GUILD AUDITS

MeProtocol

Initial Audit Report

January 29th, 2024

Contents

Executive Summary

This section will represent the summary of the whole audit process once it has concluded.

Project Summary

Project Name: MeProtocol

Description: The Me Protocol provides an instant, decentralised, trustless, autonomous, frictionless, extendable, and global infrastructure for incentivization and allows for the seamless cross-utilization of these incentives amongst participating businesses. Any business, large, medium, or small can instantly integrate its existing incentives into the protocol or create and manage new ones.

Codebase: https://github.com/Me-Protocol/solidity-protocol-v0.4

Commit: https://github.com/Me-Protocol/solidity-protocol-v0.4/tree/d265230578c4f7b4db8d0435b101001d6f0a5140

Project Audit Scope

The motive of this audit is to review the codebase of MeProtocol contracts for the purpose of achieving secured, correctness and quality smart contracts.

Number of Contracts in Scope:

- All the contract (5522 SLOC)

Duration for Audit: 3 weeks

Vulnerability Summary

Total issues: 17

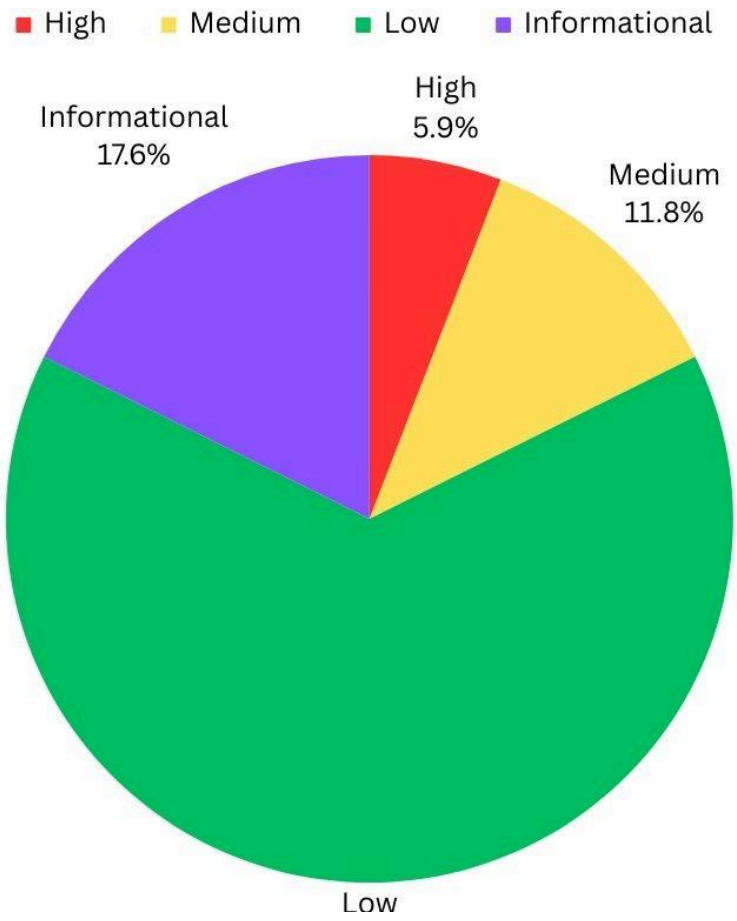
Total High: 1

Total Medium: 2

Total Low: 11

Total Informational: 3

Findings



Index	Title	Severity	Status of Issues
01	Pool manager can withdraw another pool manager's liquidity	**High**	**Open**
02	Liquidity sent to the pool by a pool manag can be recorded by another pool manager	**Medium**	**Open**
03	startOpenRewards can be frontrunned (Possible DOS Attack).	**Medium**	**Open**
04	createMoreRewards can mint to any Address.	**Low**	**Open**
05	ERC721::_mint() Safety Concern	**Low**	**Open**

06	Centralization Risk for trusted owner	**Low**	**Open**
07	Include Sanity check in constructor	**Low**	**Open**
08	**recordDepositedBountyRewards** is not being called anywhere in the codebase	**Low**	**Open**
09	Lack of reference to actual depositors whe recording deposited bounty rewards.	**Low**	**Open**
10	Missing arguments in **BrandsExtended::createMoreRewardT reasury**	**Low**	**Open**
11	Zero Address check missing	**Low**	**Open**
12	Solidity Pragma Version Ambiguity	**Low**	**Open**
13	Insecure abi.encodePacked() Usage for Hashing	**Low**	**Open**
14	Ecrecover signature malleability risk	**Low**	**Open**
15	Bad source of Brand ID generation	**Informational**	**Open**
16	Lack of adequate code comments	**Informational**	**Open**
17	Unused parameter in **Admin::updateProtocolRecords** functio	**Informational**	**Open**

Mode of Audit and Methodologies

The mode of audit carried out in this audit process is as follow:

- **Manual Review**: This is the first and principal step carried out to understand the business logic behind a project. At this point, it involves research, discussion and establishment of familiarity with contracts. Manual review is critical to understand the nitty-gritty of the contracts.
- **Automated Testing**: This is the running of tests with audit tools to further cement discoveries from the manual review. After a manual review of a contract, the audit tools which could be Slither, Echidna, or Mythril are run on the contract to find out issues.
- **Functional Testing**: Functional testing involves manually running unit, static, and dynamic testing. This is done to find out possible exploit scenarios that could be used to steal money from the contracts. This helps understand the functionality of the contracts and find out lapses in the reverts check in contract.

The methodologies for establishing severity issues:

- High Level Severity Issues
- Medium Level Severity Issues
- Low Level Severity Issues
- Informational Level Severity Issues

Types of Severity

Every issue in this report has been assigned a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impacts and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities that have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Report of Findings.

Issue: Pool managers can withdraw another pool manager liquidity.

Severity: High

Context: a-pool.sol

Description:

Pool managers can withdraw the liquidity of other pool managers, all that is needed is the liquidity position and the address of the owner of the liquidity for the withdrawal to be successful.

Impact of the vulnerability

This can lead to other pool managers not able to access their funds because it has been withdrawn by another pool manager.

POC:

```javascript
describe("withdrawAssetsFromPosition", function () {
        it("Should withdraw some token from a position of a pool manger", async function () {
          const { pool, owner, meToken, rewardToken, user1, user2, user3 } = await loadFixture(deployAPoolFixture);

          // Add new pool managers
          await pool.addOpenRewardsManager(user1.address);
          await pool.addOpenRewardsManager(user2.address);

          // Send tokens to User 2
          await meToken.transfer(user2.address, ethers.utils.parseEther("8"));
          await rewardToken.transfer(user2.address, ethers.utils.parseEther("8"));


          // Owner adds initial liquidity
          await meToken.transfer(pool.address, ethers.utils.parseEther("10"));
          await rewardToken.transfer(pool.address, ethers.utils.parseEther("10"));

          await pool.recordLiquidityProvided(
              {
                position: 0,
                rewardAmount: ethers.utils.parseEther("10"),
                meAmount: ethers.utils.parseEther("10"),
                requestor: owner.address,
                to: owner.address
              }
          );

          // Start Open Rewards
         await pool.startOpenRewards();

        // User 2 add liquidity to pool
        await meToken.connect(user2).transfer(pool.address, ethers.utils.parseEther("5"));
        await rewardToken.connect(user2).transfer(pool.address, ethers.utils.parseEther("5"));

        await pool.connect(user2).recordLiquidityProvided(
            {
              position: 0,
              rewardAmount: ethers.utils.parseEther("5"),
              meAmount: ethers.utils.parseEther("5"),
              requestor: user2.address,
              to: user2.address
            }
        );


      // User 1 withdraws User 2 liquidity
      await pool.connect(user1).withdrawLiquidity(
          {
            position: 2,
            rewardAmount: ethers.utils.parseEther("5"),
            meAmount: ethers.utils.parseEther("5"),
            requestor: user2.address,
            to: user1.address
          }
        );


      const currentMeBalance = await meToken.connect(user1).balanceOf(user1.address);
      const currentRewardBalance = await rewardToken.connect(user1).balanceOf(user1.address);

      console.log("cbMe", currentMeBalance); // 5 ethers
      console.log("cbR", currentRewardBalance); // 5 ethers

        });

      });
```

Recommendation:

There should be a check for who is initiating the withdrawLiquidity function to make sure that it is the requestor that is calling the function. (too small).

Status: **Acknowledged,** The team acknowledged the findings but posit that it is part of the system design.

Issue: Liquidity sent to the pool can be recorded by another pool manager.
Severity: Medium

Description:

Only pool managers can add liquidity to the pools, the process of adding liquidity is in two parts. The first part is to send the Me Token and the Reward Token to the pool. The second part is to record the liquidity. When we have multiple pool managers in the pool and Pool managers send liquidity to the pool at the same time, the pool manager who records the liquidity first will be the owner of the liquidity in the pool and when the other pool managers try to record their liquidity they will get an error.

Impact of the vulnerability

Pool managers will be unable to recordLiquidity as their liquidity has already been recorded by another pool manager. This is a form of disruption to the system.

POC:

```
describe("Deposit liquidity", function () {
    it("Should be able to track liquidity irrespective of order", async function () {
      const { pool, owner, meToken, rewardToken, user1 } = await loadFixture(deployAPoolFixture);

      //Owner transfer liquidity to the pool
      await meToken.transfer(pool.address, ethers.utils.parseEther("40000"));
      await rewardToken.transfer(pool.address, ethers.utils.parseEther("50000"));


      // New pool manager added
      await pool.addOpenRewardsManager(user1.address);

      // Transfers MeToken and Reward Token to User 1.
      await meToken.transfer(user1.address, ethers.utils.parseEther("20000"));
      await rewardToken.transfer(user1.address, ethers.utils.parseEther("30000"));

      // User 1 transfers liquidity to the pool.
      await meToken.connect(user1).transfer(pool.address, ethers.utils.parseEther("20000"));
      await rewardToken.connect(user1).transfer(pool.address, ethers.utils.parseEther("30000"));

      // Owner records liquidity provided
      await pool.recordLiquidityProvided(
        {
          position: 0,
          rewardAmount: ethers.utils.parseEther("50000"),
          meAmount: ethers.utils.parseEther("40000"),
          requestor: owner.address,
          to: owner.address
        }
      );

      const poolState = await pool.getOpenRewardsState();

      // The poolState has recorded the liquidity for both Owner and User1
      // or the owner

      await pool.connect(user1).recordLiquidityProvided(
        {
          position: 0,
          rewardAmount: ethers.utils.parseEther("20000"),
          meAmount: ethers.utils.parseEther("30000"),
          requestor: user1.address,
          to: user1.address
        }
      );

     // This request throws a `REQUEST_IS_NOT_WITHIN_ACCEPTED_RANGE()` error because
     // there is no liquidity in the pool as it has been recorded for the Owner

    });

});
```

Recommendation

The transferFrom should be used instead of transfer, the transferFrom and recordLiquidity should be in the same function so that the liquidity can be transferred and recorded in the same function.

Status: **Acknowledged,** The team acknowledged the findings but posit that it is part of the system design.

Issue: startOpenRewards can be frontrunned(Possible DOS Attack).
Severity: Medium
Context: A-Pool.sol

Description:

startOpenRewards verifies that the Liquidity Ratio is not greater than the rOptimal by checking the balance of the pool in both the reward token and the me token. Since the function depends on the RewardToken and MeToken. A user can make this function fail by transferring enough reward tokens to the pool address, thereby increasing the liquidity ratio before the function is called.

Impact of the vulnerability
The vulnerability opens up the possibility of a Denial of Service (DoS) attack, where an attacker deliberately disrupts the normal functioning of the startOpenRewards function, impacting the overall stability of the protocol.

POC:

```typescript
it("should revert when calling startOpenRewards", async function () {
  const { rewardToken, meToken, owner, user1 } = await loadFixture(
    deployAPoolFixture
  );

  const Pool = await ethers.getContractFactory("Pool");

  const pool = await Pool.deploy(rewardToken.address, meToken.address, {
    rOptimal: 2000000, // rOptimal is 2 * (10 ** 6)
    maximumRLimit: 3000000,
    minimumRewardAmountForConversation: ethers.utils.parseEther("1"),
    minimumMeAmountForConversation: ethers.utils.parseEther("1"),
    notifyRewardAmount: 0,
    notifyMeAmount: 0,
    defaultSlippageInPrecision: 5000000,
    allowSwaps: true,
  });

  await meToken.transfer(pool.address, ethers.utils.parseEther("40000"));
  await rewardToken.transfer(pool.address, ethers.utils.parseEther("80000"));

  await pool.recordLiquidityProvided({
    position: 0,
    rewardAmount: ethers.utils.parseEther("80000"),
    meAmount: ethers.utils.parseEther("40000"),
    requestor: owner.address,
    to: owner.address,
  });

  // User1 sends 10 tokens to the pool, disrupting the liquidity ratio calculation
  await rewardToken
    .connect(user1)
    .transfer(pool.address, ethers.utils.parseEther("0.04"));
  // Liquidity ratio will be 2000001

  await expect(pool.startOpenRewards()).to.be.rejectedWith(
    "OPEN_REWARDS_SHOULD_START_AT_R_OPTIMAL_OR_LESS()"
  );
});
```

Status: **Acknowledged.** The team acknowledged the findings but posit that it is part of the system design.

Issue: Include Sanity check in constructor
Severity: Low

Description:

The contract accrues absolute trust in the deployer to pass in the right parameters, nonetheless, any error in the argument would result in an absolute disruption in the bounty.sol contract activities.

```
Solidity
  constructor(address _meToken, address _protocolAdmin) {
          provider.setUpBounty(_meToken, _protocolAdmin);
      }
```

Impact of the vulnerability :

If there is no address zero check in the constructor, it could lead to serious consequences. For instance, if _meToken or _protocolAdmin were zero addresses, the setUpBounty function might behave unexpectedly or fail entirely. Furthermore, it could lead to other parts of the contract behaving incorrectly if they rely on these addresses being valid

Recommendation

Add an address zero check in the constructor.

```
Solidity
  constructor(address _meToken, address _protocolAdmin) {
          Validator.ensureAddressIsNotZeroAddress(_meToken);
          Validator.ensureAddressIsNotZeroAddress(_protocolAdmin);
          provider.setUpBounty(_meToken, _protocolAdmin);
      }
```

Status: Open **(Not Fixed by the Developers)**

Issue: recordDepositedBountyRewards is not being called anywhere in the code base
Severity: Low

Description:

The recordDepositedBountyRewards function checks and records deposited tokens in the bounty rewards state. Unfortunately, no contract is interacting directly with this function, how will the state be updated? as at the moment, anyone can interact with the recordDepositedBountyRewards function to update its state. What if no one calls the function? the state would remain dormant.

Impact of the vulnerability

This implies that the contract is not storing any information about the rewards deposited by users. This could be a problem if the contract needs to keep track of who has deposited what amount for future reference. Also, Allowing anyone to call the recordDepositedBountyRewards function can lead to potential security vulnerabilities. For instance, malicious actors could call this function to manipulate the recorded rewards, leading to unfair distributions or other issues. In advent of when the function is not being called, the integrity of the data could be compromised. For example, the function is supposed to update a state of all bounty rewards deposited, without calling it, this state would remain incomplete, leading to inaccuracies.This issue isnt domiciled to recordDepositedBountyRewards alone, it also appears in the following functions:

- addBrandId
- addBrandAccountManager
- removeBrandAccountManager

Recommendation

The Bounty rewards state should be updated at the point of depositing bounty rewards.

Status: **Open**

Issue: Lack of reference to actual depositors when recording deposited bounty rewards.
Severity: Low

Description:

When a deposit is made to the bounty contract, the actual depositor is not recorded. As such, anybody or bot can update the state of the deposited bounty's state without reference to the actual depositor.

Impact of the vulnerability :

Lack of reference to actual depositors when recording deposited bounty rewards can have significant implications in the context of a bounty contract. It can lead to a lack of transparency and accountability.

Recommendation

The actual depositor should be accounted for when updating the state of the bounty rewards storage. Also, the recording should be made on transfer.

Status: **Open**

Issue: Zero Address Check Missing
Severity: Low
Context: main.provider.sol

Description:

The setMainAdmin function within the MainProvider contract lacks a zero address check when updating the admin address. The absence of this check may pose a security risk, allowing the admin address to be set to the zero address, which could lead to unexpected behaviour or vulnerabilities in the system.

The code snippet without the zero address check is as follows:

```
// @audit there should be an address zero check
function setMainAdmin(address _newOwner) internal {
    MainStorage storage ms = mainStorage();
    address previousOwner = ms.admin;
    ms.admin = _newOwner;
    emit OwnershipTransferred(previousOwner, _newOwner);
}
```

Recommendation

To enhance security and prevent the admin address from being set to the zero address, include a zero address check in the setMainAdmin function. This check should be performed before updating the admin address.

Status: **Open (Not Fixed by the Developers)**

Issue: Centralization Risk for Trusted Owners
Severity: Low
Context: contracts/modules/deployables/a-reward-with-permit.sol Line: 62, 67
contracts/modules/deployables/a-reward-with-permit.sol Line: 67

Description:

The contracts in question feature owners endowed with privileged rights to execute admin tasks. However, this design introduces a centralization risk, as these trusted owners must be relied upon not to engage in malicious activities or unauthorised fund drainage. This vulnerability highlights the importance of scrutinising the roles and responsibilities of privileged entities within the smart contract ecosystem.

Impact of the vulnerability
The presence of privileged owners in contracts poses a centralization risk, requiring trust in their responsible use of admin tasks to prevent malicious updates or fund drainage.

Recommendation
1. Evaluate the necessity of centralised ownership and explore alternative governance models, such as decentralised autonomous organisations (DAOs), to mitigate centralization risks.

2. Implement additional checks and balances in the admin tasks performed by trusted owners, ensuring transparency and accountability in their actions.

3. Consider introducing multi-signature schemes or timelocks to require consensus among multiple entities before critical admin tasks can be executed.

Status: **Open** **(Not Fixed by the Developers)**

Issue: ECRecover Signature Malleability Risk
Severity: Low
Context: contracts/providers/peripherals/treasury.sol Line: 248
contracts/providers/peripherals/vault.sol Line: 57

Description:

The ecrecover function is prone to signature malleability (ecrecover is used to recover the public key from a signature and a message). This means that the same message can be signed in multiple ways, allowing an attacker to change the message signature without invalidating it [it is possible to modify the signature of a message in a way that does not invalidate the signature, but produces a different representation of the same message]. This can lead to unexpected behaviour in smart contracts, such as the loss of funds or the ability to bypass access control.

Impact of the vulnerability
Susceptibility of ecrecover to signature malleability introduces the risk of signature manipulation and potential security issues.

Recommendation
Given the susceptibility of ecrecover to signature malleability, it is advisable to consider alternative solutions for enhanced security. Utilising OpenZeppelin's ECDSA library instead of the built-in ecrecover function can provide a more robust and secure approach to signature handling. This proactive measure helps mitigate the risks associated with signature malleability, safeguarding the integrity and security of smart contract operations.

Status: **Open (Not Fixed by the Developers)**

Issue: Solidity Pragma Version Ambiguity
Severity: Low

Description:

Using a specific version of Solidity in the contracts instead of a dangling/wide version. for instance, instead of saying `pragma solidity ^0.8.0 ;` it is more encouraged and safe to `use pragma solidity 0.8.0 ;`

For example, not all blockchain networks uniformly support the PUSH0 opcode. With the introduction of Solc compiler version 0.8.20, the default target Ethereum Virtual Machine (EVM) version has been switched to Shanghai. This change implies that the bytecode generated will include PUSH0 opcodes.

Deploying smart contracts on blockchain networks that do not support PUSH0 or an earlier compiler version may result in deployment failures. It is crucial to be aware of the compatibility of generated bytecode with the target blockchain's EVM version to ensure successful and reliable deployment.

Impact of the vulnerability

The identified issue of not specifying a precise Solidity compiler version in the contracts introduces potential risks and compatibility issues across different Ethereum chains. By not locking down the pragma version, the codebase may inadvertently rely on default compiler settings, making it susceptible to inconsistencies and deployment failures on chains that do not support certain opcodes or features introduced in later compiler versions.

Recommendation

Explicitly specify the Solidity compiler version in pragma statements within the contracts. For example, use pragma solidity 0.8.0; to ensure compatibility and consistent behaviour across different Ethereum chains. Regularly review compiler release notes to make informed decisions on compiler version selection.

Status: **Open (Not Fixed by the Developers)**

Issue: Insecure abi.encodePacked() Usage for Hashing
Severity: Low
Context: contracts/providers/common/eunice.sol Line: 204

Description:

The contract exhibits potential security vulnerabilities arising from the improper use of abi.encodePacked() for hashing. Specifically, dynamic types are involved in the hashing process, and the current implementation does not ensure proper encoding, potentially leading to vulnerabilities.

```
bytes32 raw = keccak256(abi.encodePacked(mainAccount, name, onlinePresence,
datejoined, seed));
```

Impact of the vulnerability

The improper utilisation of abi.encodePacked() with dynamic types for hash functions within the contract poses a vulnerability in hashing. This vulnerability arises from the incorrect encoding of dynamic types, potentially compromising the integrity and security of the hashing process.

Recommendation

Consider using abi.encode() over abi.encodePacked() to fortify your hashing mechanisms. Unlike abi.encodePacked(), abi.encode() automatically ensures proper padding by adjusting items to 32 bytes. This mitigation strategy significantly reduces the risk of hash collisions. For instance, the potential collision issue in abi.encodePacked(0x123,0x456) generating the same hash as abi.encodePacked(0x1,0x23456) is mitigated when using abi.encode(0x123,0x456).

Unless there exists a compelling reason to use abi.encodePacked(), it is strongly recommended to embrace abi.encode() for enhanced security. Moreover, when dealing with a single argument, consider casting it to bytes() or bytes32() rather than resorting to abi.encodePacked(). For scenarios where all arguments are strings or bytes, the preferred approach is to employ bytes.concat(). This proactive approach safeguards against potential vulnerabilities and ensures the robustness of your hashing processes.

Status: **Open (Not Fixed by the Developers)**

Issue: createMoreRewards **can mint to any Address**
Severity: Low
Context: A-reward-with-permit.sol.sol

Description:

Authorised Reward managers can mint to any address when creating more rewards. This is likely not a bug if it is based on the design.

```
function createMoreRewards(address to, uint256 amount) external returns (bool) {
        {
            Validator.ensureValueIsNotZero(amount);
            Validator.ensureAddressIsNotZeroAddress(to);
        }
        if (provider.requestorIsAuthorized()) _mint(to, amount);

        emit Logs.moreRewardsCreated(amount, to, _msgSender());
        return true;
    }
```

Impact of the vulnerability
The absence of constraints may enable Authorised Reward managers to potentially misuse their minting privileges, leading to unintended consequences and potential exploitation.

Recommendation

It is advisable to allow the function to mint to only authorised addresses.

Status: Open **(Not Fixed by the Developers)**

Issue: ERC721::_mint() Safety Concern
Severity: Low
Context: a-pool.sol

Description:

The contract utilises ERC721::_mint() to mint ERC721 tokens; however, this implementation introduces a safety concern. ERC721::_mint() can potentially mint tokens to addresses that may not properly handle ERC721 tokens. To address this risk, it is advisable to replace the usage of _mint() with _safeMint(), a function specifically designed to mitigate such risks in ERC721 token minting.

Impact of the vulnerability
The use of ERC721::_mint() in the contract poses a potential risk, as it can mint ERC721 tokens to addresses that do not support ERC721 tokens

Recommendation
To enhance the security of your contract, it is strongly recommended to replace the current usage of ERC721::_mint() with ERC721::_safeMint(). The _safeMint() function includes additional checks to ensure that tokens are minted safely, mitigating potential issues that may arise when dealing with addresses that do not fully support ERC721 tokens. By adopting ERC721::_safeMint(), you add an extra layer of protection to the minting process, reducing the risk of unintended consequences and enhancing the overall robustness of your ERC721 token implementation.

Status: **Open (Not Fixed by the Developers)**

Issue: Bad source of Brand ID generation
Severity:Informational
Context: DePayForwarder.sol

Description:

The Brand ID is unique to each brand, which is a delicate means of identifying each brand within the me eco system. Off Chain computation of these brand IDs should be discouraged.

Impact of the vulnerability
If the Brand ID is generated off-chain, it could lead to inconsistencies in the data. Off-chain computations can introduce errors or manipulations that may not be immediately detectable but could cause problems later on. In addition, Malicious actors could potentially exploit this to gain unauthorised access or manipulate the data. For instance, they could alter the Brand ID to associate themselves with another brand's rewards.

Recommendation

The protocol should employ only onchain means of brandID generation while introducing a nonce to influence its randomness further.

Status: **Open (Not Fixed by the Developers)**

Issue: Unused Parameter in Admin::updateProtocolRecords function
Severity: Informational
Context: admin.controller.sol

Description:

The function updateProtocolRecords within the admin contract includes an unused parameter, ignoreDefault. This parameter is present in the function signature but is not utilised within the function body. The presence of unused parameters can potentially lead to confusion and may indicate a code smell, impacting the readability and maintainability of the codebase.

The unused parameter is observed in the following function declaration:

```
function updateProtocolRecords(Params.EditableProtocolRecords memory records, bo
ol ignoreDefault)
    external
    returns (bool)
{
    return provider.updateProtocolRecords(records);
}
```

Recommendation

To enhance code clarity and maintainability, it is recommended to either remove the unused parameter **ignoreDefault** or implement its functionality within the function if it serves a purpose. If the parameter is not intended for immediate use, consider removing it and documenting the removal in the function's comments for future reference.

Updated code without the unused parameter:

```
function updateProtocolRecords(Params.EditableProtocolRecords memory records)
    external
    returns (bool)
{
    return provider.updateProtocolRecords(records);
}
```

By eliminating unused parameters, the codebase becomes cleaner, easier to understand, and less prone to potential misunderstandings or confusion during future maintenance.

Status: Open (Not Fixed by the Developers)

Issue: Missing arguments in BrandsExtended::createMoreRewardToTreasury
Severity: Informational
Context: brand-extended.sol

Description:

The function createMoreRewardToTreasury encounters an issue where it passes only a single argument (_amount) to the internal function provider.createMoreRewardToTreasury, which expects both an address and a uint256. This inconsistency results in a mismatch in the expected and provided argument counts, leading to a compilation error.

The incorrect code snippet is as follows:

```
function createMoreRewardToTreasury(uint256 _amount) external returns (bool) {
    // @audit this function has wrong argument count
    return provider.createMoreRewardToTreasury(_amount);
}
```

The function being called internally is defined as follows:

```
function createMoreRewardToTreasury(address _rewardAddress, uint256 _amount) int
ernal returns (bool) {
    // Function implementation
    // ...
}
```

Recommendation

To address the issue and ensure correct function invocation, provide both the address and uint256 parameters when calling provider.createMoreRewardToTreasury .

Status:**Open (Not Fixed by the Developers)**

Issue: Lack of Adequate Code Comments, Including Natspec Documentation, in the Codebase
Severity:Informational
Context: All Files

Description:

The codebase lacks sufficient comments, including the absence of Natspec documentation. Natspec comments provide a standardised way to document Ethereum smart contracts, improving both human and automated understanding of the code. Comprehensive comments and Natspec documentation contribute to better code readability, maintenance, and collaboration among developers.

Impact of the vulnerability
The absence of adequate code comments and Natspec documentation in the codebase has notable consequences on the development, maintenance, and understanding of the protocol. Without proper comments, there's a risk of outdated or inaccurate documentation, leading to inconsistencies between the code and comments. This can misguide developers and result in unintended consequences during maintenance.

Recommendation

To enhance the codebase's documentation, it is advised to incorporate Natspec comments along with general comments. Natspec comments use a special syntax that is recognized by tools like Ethereum Natural Specification Format (NatSpec) and Solidity documentation generators.

Consider the following guidelines for adding Natspec comments:

1. Use Natspec comments to document contract descriptions, functions, and state variables.
2. Include details such as intended use,inputs,outputs, and any relevant considerations.
3. Leverage Natspec tags like @dev, @notice, @param, @return, etc., for structured documentation.
4. Ensure that Natspec comments are accurate, up-to-date, and reflective of the current contract state.

Status: **Open (Not Fixed by the Developers)**

Issue: Failed Function Execution Due to Bad Encoding
Severity: Medium
Context: brand-extended.sol

Description:

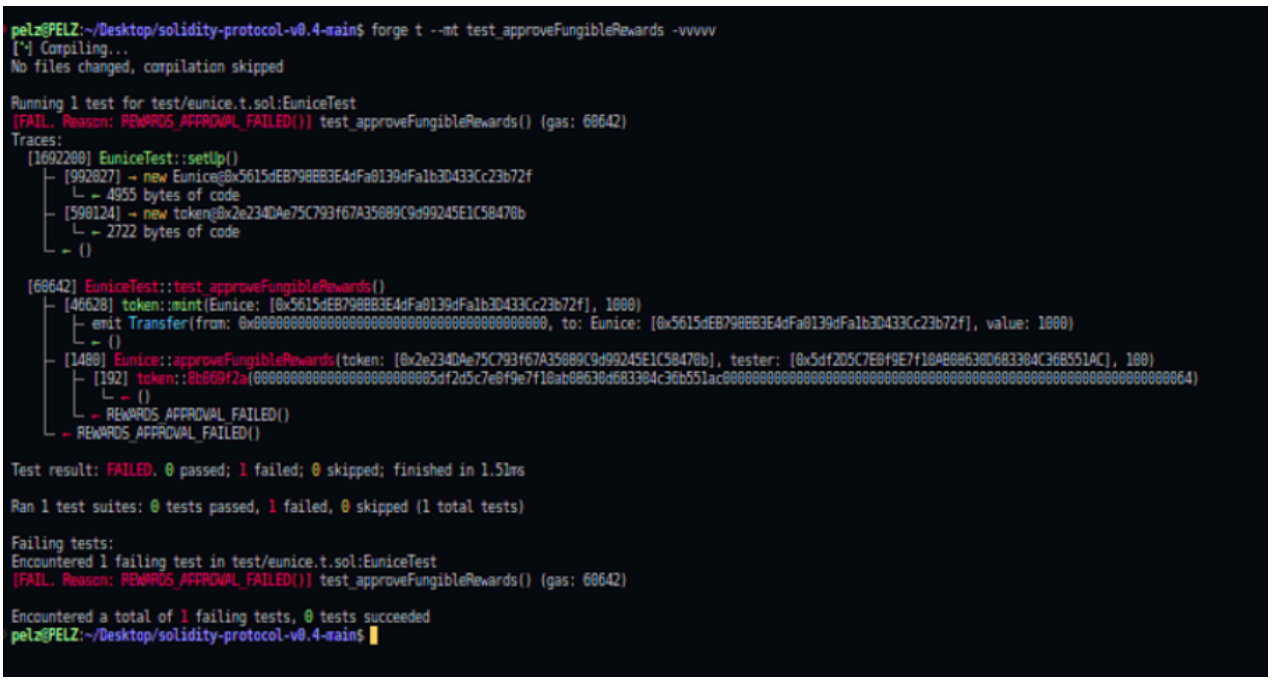
The `approveFungibleRewards` function utilizes the `call` method to execute an `approve` function on a specified ERC-20 token contract. However, the encoding of the function parameters is incorrect, there's a trailing space there which would be resulting in a failed function execution.

The issue comes from the encoding of the function parameters using the `abi.encodeWithSelector` function. This function is responsible for generating the encoded function call data to be used in the low-level `call` method. However, the encoding process is faulty, resulting in improperly formatted data being sent to the target ERC-20 token contract. As a consequence of the incorrect encoding, the function call to the `approve` function on the ERC-20 token contract will always fail.

```
// @audit-issue function will always fail due to bad encoding
    function approveFungibleRewards(address _erc20address, address _to, uint256 _am
        (bool success, bytes memory data) =
            _erc20address.call(abi.encodeWithSelector(bytes4(keccak256("approve(add

        if (!(success && (data.length == 0 || abi.decode(data, (bool))))) {
            revert Errors.REWARDS_APPROVAL_FAILED();
        }
    }
```

Function fails due to bad encoding:

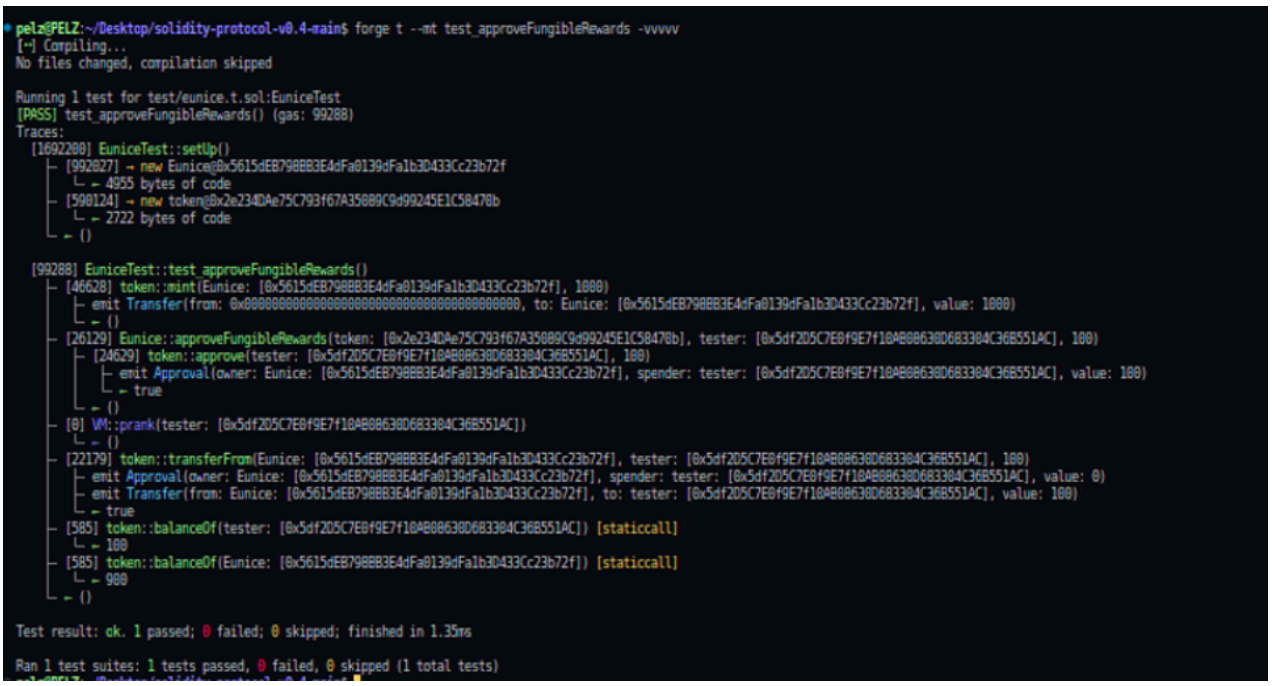


Recommendation

Ensure that the function parameters are correctly encoded to prevent failed transactions such as:

```
- _erc20address.call(abi.encodeWithSelector(bytes4(keccak256("approve(address, uint
+ _erc20address.call(abi.encodeWithSelector(bytes4(keccak256("approve(address,uint2
```

Once those changes are made, the function runs fine:



Status: **Open** **(Not Fixed by the Developers)**

Issue: Refactoring Recommendation: Consolidate Identical Functions
Severity:Informational
Context: eunice.sol

Description:

The `eunice.sol` contains two identical functions, `transferMeTokens` and `transferFungibleRewards`, with duplicate logic for transferring ERC-20 tokens. To improve code maintainability and reduce redundancy, it is recommended to consolidate these functions into a single generic function.

```
// @audit-info same function as tranfer me tokens
    function transferFungibleRewards(address _erc20address, address _to, uint256 _a
        (bool success, bytes memory data) =
            _erc20address.call(abi.encodeWithSelector(bytes4(keccak256("transfer(ad

        if (!(success && (data.length == 0 || abi.decode(data, (bool))))) {
            revert Errors.REWARDS_TRANSFER_FAILED();
        }
    }
```

```
// @audit-info same function as tranfer fungible rewards
    function transferMeTokens(address _erc20address, address _to, uint256 _amount)
        (bool success, bytes memory data) =
            _erc20address.call(abi.encodeWithSelector(bytes4(keccak256("transfer(ad

        if (!(success && (data.length == 0 || abi.decode(data, (bool))))) {
            revert Errors.REWARDS_TRANSFER_FAILED();
        }
    }
```

Recommendation

Consolidating identical functions involves creating a single, reusable function that can handle token transfers for various purposes. This approach streamlines code management and reduces the risk of inconsistencies or errors introduced by maintaining duplicate logic.

Consider the following guidelines for carrying out this recommendations

1. Create a Generic Transfer Function:

   Develop a single function capable of transferring ERC-20 tokens to a specified recipient. This function should accept parameters such as the ERC-20 token address, recipient address, and token amount.

2. Parameterize Function Purpose:

   Parameterize the function to support different token transfer scenarios, such as transferring rewards, user tokens, or any other fungible assets. This ensures flexibility and reusability across different parts of the protocol.

3. Consolidate Logic:

   Merge the duplicate logic from the transferMeTokens and transferFungibleRewards functions into the generic transfer function. Remove the redundant functions once the consolidation is complete.

Status: Open **(Not Fixed by the Developers)**

Closing Summary

There were discoveries of some high, medium, low and informational issues after the audit. The audit team thereafter suggested some remediation to help remedy the issues found in the contract.

Appendix

- Audit: The review and testing of contracts to find bugs.
- Issues: These are possible bugs that could lead exploits or help redefine the contracts better.
- Slither: A tool used to automatically find bugs in a contract.
- Severity: This explains the status of a bug.

Disclaimer

While the audit report is aimed at achieving a quality codebase with assured security and correctness, it should not be interpreted as a guide or recommendation for people to invest in **MeProtocol** contracts.

With smart contract audit being a multifaceted process, we admonish the **MeProtocol** team to carry out a bug bounty program to ensure continuous check is done to the contract.

Guild Audits

Guild Audits is geared towards providing blockchain and smart contract security in the fuming web3 world. The firm is passionate about remedying the constant hacks and exploits that deters the web3 motive.