time is needed we use this block.
```

```
        if (startTime == 0) {
            startTime = uint128(block.number);
        }
        // grant schedule check
        if (cliff >= expiration || cliff < startTime) revert AVV_Inv

        // cliff check
        if (cliffAmount >= amount) revert AVV_InvalidCliffAmount();

        if (unassigned.data < amount) revert AVV_InsufficientBalance

        // if this address already has a grant, a different address
        // topping up or editing active grants is not supported.
        if (grant.allocation != 0) revert AVV_HasGrant();

        // load the delegate. Defaults to the grant owner
        delegatee = delegatee == address(0) ? who : delegatee;

        // calculate the voting power. Assumes all voting power is i
        uint128 newVotingPower = amount;

        // set the new grant
        grant.allocation = amount;
        grant.cliffAmount = cliffAmount;
        grant.withdrawn = 0;
        grant.created = startTime;
        grant.expiration = expiration;
        grant.cliff = cliff;
        grant.latestVotingPower = newVotingPower;
        grant.delegatee = delegatee;

        // update the amount of unassigned tokens
        unassigned.data -= amount;

        // update the delegatee's voting power
        History.HistoricalBalances memory votingPower = _votingPower
        uint256 delegateeVotes = votingPower.loadTop(grant.delegatee
        votingPower.push(grant.delegatee, delegateeVotes + newVoting

        emit VoteChange(grant.delegatee, who, int256(uint256(newVoti
    }
```

Estimated gas saved:
This optimization could save around 1000 to 2000 gas per transaction, depending

on the gas cost of the function call.

Possible Optimization 2

In the claim function, there are multiple calls to ARCDVestingVaultStorage.Grant storage grant = _grants()[msg.sender];. This could be optimized by declaring it once at the beginning of the function and reusing the reference. This would save the gas used by the function call.

After Optimization:

```
function claim(uint256 amount) external override nonReentrant {
    ARCDVestingVaultStorage.Grant storage grant = _grants()[msg.

    if (amount == 0) revert AVV_InvalidAmount();

    if (grant.allocation == 0) revert AVV_NoGrantSet();
    if (grant.cliff > block.number) revert AVV_CliffNotReached(g

    // get the withdrawable amount
    uint256 withdrawable = _getWithdrawableAmount(grant);
    if (amount > withdrawable) revert AVV_InsufficientBalance(wi

    // update the grant's withdrawn amount
    if (amount == withdrawable) {
        grant.withdrawn += uint128(withdrawable);
    } else {
        grant.withdrawn += uint128(amount);
        withdrawable = amount;
    }

    // update the user's voting power
    _syncVotingPower(msg.sender, grant);

    // transfer the available amount
    token.safeTransfer(msg.sender, withdrawable);
}
```

Estimated gas saved:
This optimization could save around 1000 to 2000 gas per transaction, depending on the gas cost of the function call.

Possible Optimization 3

In the revokeGrant function, the contract performs a number of operations after checking if the grant allocation is zero. If the allocation is zero, these operations are unnecessary. By using a return statement after the revert, we can avoid these unnecessary operations.

Before:

```
// if the grant has already been removed or no grant available,
if (grant.allocation == 0) revert AVV_NoGrantSet();
```

After:

```
// if the grant has already been removed or no grant available,
if (grant.allocation == 0) {
    revert AVV_NoGrantSet();
    return;
}
```

Estimated gas saved:
This optimization could save around 5,000 to 10,000 gas per transaction, depending on the number of operations that are skipped.

Possible Optimizations in NFTBoostVault.sol

Possible Optimization 1

In the addNftAndDelegate function, the amount is checked to be non-zero at the beginning of the function. However, the amount is not used until after the _registerAndDelegate function call. Moving the zero check closer to the usage of the amount could save gas in the case where _registerAndDelegate reverts, as the zero check would not have been performed.

After Optimization:

```
function addNftAndDelegate(
```

```
        uint128 amount,
        uint128 tokenId,
        address tokenAddress,
        address delegatee
    ) external override nonReentrant {
        _registerAndDelegate(msg.sender, amount, tokenId, tokenAddre
        if (amount == 0) revert NBV_ZeroAmount();
        _lockTokens(msg.sender, uint256(amount), tokenAddress, toker
    }
```

Estimated gas saved:
This optimization could save around 200 to 500 gas per transaction, depending on the gas cost of the _registerAndDelegate function call.

Possible Optimization 2

In the _registerAndDelegate function, the multiplier is set to 1e3 by default and then potentially updated if _tokenAddress and _tokenId are non-zero. However, the getMultiplier function already returns 1e3 if _tokenAddress or _tokenId are zero. Therefore, the initial setting of the multiplier to `1e3` is unnecessary and can be removed.

After Optimization:

```
    if (_tokenAddress != address(0) && _tokenId != 0 && IERC1155(_tc
        revert NBV_DoesNotOwn();
    }
    uint128 multiplier = getMultiplier(_tokenAddress, _tokenId);
    if (multiplier == 0) revert NBV_NoMultiplierSet();
```

Estimated gas saved:
This optimization could save around 200 to 500 gas per transaction, depending on the gas cost of the `getMultiplier` function call.

Possible Optimization 3

In the _syncVotingPower function, the change variable is calculated and then checked to be non-zero. If the change is zero, the function returns early. However, the change calculation involves subtraction and two castings, which are relatively

expensive operations. By checking if newVotingPower and registration.latestVotingPower are equal before performing the change calculation, we can potentially avoid these operations.

After Optimization:

```
if (newVotingPower == registration.latestVotingPower) return;
int256 change = int256(newVotingPower) - int256(uint256(registra
```

Estimated gas saved:
This optimization could save around 500 to 1000 gas per transaction, depending on the gas cost of the subtraction and casting operations.

Possible Optimization in ArcadeToken.sol

In the mint function, the mintingAllowedAfter variable is updated before the minting cap check. This means that even if the minting cap check fails and reverts the transaction, the mintingAllowedAfter variable will still have been updated, wasting gas. Moving the mintingAllowedAfter update after the minting cap check could save gas in the case where the minting cap check fails.

After Optimization:

```
uint256 mintCapAmount = (totalSupply() * MINT_CAP) / PERCENT_DEN
if (_amount > mintCapAmount) {
    revert AT_MintingCapExceeded(totalSupply(), mintCapAmount, _
}
mintingAllowedAfter = block.timestamp + MIN_TIME_BETWEEN_MINTS;
```

Estimated gas saved:
This optimization could save around 5000 to 10000 gas per transaction, depending on the gas cost of the totalSupply function call and the multiplication and division operations.

Possible Optimization in ArcadeTokenDistributor.sol

This contract uses separate **boolean** flags to track if each **distribution** has been sent. These flags could be combined into a single `uint256` (or `uint8` for even more gas savings) with each bit representing a different `distribution`. This would save gas by reducing the number of storage slots used.

Before Optimization

After Optimization:

```
// Combine distribution flags into a single uint8
uint8 public distributionsSent;

function toTreasury(address _treasury) external onlyOwner {
    // Check if the treasury distribution has not been sent (fir
    // and if the _treasury address is not zero
    require((distributionsSent & 1 == 0) && _treasury != address

    // Set the first bit of distributionsSent to 1 (indicating t
    distributionsSent |= 1;

    // Transfer the treasury amount of tokens to the _treasury a
    arcadeToken.safeTransfer(_treasury, treasuryAmount);

    // Emit a Distribute event
    emit Distribute(address(arcadeToken), _treasury, treasuryAmo
}
```

Estimated gas saved:
This optimization could save around 5000 to 20000 gas per transaction, depending on the number of distributions. This is because the `SSTORE` opcode, which is used to update storage, is one of the most expensive operations in terms of gas cost. By reducing the number of storage slots used, we can save a significant amount of gas.

Possible Optimizations in ReputationBadge.sol

Possible Optimization 1

In the **mint** function, the **mintPrice** is calculated by multiplying **mintPrices[tokenId]** with amount. However, this calculation is performed before the checks for

claimExpiration, msg.value < mintPrice, and _verifyClaim. If any of these checks fail, the mintPrice calculation would have been performed unnecessarily. Moving the mintPrice calculation after these checks could save gas in the case where any of these checks fail.

After Optimization:

```
uint48 claimExpiration = claimExpirations[tokenId];

if (block.timestamp > claimExpiration) revert RB_ClaimingExpirec
if (!_verifyClaim(recipient, tokenId, totalClaimable, merkleProc
if (amountClaimed[recipient][tokenId] + amount > totalClaimable)
    revert RB_InvalidClaimAmount(amount, totalClaimable);
}

uint256 mintPrice = mintPrices[tokenId] * amount;
if (msg.value < mintPrice) revert RB_InvalidMintFee(mintPrice, m

// increment amount claimed
amountClaimed[recipient][tokenId] += amount;

// mint to recipient
_mint(recipient, tokenId, amount, "");
```

Estimated gas saved:
This optimization could save around 200 to 500 gas per transaction, depending on the gas cost of the multiplication operation.

Possible Optimization 2

The publishRoots function uses a loop to update claimRoots, claimExpirations, and mintPrices for each tokenId. However, the tokenId is not checked for uniqueness, which means that the same tokenId could be updated multiple times in a single transaction, wasting gas. Adding a check to ensure that each tokenId is unique could save gas.

After Optimization:

```
function publishRoots(ClaimData[] calldata _claimData) external
    if (_claimData.length == 0) revert RB_NoClaimData();
```

```
        if (_claimData.length > 50) revert RB_ArrayTooLarge();

        for (uint256 i = 0; i < _claimData.length; i++) {
            // uniqueness check
            if (claimRoots[_claimData[i].tokenId] != bytes32(0)) rev

            // expiration check
            if (_claimData[i].claimExpiration <= block.timestamp) {
                revert RB_InvalidExpiration(_claimData[i].claimRoot,
            }

            claimRoots[_claimData[i].tokenId] = _claimData[i].claimR
            claimExpirations[_claimData[i].tokenId] = _claimData[i].
            mintPrices[_claimData[i].tokenId] = _claimData[i].mintPr
        }

        emit RootsPublished(_claimData);
    }
```

Estimated gas saved:
This optimization could save around 5000 to 10000 gas per transaction, depending on the number of `tokenIds` that are not unique. This is because the `SSTORE` opcode, which is used to update storage, is one of the most expensive operations in terms of gas cost. By reducing the number of unnecessary `SSTORE` operations, we can save a significant amount of gas.

PowVT (Arcade.xyz) acknowledged and commented:

> A couple of these optimizations will cause security issues. Marking as acknowledged.

Audit Analysis

For this audit, 14 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The **report highlighted below** by **Sathish9098** received the top score from the judge.

The following wardens also submitted reports: **DadeKuma**, **kutugu**, **ktg**, **peanuts**, **3agle**, **Udsen**, **0x3b**, **foxb868**, **K42**, **0xnev**, **BugBusters**, **oakcobalt**, *and*

squeaky_cactus.

Audit Analysis Summary

List	Head	Details
1	Overview of Arcade.xyz platform	Overview of the key components and features of Arcade.xyz
2	My Thoughts	My thoughts about future of the this protocol
3	Audit approach	Process and steps I followed
4	Learnings	Learnings from this protocol
5	Possible Systemic Risks	The possible systemic risks based on my analysis
6	Code Commentary	Suggestions for existing code base
7	Centralization risks	Concerns associated with centralized systems
8	Gas Optimizations	Details about my gas optimizations findings and gas savings
9	Possible Risks as per my analysis	Possible risks
10	Time spent on analysis	The overall time spent for this report

Overview of Arcade.xyz platform

Arcade.xyz is a platform for autonomous borrowing, lending, and escrow of NFT collateral on EVM blockchains. This governance system is built on the `Council Framework`.

Arcade governance is classified into 4 types:

- `Voting Vaults` : Each voting vault contract is a separate deployment, which handles its deposits and vote-counting mechanisms for those deposits.
- `Core Voting` : These contracts can be used to submit and vote on proposed governance transactions. Core voting contracts may either administrate the protocol directly or may be intermediated by a `Timelock` contract.
- `Token` : The ERC20 governance token, along with contracts required for initial deployment and distribution of the token (airdrop contract, initial distributor

contract).

- `NFT` : The ERC1155 token contract along with its `tokenURI` descriptor contract. The ERC1155 token is used in governance to give a multiplier to a user's voting power.

My Thoughts

Arcade.xyz is a platform to change the future in the following areas:

- It provides a way for users to have a say in how the protocol is governed. This is important because it gives users more control over their assets and ensures that the protocol is aligned with their interests.
- The Arcade governance system is designed to be scalable. This means that it can be used to govern large and complex DeFi protocols. This is important because it will allow DeFi protocols to grow and evolve without sacrificing decentralization.
- The Arcade governance system is secure. This is because it is built on the Council Framework, which is a battle-tested governance system. This means that the Arcade governance system is less likely to be exploited by malicious actors

Audit approach

I followed the steps below while analyzing and auditing the code base:

1. Read the audit Readme.md and took the required notes.
    - Arcade.xyz platform uses
    - Inheritance
    - NFT
    - Timelock Functions
    - ERC20
    - The test coverage is 99%
2. Analyzed the overall codebase on iterations very fast.
3. Study of documentation to understand each contract's purpose, its functionality, how it is connected with other contracts, etc.

4. Then I read old audits and known findings. Then went through the bot race findings.
5. Then I setup my testing environment things. Run the tests to checks all test passed.
6. Finally, I started with auditing the codebase in-depth. I started understanding the code line by line and took the necessary notes to ask some questions to sponsors.

Stages of audit

- `The first stage of the audit` : During the initial stage of the audit for the Arcade.xyz platform, the primary focus was on analyzing gas usage and quality assurance (QA) aspects. This phase of the audit aimed to ensure the efficiency of gas consumption and verify the robustness of the platform.
- `The second stage of the audit` : In the second stage of the audit for the Arcade.xyz platform, the focus shifted towards understanding the protocol usage in more detail. This involved identifying and analyzing the important contracts and functions within the system. By examining these key components of the audit, I aimed to gain a comprehensive understanding of the protocol's functionality and potential risks.
- `The third stage of the audit` : During the third stage of the audit for the Arcade.xyz platform, the focus was on thoroughly examining and marking any doubtful or vulnerable areas within the protocol. This stage involved conducting comprehensive vulnerability assessments and identifying potential weaknesses in the system. Found `60-70` , `vulnerable` , and `weakness` code parts all marked with `@audit tags` .
- `The fourth stage of the audit` : During the fourth stage of the audit for the Arcade.xyz platform, a comprehensive analysis and testing of the previously identified doubtful and vulnerable areas were conducted. This stage involved diving deeper into these areas, performing in-depth examinations, and subjecting them to rigorous testing; including fuzzing with various inputs. Finally concluded findings after all research and tests. Then I reported to C4 with proper formats.

Learnings

Arcade.xyz's governance system, we can understand several key learnings:

- `Decentralized Governance` : Arcade.xyz allows token holders to vote on platform decisions, ensuring that no single entity or group has complete control. It's a democratic way of making choices for the platform.
- `Voting Vaults and Delegation` : Users can deposit their voting tokens in separate vaults and also delegate their voting power to someone they trust, making it easier for more people to participate in voting.
- `ERC20 and ERC1155 Tokens` : Arcade.xyz uses two types of tokens - ERC20 for basic voting and ERC1155 for more advanced governance features, like giving some tokens more voting influence.
- `Council Framework` : The governance system follows a specific Council Framework, which outlines rules and guidelines for how decisions are made.
- `Protocol Administrations and Timelock` : Proposals can directly impact the platform or go through a `Timelock` first, adding a delay to allow community review and transparency.
- `Autonomy and Smart Contracts` : Once set up, the governance system operates autonomously using smart contracts, without needing central control.
- `Transparency and Openness` : Arcade.xyz is transparent about how things work, providing clear technical details for users to understand and participate effectively.
- `NFT Integration` : Non-fungible tokens (NFTs) can enhance voting power, incentivizing community involvement and participation.

Possible Systemic Risks

- **Centralization Risk**: Voting power could concentrate among a few, leading to centralization.
- **Delegate Control Concerns**: Delegation may reduce voter engagement and representativeness.
- **Low Participation**: Low turnout may undermine governance legitimacy.
- **Ineffective Proposals**: Poorly designed proposals may hinder progress.
- **Voting Manipulation**: Malicious actors may try to influence outcomes.
- **Token Concentration**: Unequal token distribution may skew decisions.
- **NFT Impact Uncertainty**: ERC1155 NFTs' effects may have unintended consequences.

- **Technical Vulnerabilities**: Smart contract bugs could compromise governance.
- **Community Disagreements**: Conflicts may impede decision-making.
- **Governance Rule Updates**: Changing rules may be challenging and require consensus.

Code Commentary

- Instead of using bytes32 constants for role identifiers, consider using an enum for better readability and code organization.
- Functions like `smallSpend`, `mediumSpend`, and `largeSpend` have similar logic. Consider refactoring the code to combine the common logic into a single internal function and call it from each respective function. This can reduce code duplication.
- For events like `TreasuryTransfer` and `TreasuryApproval`, consider emitting only relevant data instead of emitting the entire amount and destination/spender.
- The “immutable” variable should be in upper case.
- Instead of using error messages in revert statements, consider using an enum to define error codes. This can make error handling more structured and easier to understand.
- Ensure that error messages are clear and informative to help developers and users understand contract behavior and potential issues.
- The contract could benefit from more detailed comments and documentation to explain the purpose of various functions, their expected behavior, and any potential risks or constraints. This would make it easier for developers to understand and interact with the contract.
- Consider reordering the modifiers in the function declarations to improve readability. For example, move the `onlyManager` modifier to be the last one, as it’s the most specific and should be applied after other access control checks.
- Consider using a struct to represent grant data instead of individual storage variables. This can make the code cleaner and easier to manage.
- Instead of importing the entire ERC20 contract, consider using an ERC20 interface with only the necessary functions. This helps to reduce contract deployment costs.

- The contract uses a mix of `camelCase` and `snake_case` for function and variable names. It's advisable to use a consistent naming convention throughout the contract to enhance readability.
- There are some duplicated code segments (e.g., handling delegation and voting power updates) that could be abstracted into separate helper functions to improve code readability and maintainability.
- While the contract has some comments, additional documentation could be provided to clarify the purpose and behavior of certain functions and variables.
- In functions where multiple events are emitted, consider consolidating them into a single emit statement for better gas efficiency.

Centralization risks

A single point of failure is not acceptable for this project. Centrality risk is high in the project, the role of `onlyOwner` detailed below has very critical and important powers.

Project and funds may be compromised by a malicious or stolen private key `onlyOwner msg.sender`:

```
File: contracts/nft/BadgeDescriptor.sol

57:     function setBaseURI(string memory newBaseURI) external c
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/nft/BadgeDescriptor.sol#L57

```
File: contracts/token/ArcadeAirdrop.sol

62:     function reclaim(address destination) external onlyOwner

75:     function setMerkleRoot(bytes32 _merkleRoot) external onl
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/token/ArcadeAirdrop.sol#L62

```
File: contracts/token/ArcadeTokenDistributor.sol

73:     function toTreasury(address _treasury) external onlyOwne

89:     function toDevPartner(address _devPartner) external only

105:     function toCommunityRewards(address _communityRewards)

121:     function toCommunityAirdrop(address _communityAirdrop)

138:     function toTeamVesting(address _vestingTeam) external c

155:     function toPartnerVesting(address _vestingPartner) exte

174:     function setToken(IArcadeToken _arcadeToken) external c
```

https://github.com/code-423n4/2023-07-arcade/blob/main/contracts/token/ArcadeTokenDistributor.sol#L73

Gas Optimizations

During the audit of the smart contracts on `Arcade.xyz's` governance platform, several key gas optimization issues were identified. These issues could potentially lead to increased transaction costs and inefficiencies on the Ethereum network. One of the notable findings was related to state variables, where it was observed that they can be packed together to utilize fewer storage slots. By organizing related variables in this way, the number of storage operations, such as `SSTORE` and `SLOAD` opcodes, can be reduced, resulting in significant gas savings.

Another prevalent optimization opportunity involved caching state variables in stack variables instead of repeatedly re-reading them from storage. This approach can eliminate redundant storage operations, leading to improved gas efficiency. Utilizing opcodes like `SWAP`, `DUP`, and `MLOAD` efficiently allows for cost-effective access to state variables during contract execution.

In addition, a gas-saving technique was found in arithmetic operations on state variables. It was observed that using the `=` operator rather than `+=` and `-=` could result in reduced gas consumption for the `ADD`, `SUB`, and `MSTORE` opcodes. Moving `IF` statements or `require()` functions that check input arguments to the

top of the functions was another recommendation to avoid unnecessary code execution; thereby, saving gas. This optimization takes advantage of the `JUMP` and `JUMPI` opcodes, which control the flow of execution within the contract.

Furthermore, multiple accesses of mappings or arrays were identified as potential areas for improvement. By employing local variable caches, redundant read operations from storage can be minimized, reducing gas costs. The relevant opcodes in this context include `MLOAD`, `MSTORE`, and `SLOAD`.

Moreover, it was noticed that emitting state variables when stack variables are available could lead to unnecessary gas consumption. Emitting state variables involves storage operations, while stack variables are more gas-efficient. The `LOG` opcode is used for emitting event logs.

Lastly, optimizing function parameters by using `calldata` instead of `memory` can save gas. The `CALLDATASIZE` opcode is used to access the size of the input data, and `CALLDATALOAD` is used to retrieve specific function parameters from the input.

https://github.com/code-423n4/2023-07-arcade-findings/issues/151

Possible Risks as per my analysis

- The `batchCalls` function allows executing multiple low-level calls in a single transaction. If a large number of calls are included or if any of the calls fail, the entire transaction will be reverted, leading to gas wastage. This can result in higher transaction costs and failed transactions due to gas limits.
- The `smallSpend`, `mediumSpend`, `largeSpend`, `approveSmallSpend`, `approveMediumSpend`, and `approveLargeSpend` functions do not perform input validation to check whether the `amount` parameter exceeds the token balance or allowance. This could lead to unintended spending or approval of more tokens than the treasury holds or intends to allow.
- The `_spend` and `_approve` functions enforce a per-block spending/approval limit (`limit`) to avoid excessive spending. However, there is no mechanism to reset the `blockExpenditure` storage variable when a new block starts, which could lead to inconsistencies if the contract is active across multiple blocks.

- In the `_spend` function, if the destination is a contract without a `receive function`, the transfer will fail, resulting in a `loss of funds`. The function should include checks to handle such cases appropriately.
- Hardcoded `cooldown period` may cause problems in the future.
- Lack of same value input control in critical `setMinter()` functions.
- Use `safeInceaseAllowance()` / `safeDecreaseAllowance()` instead of `approve()` or deprecated `safeApprove()` functions.
- Don't use `payable(address).tranfer()` to avoid unexpected fund loss.
- The function `addGrantAndDelegate` allows the manager to create a new grant without checking if the grant recipient already has an active grant. If a user already has an active grant, this function will overwrite the existing grant, which might lead to unintended behavior. It would be better to check if the recipient already has a grant and handle this situation accordingly.
- In the `_getWithdrawableAmount` function, there are multiple arithmetic calculations involving subtraction, addition, and multiplication. Ensure that these calculations do not result in underflow or overflow situations, which could lead to unintended behavior or vulnerabilities.
- The contract uses a custom state management system with the Storage library, which might make it harder to understand for developers not familiar with this approach. Consider using standardized storage patterns to improve code readability and maintainability.
- Some functions, such as `_lockNft`, `_lockTokens`, and `_grantVotingPower`, are marked as internal but are not strictly necessary to be internal. It's essential to assess whether these functions need to be called from other contracts or external interactions.

Time spent on analysis

15 Hours

PowVT (Arcade.xyz) acknowledged and commented:

> Mostly duplicate findings as previous issues. Everything else is expected behavior. Will not make any fixes from this.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit 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