Psydelve NFT Audit Report

29th May 2023

Executive Summary

This audit aimed to assess the security of the Psydelve NFT smart contract with a thorough review of the Solidity codebase, testing for common smart contract vulnerabilities, and evaluating the business logic of the contract to ensure it aligned with intended functionality. The code review identified several areas that require attention in order to improve the overall quality and security of the codebase. The most significant issue is the presence of external function calls within loops, which can potentially lead to a **Denial of Service (DoS)** vulnerability. Another noteworthy issue is the lack of a **2-step process for transferring ownership** in the "**Ownable**" contract. The code also lacks event emissions for critical functions that require the "**onlyOwner**" modifier. Furthermore, the use of a floating pragma, which does not specify a fixed version, may introduce compatibility issues or unintended behaviour when compiling the code with different Solidity compiler versions. Lastly, there is an informational issue regarding the missing upper limit check for royalty parameters in the "**setDefaultRoyalty**" and "**setTokenRoyalty**" functions.

Severity classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Overview

Project Name	Psydelve
Language	Solidity
CodeBase	Psydelve.sol

Audit Methodology	Business Logic & Code Review

Code Vulnerability Review Summary

The following number of issues were found, categorized by their severity:

- Critical & High: 0 issues
- Medium: 2 issues
- Low: 2 issues
- Informational : 1 issue

ID	Title	Category	Severity
Issue - 1	**External function calls within loops**	**DoS**	**Medium**
Issue - 2	**Ownable: Does not implement 2-Step-Process for transferring ownership**	**Privilege Related**	**Medium**
Issue - 3	**Missing events for `onlyOwner` functions that change critical parameters**	**Code Style**	**Low**
Issue - 4	**Use of floating pragma**	**Code Style**	**Low**
Issue - 5	**Missing Upper Limit Check for Royalty in `setDefaultRoyalty` & `setTokenRoyalty`**	**User Experience**	**Informational**

[Issue - 1] External function calls within loops

Severity

Impact: High, as important functionality in the protocol won't work

Likelihood: Low, as the user needs to technically strong, as there is a need to deploy a smart contract to perform DoS.

Description

The current implementation of `airdrop` function such that, the `onlyOwner` calls gives a list of eligible address and the tokenIds each of them are supposed to receive. This may look like a reasonable approach but there are a few considerations to keep in mind.

Each iteration of the for loop consumes gas, and there is a limit to the amount of gas that can be used in a single transaction. If the number of NFTs to be airdropped is large or if there are a large number of users, the gas cost for the entire airdrop operation might exceed the gas limit. This can lead to the airdrop failing or becoming prohibitively expensive.

When airdropping NFTs using a for loop, the operation is typically performed within a single transaction. This means that if an error occurs during the airdrop, the entire transaction can fail, and all previous airdropped NFTs will be reverted. This lack of transaction granularity might be undesirable if you want to ensure that partial airdrops are completed even if errors occur.

Lets understand this via an example: A malicious user can also perform a **Denial of Service** via a obscured attack. Suppose there are 10 users who are receiving the airdrop. The attacker is the one of the users in the list, and instead of EOA, their **smart contract** is eligible for the airdrop. The `onlyOwner` calls the `airdrop` function, since the code use `_safeMint`, which uses `onERC721Received` as a call-back mechanism to check if the contract is safe to receive NFTs, the attacker can cause a possible revert by writing a custom `onERC721Received` and block other users from getting a airdrop since the whole transaction reverts.

Example Code:

```
//Code used by attacker to perform Denial of Service

function onERC721Received(address _operator, address _from, uint256 _tokenId, bytes me
mory _data) public returns (bytes4) {
        // Cause a revert to demonstrate an error
        require(false, "Revert example");
        // Return the ERC721Received interface identifier
        return this.onERC721Received.selector;
}
```

[**Recommendation]:** A better approach would be to use **batch transfers or implementing off-chain solutions that distribute NFTs via merkle proof**. Batch transfers allow multiple NFT transfers to be performed in a single transaction, reducing the overall gas costs. Off-chain solutions involve storing the airdrop details outside the blockchain and providing a mechanism for users to claim their NFTs, using **pull-over-push** pattern.

[**Discussion]**

Issue Fixed.

[Issue - 2] Ownable: Does not implement 2-Step-Process for transferring ownership

Severity

Impact: High, Owner loses control of the smart contract.

Likelihood: Low, The owner verifies the address to assign ownership to when transferring the ownership.

Description

`Psydelve` inherits from the `Ownable` contract. The `Psydelve` contract does not implement a `2-Step-Process` for transferring ownership. So ownership of the contract can easily be lost when making a mistake when transferring ownership.

Since the `onlyOwner` modifies controls important function like pause, unpause and updating various addresses. It is better to follow a `2-Step-Process` for transferring ownership to prevent accident transfer of ownership to some other address.

[**Recommendation]:** Consider using the `Ownable2Step` contract from OZ (https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/Ownable2Step.sol) instead.

[**Discussion]**

Issue Fixed.

[Issue - 3] Missing events for `onlyOwner` functions that change critical parameters

Description

`onlyOwner` functions that change critical parameters should **emit events**. Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services. The alternative of directly querying on-chain contract state for such changes is not considered practical for most users/usages.

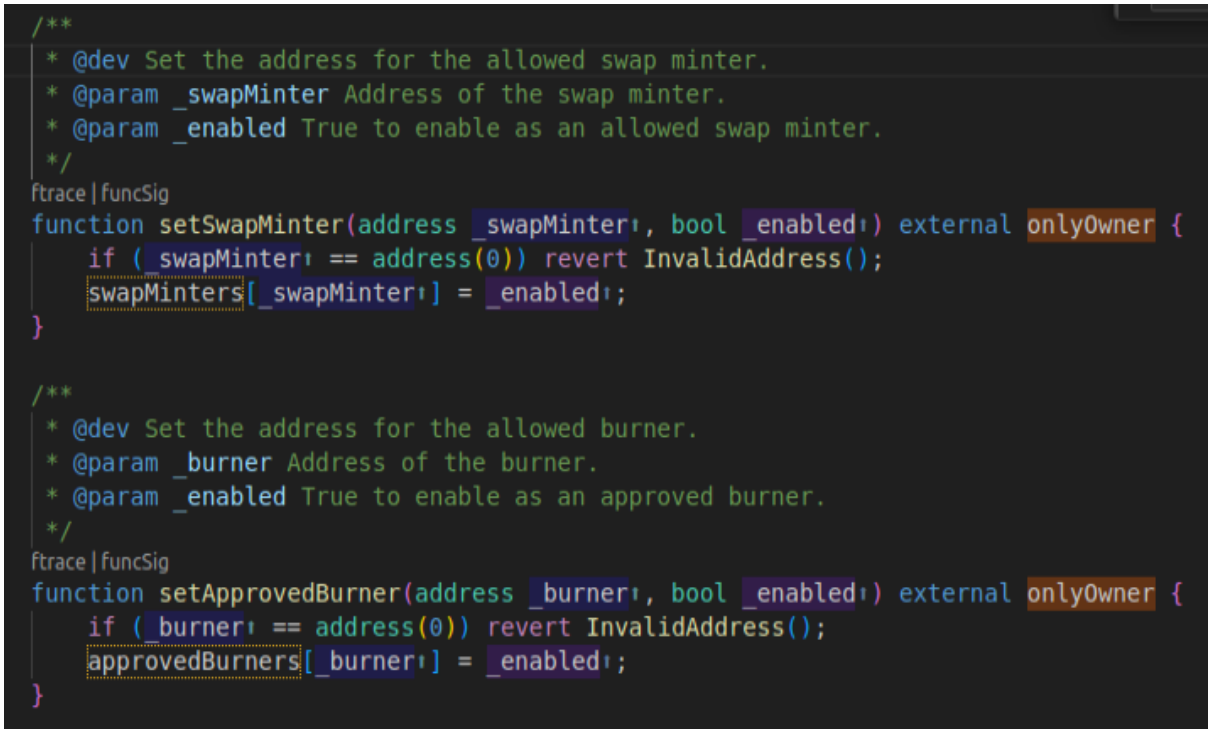
```
/**
 * @dev Set a valid signer to expect signatures to originate from.
 * @param _signer signer address.
 */
ftrace | funcSig
function setSigner(address _signer) external onlyOwner {
    if (_signer == address(0)) revert InvalidAddress();
    signer = _signer;
} //@audit -> no events emitted for critical changes in the code
```

Missing Event #1

```
/**
 * @dev Set the address for the Psydelve caps contract.
 * @param _psydelveCaps Address to the contract.
 */
ftrace | funcSig
function setPsydelveCaps(address _psydelveCaps) external onlyOwner {
    if (_psydelveCaps == address(0)) revert InvalidAddress();
    psydelveCaps = IERC721(_psydelveCaps);
}

/**
 * @dev Set the "dead" address to send caps to.
 * @param _deadAddress "Dead" address.
 */
ftrace | funcSig
function setDeadAddress(address _deadAddress) external onlyOwner {
    if (_deadAddress == address(0)) revert InvalidAddress();
    deadAddress = _deadAddress;
}
```

Missing Event #2



```
/**
 * @dev Set the address for the allowed swap minter.
 * @param _swapMinter Address of the swap minter.
 * @param _enabled True to enable as an allowed swap minter.
 */
ftrace | funcSig
function setSwapMinter(address _swapMinter, bool _enabled) external onlyOwner {
    if (_swapMinter == address(0)) revert InvalidAddress();
    swapMinters[_swapMinter] = _enabled;
}

/**
 * @dev Set the address for the allowed burner.
 * @param _burner Address of the burner.
 * @param _enabled True to enable as an approved burner.
 */
ftrace | funcSig
function setApprovedBurner(address _burner, bool _enabled) external onlyOwner {
    if (_burner == address(0)) revert InvalidAddress();
    approvedBurners[_burner] = _enabled;
}
```

Missing Event #3

[**Recommendation]:** Add events to all `onlyOwner` functions that change critical parameters.

[**Discussion]**

Issue Fixed.

[Issue - 4] Use of floating pragma

Description

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

Reference - https://swcregistry.io/docs/SWC-103

[**Recommendation] :** Lock the pragma version to version 0.8.19 and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

[**Discussion]**

Issue Fixed.

[Issue - 5] Missing Upper Limit Check for Royalty in `setDefaultRoyalty` & `setTokenRoyalty`

Description

There is a currently no cap for the royalty to be received for secondary sales. The `setDefaultRoyalty` & `setTokenRoyalty` update the royalty based on the value passed. To improve the transparency, a **constant** can be defined which basically tells that this is the maximum royalty we plan to collect from users.

[**Recommendation]:** Define a constant with a maximum royalty that can be collected and add a check of the same in `setDefaultRoyalty` & `setTokenRoyalty`

Example Fix

```
uint256 constant MAX_ROYALTY = 2500 //2500 would make the fee 25% (2500/10000).

function setDefaultRoyalty(address _receiver, uint96 _feeNumerator) external onlyOwner
{
    require(_feeNumerator <= MAX_ROYALTY, "_feeNumerator is greater than MAX_ROYALT
Y");
    _setDefaultRoyalty(_receiver, _feeNumerator);
}

function setTokenRoyalty(uint256 _tokenId, address _receiver, uint96 _feeNumerator)
external onlyOwner {
  require(_feeNumerator <= MAX_ROYALTY, "_feeNumerator is greater than MAX_ROYALTY");
  _setTokenRoyalty(_tokenId, _receiver, _feeNumerator);
}
```

[Discussion]

Issue Acknowledged.

Conclusion

The Psydelve NFT smart contract exhibits a strong commitment to security. However, the audit has identified several areas where improvements can be made to enhance both the contract's security and the user experience. To ensure the ongoing security and success of the Psydelve NFT project, addressing these findings through code modifications and improvements will help enhance the codebase's security, maintainability, and adherence to best practices. It is recommended to prioritize the resolution of the issues based on their severity and potential impact on the system.