SpartaDEX - Launchpad Audit Report

Prepared by bytes032

Contents

1 About bytes032

bytes032 is an independent smart contract security researcher.

His knack for identifying smart contract security vulnerabilities in a range of protocols is more than a skill; it's a passion. Committed to enhancing the blockchain ecosystem, he invests his time and energy into thorough security research and reviews. Want to know more or collaborate? bytes's always up for a chat about all things security.

Feel free to reach out on X.

2 Protocol Summary

SpartaDEX is a combination of real-time strategy game set in the realities of ancient Greece and a decentralized cryptocurrency exchange. They call it a **gamified DEX**. The main goal is to provide the exchange with user engagement known from video games, which builds loyalty and consistency in providing liquidity.

This audit is in regards to their 2nd product, called SpartaPad, which gives huge utility to NFT holders and $SPARTA tokens.

It's a launchpad on which the users will be able to grab an allocation in top Arbitrum projects.

It gives a huge utility to both Revealed Spartans NFTs and SPARTA token holders. Each fundraising will have a dedicated, pre-public round only for NFT holders and WL, where each of the NFTs will translate into a part in the allocation.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of {X} according to the specific commit. Any modifications to the code will require a new security review.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Overview

Project	SpartaDEX - Launchpad
Repository	sdex-smart-contracts
Commit	44ae03a132f4...
Date	May 2023

Issues Found

Severity	Count
Critical Risk	1
High Risk	1
Medium Risk	2
Low Risk	2
Informational	0
Gas Optimizations	0
Total Issues	**6**

Summary of Findings

Title	Status
[C-1] Tokens can be used multiple times for the same fundraiser	Resolved
[H-1] release() doesnt return excess fees	Resolved
[M-1] Inaccurate timing mechanism for daily token claim	Resolved
[M-2] Possibility of overwriting the current vesting	Resolved
[L-1] Ensure that nftStartTime > openStartTime	Resolved
[L-2] Lack of restriction on vesting start before fundraising ends	Resolved

5 Findings

5.1 Critical Risk

[C-1] Tokens can be used multiple times for the same fundraiser

Context: Fundraiser.sol

Impact: This vulnerability can lead to substantial imbalances in token allocation among users, providing an unfair advantage to those who exploit it.

Description: The function `calculateMaxAllocation` computes the highest allocation a user can obtain, predicated on their deposit amount and the quantity of their valid and invalid NFTs.

The base allocation per wallet is the first factor in this calculation: if a user's deposit equals or falls short of this figure, the maximum allocation derives from this base plus an additional amount proportionate to the number of valid NFTs.

For deposits exceeding the base allocation, the function employs distinct thresholds, taking into account both valid and invalid NFTs.

When the function is first invoked and the user's deposit is less than or equal to the `baseAllocationPerWallet`, the maximum allocation calculation is as follows:

$$allocation = basePerWallet + (numOfValidNfts * allocationPerNft)$$

Should the deposit exceed the base allocation, the function calculates two thresholds:

```
uint256 threshold1 = allocationConfig.baseAllocationPerWallet.add(allocationConfig.nftTicketAllocation);

uint256 threshold2 = allocationConfig.baseAllocationPerWallet.add(allocationConfig.nftTicketAllocation.mul(2));
```

Threshold 1 is supposed to detect if the user deposited 1 nft. Threshold 2 is supposed to detect i the user deposited 2 nft's.

Threshold 1 and threshold 2 correspond to deposits for one and two NFTs, respectively. The function subsequently uses these thresholds to determine the allowable NFTs for a user.

```
        // threshold for 1 nft
        if (userDeposit <= threshold1) {
            allowedNfts = invalidNftCount >= 1 ? 1 + validNftCount : validNftCount;

        // threshold for 2 nft's
        } else if (userDeposit <= threshold2) {
            allowedNfts = invalidNftCount >= 2 ? 2 + validNftCount : invalidNftCount == 1 ? validNftCount + 1 :
            ↪  validNftCount;

        // threshold for 3 nft's
        } else {
            allowedNfts = invalidNftCount >= 3 ? 3 : validNftCount + invalidNftCount;
        }
```

If the **current** deposit of the user is below or equal threshold 1

- If the user has one or more invalid NFTs (`invalidNftCount >= 1`), they can have a total number of NFTs that equals their number of valid ones plus one more (`1 + validNftCount`).
- If the user doesn't have any invalid NFTs, then they're only allowed to have as many NFTs as they have valid ones (`validNftCount`), though the max amount of NFT's passed can be no more than 3.

If the **current** deposit of the user is below or equal threshold 2:

- If the user has two or more invalid NFTs (`invalidNftCount >= 2`), they can have a total number of NFTs equal to their number of valid ones (`validNftCount`) plus two.
- If the user has exactly one invalid NFT (`invalidNftCount == 1`), they can have a total number of NFTs equal to their number of valid ones plus one.
- If the user doesn't have any invalid NFTs, they can have only the number of valid NFTs they already have (`validNftCount`).

If none of the requirements above are met:

- If the user has provided three or more invalid NFT's, his **allowed** NFT's equal 3
- Otherwise, its the sum of the invalidNftCount and validNftCount

The vulnerability here stems from two things:

1. In the calculation logic, a whitelisted user who deposits an amount that surpasses both `threshold1` and `threshold2` will automatically fall into the third condition, where the allocation calculation does not factor in whether the NFTs are valid or not.
2. As a consequence, users can get NFT's from another user which are considered as invalid, but the function will still count them as valid, because it doesn't expect the users to have balance above the threshold.

The validity and invalidity of NFTs are determined by the function below, enabling users to transfer used (thus invalid) tokens to others, who can then treat them as their own.

```
function countValidAndInvalidNfts(address userAddress, uint256[] memory nftIds) private view returns (uint256
↪   validNftCount, uint256 invalidNftCount) {

    for (uint256 i = 0; i < nftIds.length; i++) {
        uint256 nftId = nftIds[i];

        if (IERC721(nftToken).ownerOf(nftId) == userAddress) {
            if (!usedNftId[nftId]) {
                validNftCount++;
            } else {
                invalidNftCount++;
            }
        }
    }
}
```

Consider a scenario involving two users, Amelie and Noemi. Assume the following:

```
uint256 baseAllocationPerWallet = 50e18;
uint256 maxTotalAllocation = 200e18;
uint256 nftTicketAllocation = 1e18;
uint256 whitelistAllocation = 10e18;
```

1. Amelie possesses 3 NFTs and 100 tokens.
2. Amelie invokes `depositWithNFT(63e18, amelieTokenIds)` with her three NFTs, receiving the maximum allocation permissible by the system.
3. Amelie's tokens are marked as used, rendering them invalid for future use.
4. Noemi, lacking NFTs, resorts to a standard deposit of up to 60e18.
5. Cunningly, Noemi requests temporary custody of Amelie's used tokens.
6. Amelie, obliging her friend, transfers her used tokens to Noemi.
7. Noemi then invokes `depositWithNFT(3e18, amelieTokenIds)`, attaining the maximum allocation permitted by the system, despite not actually owning any NFTs.

The following Proof of Concept demonstrates this vulnerability in action:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.15;
import "forge-std/Test.sol";
import "openzeppelin/contracts/utils/math/SafeMath.sol";
import "openzeppelin/contracts/token/ERC20/IERC20.sol";
import "openzeppelin/contracts/token/ERC20/ERC20.sol";
import "openzeppelin/contracts/access/Ownable.sol";
import "openzeppelin/contracts/token/ERC721/IERC721.sol";
import "openzeppelin/contracts/token/ERC721/ERC721.sol";
import "openzeppelin/contracts/security/Pausable.sol";
import "openzeppelin/contracts/utils/cryptography/MerkleProof.sol";
import "openzeppelin/contracts/access/Ownable.sol";
import "forge-std/Test.sol";

contract MockNFT is ERC721 {
    constructor() ERC721("MockNFT", "MNFT") {}

    function mint(uint256 tokenId) external {
        _mint(msg.sender, tokenId);
    }
}

contract MockERC20 is ERC20 {
    constructor() ERC20("MockERC20", "MERC20") {}

    function mint(uint256 amount) external {
        _mint(msg.sender, amount);
    }
}


contract Fundraiser is Pausable, Ownable {
    using SafeMath for uint256;
    struct TokenConfig {
        address depositToken;
        address factory;
    }

    struct AllocationConfig {
        uint256 nftTicketAllocation;
        uint256 baseAllocationPerWallet;
        uint256 maxTotalAllocation;
        uint256 rate;
    }

    struct TimeConfig {
        uint256 nftStartTime;
        uint256 openStartTime;
        uint256 endTime;
        uint256 distributionStartTime;
    }

    TokenConfig public tokenConfig;
    AllocationConfig public allocationConfig;
    TimeConfig public timeConfig;
    address public nftToken;
    uint256 public whitelistedAllocation;
    uint256 public constant ALLOCATION_DIVIDER = 10000;


    mapping(address => uint256) public depositedAmount;
```

```
uint256 public totalDeposited;
mapping(uint256 => bool) public usedNftId;

event Deposit(address indexed user, uint256 amount);
event Withdraw(address indexed user, uint256 amount);
event Claim(address indexed user, uint256 amount);
event VestingDeployed(address vestingAddress);
event RegisterWhitelist(bytes32 merkleRoot, uint256 whitelistedAllocation);

modifier canDeposit(uint256 _amount) {
    require(_amount > 0, "Fundraiser: deposit amount must be greater than 0");
    require(block.timestamp <= timeConfig.endTime, "Fundraising has already ended");
    require(totalDeposited.add(_amount) <= allocationConfig.maxTotalAllocation, "Max total allocation reached");
    _;
}

error TransferFailed();

constructor(
    address _depositToken,
    uint256 _baseAllocationPerWallet,
    uint256 _maxTotalAllocation,
    uint256 _nftTicketAllocation,
    uint256 _nftFundraiseStartTime,
    uint256 _openFundraiseStartTime,
    uint256 _fundraiseEndTime,
    address _owner,
    address _nftToken
) {
    tokenConfig.depositToken = _depositToken;
    allocationConfig.baseAllocationPerWallet = _baseAllocationPerWallet;
    allocationConfig.maxTotalAllocation = _maxTotalAllocation;
    allocationConfig.nftTicketAllocation = _nftTicketAllocation;
    timeConfig.nftStartTime = _nftFundraiseStartTime;
    timeConfig.openStartTime = _openFundraiseStartTime;
    timeConfig.endTime = _fundraiseEndTime;
    nftToken = _nftToken;
    _transferOwnership(_owner);
}


function depositWithWhitelist(uint256 _amount) external whenNotPaused canDeposit(_amount) {
    require(block.timestamp >= timeConfig.nftStartTime, "Fundraising has not started yet");

    // require(MerkleProof.verify(proof_, merkleRoot, keccak256(abi.encodePacked(msg.sender))), "User is not
    ↪  whitelisted.");

    uint256 userDeposit = depositedAmount[msg.sender];

    require(_amount.add(userDeposit) <= whitelistedAllocation.add(allocationConfig.baseAllocationPerWallet),
    ↪  "Whitelist max allocation overflow");
    if (
        !IERC20(tokenConfig.depositToken).transferFrom(msg.sender, address(this), _amount)
    ) {
        revert TransferFailed();
    }
    depositedAmount[msg.sender] = depositedAmount[msg.sender].add(_amount);
    totalDeposited = totalDeposited.add(_amount);

    emit Deposit(msg.sender, _amount);
}
```

```
function depositWithWhitelistAndNft(uint256 _amount, uint256[] memory nftIds) external whenNotPaused
↪   canDeposit(_amount) {
    require(block.timestamp >= timeConfig.nftStartTime, "Fundraising has not started yet");

    require(nftIds.length <= 3, "Max nftIds is 3");
    require(nftIds.length >= 1, "Min nftIds is 1");

    // require(MerkleProof.verify(proof_, merkleRoot, keccak256(abi.encodePacked(msg.sender))), "User is not
    ↪   whitelisted.");

    uint256 userDeposit = depositedAmount[msg.sender];

    (uint256 validNftCount, uint256 invalidNftCount) = countValidAndInvalidNfts(msg.sender, nftIds);
    uint256 nftMaxAllocation = calculateMaxAllocation(userDeposit, validNftCount, invalidNftCount);

    require(userDeposit.add(_amount) <= nftMaxAllocation.add(whitelistedAllocation), "Max whitelisted and nft
    ↪   allocation overflow");

    for (uint256 i = 0; i < nftIds.length; i++) {
        uint256 nftId = nftIds[i];

        if (IERC721(nftToken).ownerOf(nftId) == msg.sender && !usedNftId[nftId]) {
            usedNftId[nftId] = true;
        }
    }

    if (
        !IERC20(tokenConfig.depositToken).transferFrom(msg.sender, address(this), _amount)
    ) {
        revert TransferFailed();
    }

    depositedAmount[msg.sender] = depositedAmount[msg.sender].add(_amount);
    totalDeposited = totalDeposited.add(_amount);

    // emit Deposit(msg.sender, _amount);
}

function setWhitelist(uint256 _whitelistedAllocation) external onlyOwner {

    whitelistedAllocation = _whitelistedAllocation;
    // merkleRoot = _merkleRoot;
    // emit RegisterWhitelist(_whitelistedAllocation);
}

function useAllocationWithNfts(uint256[] memory nftIds, uint256 additionalDeposit) private {
    // Currently deposited amount
    uint256 userDeposit = depositedAmount[msg.sender];

    // validNftCount = number of valid nfts that are owned by the address and are not used yet
    // invalidNftCount = reverse of validNftCount
    (uint256 validNftCount, uint256 invalidNftCount) = countValidAndInvalidNfts(msg.sender, nftIds);
    uint256 maxAllocation = calculateMaxAllocation(userDeposit, validNftCount, invalidNftCount);

    require(userDeposit.add(additionalDeposit) <= maxAllocation, "Max allocation overflow");

    for (uint256 i = 0; i < nftIds.length; i++) {
        uint256 nftId = nftIds[i];
        // cannot share nftIds between users, because nftId is used to compare
        if (IERC721(nftToken).ownerOf(nftId) == msg.sender && !usedNftId[nftId]) {
            usedNftId[nftId] = true;
        }
```

```
    }
}

function countValidAndInvalidNfts(address userAddress, uint256[] memory nftIds) private view returns (uint256
↪   validNftCount, uint256 invalidNftCount) {

    for (uint256 i = 0; i < nftIds.length; i++) {
        uint256 nftId = nftIds[i];

        if (IERC721(nftToken).ownerOf(nftId) == userAddress) {
            if (!usedNftId[nftId]) {
                validNftCount++;
            } else {
                invalidNftCount++;
            }
        }
    }
}

// if you hold NFT's, you can deposit beyond the limit of the base allocation per wallet
function calculateMaxAllocation(uint256 userDeposit, uint256 validNftCount, uint256 invalidNftCount) private view
↪   returns (uint256) {
    uint256 maxAllocation;
    uint256 allowedNfts;

    if (userDeposit <= allocationConfig.baseAllocationPerWallet) {
        // @audit-info the user has exercised on 100% his "erc20 deposit allocation"
        // or the user has not deposited at all yet

        // the maximum allocation is based on the
        // maximum per wallet + the valid nft's * the allocation per nft
        maxAllocation =
        ↪   allocationConfig.baseAllocationPerWallet.add(validNftCount.mul(allocationConfig.nftTicketAllocation));
    } else {

        // @audit-info when trying to deposit more than the base allocation per wallet

        uint256 threshold1 = allocationConfig.baseAllocationPerWallet.add(allocationConfig.nftTicketAllocation);

       uint256 threshold2 = allocationConfig.baseAllocationPerWallet.add(allocationConfig.nftTicketAllocation.mul(2));

        // this here practically means that if a user has already deposited with 1 valid nft
        // on the next transaction he has to provide 1 invalid and 2 valids to get the maximum allocation

        // threshold for 1 nft
        if (userDeposit <= threshold1) {
            allowedNfts = invalidNftCount >= 1 ? 1 + validNftCount : validNftCount;

        // threshold for 2 nft's
        } else if (userDeposit <= threshold2) {
            allowedNfts = invalidNftCount >= 2 ? 2 + validNftCount : invalidNftCount == 1 ? validNftCount + 1 :
            ↪   validNftCount;

        // threshold for 3 nft's
        } else {
            allowedNfts = invalidNftCount >= 3 ? 3 : validNftCount + invalidNftCount;
        }

        // the maximum allocation based on the
        // maximum per wallet + the nft threshold * the allocation per nft

        // if the user doesn't provide at least 1 invalid nft, the max allocation
```

```
            // equals the base allocation per wallet
            maxAllocation =
            ↪  allocationConfig.baseAllocationPerWallet.add(allowedNfts.mul(allocationConfig.nftTicketAllocation));
        }

        // total deposited by everybody so far + the maximum allocation for this deposit
        uint256 currentTotal = totalDeposited.add(maxAllocation);
        if (currentTotal > allocationConfig.maxTotalAllocation) {

            // if the current total is bigger, the maximum allocation is
            // the difference between the max and the total deposited
            return allocationConfig.maxTotalAllocation.sub(totalDeposited);
        }

        return maxAllocation;
    }

    function checkUsed(uint256[] memory nftIds) external view returns (uint256[] memory) {
        uint256[] memory validNfts = new uint256[](nftIds.length);

        uint256 validCount = 0;
        for (uint256 i = 0; i < nftIds.length; i++) {
            uint256 nftId = nftIds[i];
            if (!usedNftId[nftId]) {
                validNfts[validCount] = nftId;
                validCount++;
            }
        }

        uint256[] memory result = new uint256[](validCount);
        for (uint256 i = 0; i < validCount; i++) {
            result[i] = validNfts[i];
        }

        return result;
    }
}



contract Test30 is Test {
  using SafeMath for uint256;
  Fundraiser fundraiser;
  MockERC20 token;
  MockNFT nft;

  address amelie = makeAddr("amelie");
  address noemi = makeAddr("noemi");
  uint256 baseAllocationPerWallet = 50e18;
  uint256 maxTotalAllocation = 200e18;
  uint256 nftTicketAllocation = 1e18;
  uint256 whitelistAllocation = 10e18;

  function setUp() public {
    vm.createSelectFork("https://eth.llamarpc.com/rpc/01GYVK2PRR9PG6E3KP7P8DSEFX");
    token = new MockERC20();
    nft = new MockNFT();

    fundraiser = new Fundraiser(address(token), baseAllocationPerWallet, maxTotalAllocation, nftTicketAllocation,
    ↪  block.timestamp, block.timestamp, block.timestamp + 1 days, address(this), address(nft));
    fundraiser.setWhitelist(whitelistAllocation);
  }
```

```
  function test_whiteListedCanUseInvalidNFT() external {

    // Amelie has 3 NFT's and 100 tokens
    vm.startPrank(amelie);
    uint256[] memory amelieTokenIds = new uint256[](3);
    amelieTokenIds[0] = 1;
    amelieTokenIds[1] = 2;
    amelieTokenIds[2] = 3;

    nft.mint(1);
    nft.mint(2);
    nft.mint(3);
    token.mint(100e18);

    token.approve(address(fundraiser), 100e18);
    fundraiser.depositWithWhitelistAndNft(63e18, amelieTokenIds);
    assertEq(fundraiser.depositedAmount(amelie) == 63e18, true);
    vm.stopPrank();

    vm.startPrank(noemi);

    token.mint(100e18);
    token.approve(address(fundraiser), 100e18);
    fundraiser.depositWithWhitelist(60e18);
    assertEq(fundraiser.depositedAmount(noemi) == 60e18, true);
    vm.stopPrank();


    // Noemi borrows tokens from Amelie
    vm.startPrank(amelie);

    nft.transferFrom(amelie, noemi, 1);
    nft.transferFrom(amelie, noemi, 2);
    nft.transferFrom(amelie, noemi, 3);

    vm.stopPrank();

    vm.startPrank(noemi);

    fundraiser.depositWithWhitelistAndNft(3e18, amelieTokenIds);

    assertEq(fundraiser.depositedAmount(noemi) == 63e18, true);
  }
}
```

Recommendation: To mitigate this vulnerability, it is advised to implement the following measures:

1. Adjust the system to account for any changes to token ownership during the time of the fundraising.
2. The `calculateMaxAllocation` function should be refactored so that the allocation is computed based solely on valid tokens. This means the system should exclude invalid tokens from the calculation, ensuring that only NFTs owned by the user and not marked as used are considered in the allocation. This can prevent users from taking advantage of invalid or used NFTs to gain an unfair share of allocation.

5.2 High Risk

[H-1] release() doesnt return excess fees

Context: Vesting.sol

Impact: The impact is primarily financial, as the current implementation does not return any excess Ether (`msg.value`) paid beyond the required `ethFee`. Users who pay more than the required fee will not receive their excess payment back, leading to an unintended loss of funds.

Description: In the `release()` function, the contract requires the users to pay a fee (`ethFee`) to release their tokens.

However, it does not handle the situation where the user pays more (`msg.value`) than the required fee. Currently, if `msg.value > ethFee`, the contract simply accepts the payment without refunding the difference to the user.

Consider the following scenario:

1. The `ethFee` is 0.1e18
2. A user sends 0.2e18 as `msg.value`
3. The fee is correctly charged at 0.1e18, but the user is not refunded the excess of 0.1e18

As a result, users may unknowingly overpay and lose Ether when using this function.

```
function release() external payable {
    uint256 unreleased = releasableAmount(msg.sender);
    require(unreleased > 0, "No tokens to release");
    require(msg.value >= ethFee, "Insufficient fee: the required fee must be covered");

    tokensReleased[msg.sender] = tokensReleased[msg.sender].add(unreleased);
    if (
        !config.token.transfer(msg.sender, unreleased)
    ) {
        revert TransferFailed();
    }

    // @audit-issue if msg.value >= ethFee, the function doesn't refund the user with the difference
    emit TokensReleased(unreleased, msg.sender);
}
```

Recommendation: Return the difference to the user

```
+       uint256 dust = msg.value - ethFee;
+       (bool sent, ) = address(msg.sender).call{value: dust}("");
+       require(sent, "Failed to send Ether");
```

5.3 Medium Risk

[M-1] Inaccurate timing mechanism for daily token claim

Context: TokenFaucet.sol

Description: The `claimTokens` function within the smart contract was designed to allow users to claim a certain amount of tokens within a day.

However, the timing mechanism isn't truly 'daily' but rather depends on the last time the user claimed tokens.

This means if a user claims tokens multiple times within a 24-hour period, the time for the next 'daily' limit reset will be pushed further ahead, causing potential confusion and token loss.

```
function claimTokens(uint256 _amount) external {
    uint256 timeSinceLastClaim = block.timestamp - lastClaim[msg.sender];

    // If you haven't claimed in a day, your claimed tokens get reset.
    if (timeSinceLastClaim >= DAY_IN_SECONDS) {
        claimedTokens[msg.sender] = 0;
    }

    // Available tokens in a day
    uint256 availableTokens = dailyLimit - claimedTokens[msg.sender];

    // Claimable tokens since the last claim
    uint256 claimableTokens = (dailyLimit * timeSinceLastClaim) / DAY_IN_SECONDS;

    if (claimableTokens > availableTokens) {
        claimableTokens = availableTokens;
    }

    require(_amount <= claimableTokens, "Claim limit exceeded");

    claimedTokens[msg.sender] += _amount;

    // Set last claim to **now**
    lastClaim[msg.sender] = block.timestamp;

    // @audit-issue check the return value
    token.transfer(msg.sender, _amount);

    emit Claimed(msg.sender, _amount);
}
```

Consider the following scenario:

1. The daily token limit is 100e18.
2. Amelie claims 50e18. The `lastClaim[amelie]` variable is set to now, so in 24 hours she will be due **another** 100e18 from the **next daily limit**
3. 12 hours after the first claim, Amelie claims the other 50e18 from her **current** daily limit.
4. 12 more hours pass and her daily limit should have been reset, so on theory she should be due 100e18 more.
5. However, because she did a second claim 12h after her first one, her "daily" limit actually became a "day and a half" limit. Hence, if she attempts to claim now, the function will assume that she exceeded her claim limit

The issue arises from these lines in the contract:

```
    uint256 timeSinceLastClaim = block.timestamp - lastClaim[msg.sender];

    // If you haven't claimed in a day, your claimed tokens get reset.
    if (timeSinceLastClaim >= DAY_IN_SECONDS) {
        claimedTokens[msg.sender] = 0;
    }
```

Here's a runnable PoC that showcases the issue:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.15;
import "openzeppelin/contracts/token/ERC20/IERC20.sol";
import "openzeppelin/contracts/access/Ownable.sol";
import "forge-std/Test.sol";


contract DailyTokenClaim is Ownable {
    uint256 public dailyLimit;
    uint256 constant public DAY_IN_SECONDS = 86400;

    mapping(address => uint256) public lastClaim;
    mapping(address => uint256) public claimedTokens;

    event Claimed(address indexed claimer, uint256 amount);

    constructor(uint256 _dailyLimit) {
        dailyLimit = _dailyLimit;
    }

    function claimTokens(uint256 _amount) external {
        uint256 timeSinceLastClaim = block.timestamp - lastClaim[msg.sender];

        // If you haven't claimed in a day, your claimed tokens get reset.
        if (timeSinceLastClaim >= DAY_IN_SECONDS) {
            claimedTokens[msg.sender] = 0;
        }

        // Available tokens in a day
        uint256 availableTokens = dailyLimit - claimedTokens[msg.sender];

        // 1. 50 tokens left

        // Claimable tokens since the last claim
        uint256 claimableTokens = (dailyLimit * timeSinceLastClaim) / DAY_IN_SECONDS;
        console.log(availableTokens);
        console.log("claimableTokens: %s", claimableTokens);

        if (claimableTokens > availableTokens) {
            claimableTokens = availableTokens;
        }

        require(_amount <= claimableTokens, "Claim limit exceeded");

        claimedTokens[msg.sender] += _amount;

        // Set last claim to **now**
        lastClaim[msg.sender] = block.timestamp;

        console.log(_amount);

        emit Claimed(msg.sender, _amount);
```

```
    }

    function setDailyLimit(uint256 _newDailyLimit) external onlyOwner {
        dailyLimit = _newDailyLimit;
    }
}

contract Test28 is Test {
  DailyTokenClaim claimer;

  address amelie = makeAddr("amelie");

  function setUp() public {
    vm.createSelectFork("https://eth.llamarpc.com/rpc/01GYVK2PRR9PG6E3KP7P8DSEFX");
    claimer = new DailyTokenClaim(100e18);
  }

  function test_unfairAllocation() external {
    vm.startPrank(amelie);

    // Amelie claims 50 tokens
    claimer.claimTokens(50e18);

    // Limit resets in 1 day
    uint256 dailyLimitResetForAmelie = claimer.lastClaim(amelie) + 86400;

    // Move time forward 12 hours
    vm.warp(block.timestamp + 43200);

    // Amelie claims 50 tokens
    claimer.claimTokens(50e18);

    vm.warp(block.timestamp + 43201);
    claimer.claimTokens(50e18);

    // Limit is increased to 1 day + 12 hours.
  }
}
```

Recommendation: One feasible approach to resolving this issue would be to automatically transfer the maximum claimable amount to users, instead of permitting them to specify the amount themselves. This can prevent potential misunderstandings and token loss caused by the current claim system.

An alternative suggestion would involve modifying your claim mechanism to reset the amount of claimable tokens at a specific, fixed time (for instance, midnight UTC), rather than basing it on the timestamp of the last claim. This change would guarantee a uniform 24-hour day for all users, independent of their individual claim timings.

Finally, consider implementing a rolling window for claim limits. This system could involve recording each claim's timestamp and amount over a 24-hour period. Prior to each claim, you would tally all the tokens claimed within the previous 24 hours, eliminating any claims that fall outside this timeframe. Though this method might require additional computational resources, it would ensure accurate enforcement of the daily claim limit, irrespective of when claims are made.

[M-2] Possibility of overwriting the current vesting

Context: Fundraiser.sol

Impact: The vulnerability allows overwriting of an existing vesting contract when the `startVesting` function is called multiple times.

Description: The `startVesting` function creates a new instance of the `Vesting` contract, specifying the vesting start and end timestamps, token address, Ether fee, and owner's address. The address of the new `Vesting` contract instance is stored in `vestingAddress`.

```
function startVesting(uint256 _vestingStart, uint256 _vestingEnd, address _tokenAddress, uint256 _tokenAmount,
↪  uint256 _ethFee) external whenNotPaused onlyOwner {
    Vesting vesting = new Vesting(
        address(this),
        _vestingStart,
        _vestingEnd,
        _tokenAddress,
        _ethFee,
        owner()
    );

    vestingAddress = address(vesting);

    if (
        !IERC20(_tokenAddress).transferFrom(msg.sender, vestingAddress, _tokenAmount)
    ) {
        revert TransferFailed();
    }

    emit VestingDeployed(vestingAddress);
}
```

The issue here that even if a vesting is started and the function is called again, it will overwrite the previous vesting

Recommendation: To mitigate the vulnerability, it is recommended to add a check before creating a new vesting contract. If a vesting contract for a fundraiser already exists, the function should revert instead of creating a new contract. This ensures that only one vesting contract is created per fundraiser.

```
function startVesting(uint256 _vestingStart, uint256 _vestingEnd, address _tokenAddress, uint256 _tokenAmount,
↪  uint256 _ethFee) external whenNotPaused onlyOwner {
+     require(vestingAddress == address(0), "Vesting already exists")
}
```

5.4 Low Risk

[L-1] Ensure that nftStartTime > openStartTime

Context: Fundraiser.sol

Impact: The current implementation of the contract lacks a restriction that enforces the condition stated in the readme regarding the `nftStartTime` variable.

Description

According to the readme, the `nftStartTime` should represent a period where users holding NFTs are allowed to deposit before users who do not hold NFTs. However, as it stands, there are no checks in place to ensure this condition is met during the initialization of the contract.

However, currently there are no restrictions that actually force this when assigning the variables.

```
constructor(
        address _depositToken,
        uint256 _baseAllocationPerWallet,
        uint256 _maxTotalAllocation,
        uint256 _nftTicketAllocation,
        uint256 _rate,
        uint256 _nftFundraiseStartTime,
        uint256 _openFundraiseStartTime,
        uint256 _fundraiseEndTime,
        uint256 _distributionStartTime,
        address _owner,
        address _factory,
        address _nftToken
    ) {
        tokenConfig.depositToken = _depositToken;
        allocationConfig.baseAllocationPerWallet = _baseAllocationPerWallet;
        allocationConfig.maxTotalAllocation = _maxTotalAllocation;
        allocationConfig.nftTicketAllocation = _nftTicketAllocation;
        allocationConfig.rate = _rate;
        timeConfig.nftStartTime = _nftFundraiseStartTime;
        timeConfig.openStartTime = _openFundraiseStartTime;
        timeConfig.endTime = _fundraiseEndTime;
        timeConfig.distributionStartTime = _distributionStartTime;
        tokenConfig.factory = _factory;
        nftToken = _nftToken;
        _transferOwnership(_owner);
    }
```

Recommendation: Ensure that `nftStartTime` > `openStartTime` when initializing the contract.

```
+       require(timeConfig.nftStartTime > timeConfig.openStartTime, "NFT fundraise start time must be greater than
↪   open fundraise start time");
```

[L-2] Lack of restriction on vesting start before fundraising ends

Context: Fundraiser.sol

Impact The current implementation of the `Fundraiser` contract lacks restrictions to ensure that the vesting process can only be started after the fundraising period has ended. This vulnerability allows the vesting to be initiated before the fundraising period concludes.

Description: The `startVesting` function in the `Fundraiser` contract allows the owner to commence the vesting process by specifying the vesting start and end times, along with other parameters.

However, there are no checks in place to prevent the vesting from being initiated if the fundraising period has not yet ended.

Recommendation: To mitigate this vulnerability, it is advised to add a restriction that ensures the vesting process can only be initiated once the fundraising period has concluded. The following modification can be applied to the code:

```
    function startVesting(uint256 _vestingStart, uint256 _vestingEnd, address _tokenAddress, uint256 _tokenAmount,
    ↪   uint256 _ethFee) external whenNotPaused onlyOwner {
+       require(block.timestamp > timeConfig.endTime, "Fundraise has not ended yet");
```