TITLES Audit Report

Version 1.0

Ardeshir Gholami

April 25, 2024

TITLES Protocol Audit Report

Ardeshir Gholami

4 april 2024

Prepared by: Ardeshir Lead Auditors: - 4rdii

Table of Contents

Protocol Summary

TITLES builds creative tools powered by artist-owned AI models. The underlying TITLES protocol enables the publishing of referential NFTs, including managing attribution and splitting payments with the creators of the attributed works.

Disclaimer

I make all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of 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.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

The findings described in this document correspond the following commit hash:

wallflower-contract-v2 @ d23c44def46ce4fd74f3daae36df0135acae7505

Scope

- wallflower-contract-v2/src/TitlesCore.sol
- wallflower-contract-v2/src/editions/Edition.sol
- wallflower-contract-v2/src/fees/FeeManager.sol
- wallflower-contract-v2/src/graph/TitlesGraph.sol
- wallflower-contract-v2/src/shared/Common.sol

Roles

There are a few roles in the system with varying levels of power.

ADMIN_ROLE (Trusted) => Granted by the deployer to internal, trusted addresses only. On TitlesCore, this role can: 1) Change the ERC-1155 Edition implementation contract to an arbitrary address (`setEditionImplementation`). No post-auth validation is performed. 2) Upgrade the contract to an arbitrary new implementation (via `_authorizeUpgrade`, inherited and overridden with auth check from Solady's `UUPSUpgradeable`)

On TitlesGraph, this role can: 1) Create new Edges at will (`createEdges`). No post-auth validation is applied, except the typical uniqueness checks. 2) Upgrade the contract to an arbitrary new implementation (via `_authorizeUpgrade`, inherited and overridden with auth check from Solady's `UUPSUpgradeable`) 3) Grant or revoke any role to/from any address (`grantRole`, `revokeRole`).

On FeeManager, this role can: 1) Set the protocol fees (`setProtocolFees`). All fees are constrained to a constant range. 2) Create or change a fee route for any work within any Edition (`createRoute`). This is the only way to change the fee route for a work after publication. 3) Withdraw any funds locked in the contract (`withdraw`). This is the only way to withdraw funds from the contract. 3) Grant or revoke any role to/from any address (`grantRole`, `revokeRole`).

EDITION_MANAGER_ROLE (Restricted) => On an Edition, this role can: 1) Publish a new work with any desired configuration (`publish`). This is the only way to create new works after the Edition is created. 2) Mint promotional copies of any work (`promoMint`). There are no limitations on this action aside from the work's supply cap and minting period. 3) Set the Edition's ERC2981 royalty receiver (`setRoyaltyTarget`). This is the only way to change the royalty receiver for the Edition. 4) Grant or revoke any role to/from any address (`grantRole`, `revokeRole`).

EDITION_PUBLISHER_ROLE (Restricted) => On TitlesCore, this role can: 1) Publish a new work under any Edition for which they have been granted the role (i.e. `edition.hasAnyRole(msg.sender, EDITION_PUBLISHER_ROLE)` is true) (`publish`). After auth, the request is passed to the Edition contract for further handling.

EDITION_MINTER_ROLE (Restricted) => On an Edition, this role can: 1) Mint promotional copies of any work (`promoMint`). There are no limitations on this action aside from the work's supply cap and minting period.

Other roles which don't have specific role IDs: - Editions have an Ownable `owner` who can: 1) Mint promotional copies of any work (`promoMint`). There are no limitations on this action aside from the work's supply cap and minting period. 2) Grant or revoke EDITION_PUBLISHER_ROLE to/from any address (`grantPublisherRole`, `revokePublisherRole`). 3) Manage the ERC1155 contract in typical ways (e.g. transfer ownership). Notably, the owner CANNOT manage roles other than EDITION_PUBLISHER_ROLE.

- Works within an Edition have a `creator` who can:
  1) Update the minting period for the work (`setTimeframe`). This is the only way to change the minting period for a work after publication.
  2) Set the fee strategy for any work within the Edition (`setFeeStrategy`). This is the only way to change the fee strategy for a work after publication. The fee strategy is validated by the Fee Manager, and the final strategy (which may have been modified during validation) is applied immediately.
  3) Set the metadata for their own works. This is the only way to change the metadata for a work after publication.
  4) Transfer full ownership of the work to a new address (`transferWork`). This is the only way to change the creator for a work.
- FeeManager has an Ownable `owner` (essentially synonymous with `ADMIN_ROLE`, held by TitlesCore) who can:
  1) Set the protocol fees (`setProtocolFees`). All fees are constrained to a constant range. This role is granted to the TitlesCore contract whose currently scoped version does not have a mechanism for leveraging this permission directly.
  2) Create or change a fee route for any work within any Edition (`createRoute`). This is the only way to change the fee route for a work after publication.

Executive Summary

Issues found

Severity	number of issues found
High	0
Medium	2
Low	0
Info	0
Total	2

Findings

[M-1] `TitlesGraph::_setAcknowledged` does not change the state variable correctly

Summary

the `_setAcknowledged` function is an internal function which is called to change the `edge.acknowledged` bool in a certain edge given its `edgeId`, but instead of changing the `edges` mapping which is a state variable it changes the memory struct `edge`.

Vulnerability Detail

The _setAcknowledged function is designed to modify the acknowledged status of an Edge object within the edges mapping. However, the function only modifies a local memory copy of the Edge object, rather than updating the state variable directly. This means that the acknowledged status is not persisted across transactions, causing all edges to appear unacknowledged and incapable of being changed.

```
function _setAcknowledged(
        bytes32 edgeId_,
        bytes calldata data_,
        bool acknowledged_
    ) internal returns (Edge memory edge) {
        if (!_edgeIds.contains(edgeId_)) revert NotFound();
@>  edge = edges[edgeId_];
@>  edge.acknowledged = acknowledged_;

    if (acknowledged_) {
      emit EdgeAcknowledged(edge, msg.sender, data_);
    } else {
      emit EdgeUnacknowledged(edge, msg.sender, data_);
    }
  }
```

Impact

This bug impacts the functionality of the protocol, as it prevents the intended acknowledgment of edges. Consequently, all edges created within the protocol will remain unacknowledged, rendering them useless for their intended purposes. This issue could potentially lead to a halt in the protocol's operations, as the ability to acknowledge edges is a fundamental requirement for its functionality.

Code Snippet

to prove this concept just add `isAcknowledged` view function to `TitlesGraph.sol`, this function retruns the acknowledged bool for a given edgeId:

isAcknowledged view function:

```
contract TitlesGraph is
  IOpenGraph,
  IEdgeManager,
  OwnableRoles,
  EIP712,
  UUPSUpgradeable
{
  .
  .
  .
  //my functions
  function isAcknowledged(bytes32 edgeId_) external view returns (
     bool) {
    Edge memory edge;
    edge = edges[edgeId_];
    bool acknowledged = edge.acknowledged;
    return acknowledged;
  }

}
```

And make these changes to already exisiting test suits: `test_acknowledgeEdge_withSignature` and `test_acknowledgeEdge` in `TitlesGraph.t.sol`:

Test suit changes:

```
function test_acknowledgeEdge() public {
        Node memory from = Node({
            nodeType: NodeType.COLLECTION_ERC1155,
            entity: Target({target: address(1), chainId: block.chainid}),
            creator: Target({target: address(2), chainId: block.chainid}),
            data: ""
        });

        Node memory to = Node({
            nodeType: NodeType.TOKEN_ERC1155,
            entity: Target({target: address(3), chainId: block.chainid}),
            creator: Target({target: address(4), chainId: block.chainid}),
            data: abi.encode(42)
        });

        // Only the `from` node's entity can create the edge.
        vm.prank(from.entity.target);
        titlesGraph.createEdge(from, to, "");

        vm.expectEmit(true, true, true, true);
        emit IEdgeManager.EdgeAcknowledged(
            Edge({from: from, to: to, acknowledged: true, data: ""}),
            to.creator.target,
            ""
        );

        // Only the `to` node's creator (or the entity itself) can acknowledge it
        vm.prank(to.creator.target);
        titlesGraph.acknowledgeEdge(keccak256(abi.encode(from, to)), "");
+       bytes32 edgeId = titlesGraph.getEdgeId(from, to);
+       assertEq(titlesGraph.isAcknowledged(edgeId), false);
    }

    function test_acknowledgeEdge_withSignature() public {
        .
        .
        .
        // A different signature will acknowledge the edge again
        vm.expectEmit(true, true, true, true);
        emit IEdgeManager.EdgeAcknowledged(edge, address(this), "");
        titlesGraph.acknowledgeEdge(edgeId, new bytes(0), signer.SRSLY
           ());

+       assertEq(titlesGraph.isAcknowledged(edgeId), false);
    }
```

run these tests using command `forge test --mt test_acknowledgeEdge`, both tests pass meaning the acknowledged did not change, the same thing can be done for `unacknowledgeEdge` function tests.

Tool used

Manual Review, Foundry test suit

Recommendation

To fix this just change the state var in the `_setAcknowledged` function:

```
    function _setAcknowledged(
        bytes32 edgeId_,
        bytes calldata data_,
        bool acknowledged_
    ) internal returns (Edge memory edge) {
        if (!_edgeIds.contains(edgeId_)) revert NotFound();
        edge = edges[edgeId_];
-       edge.acknowledged = acknowledged_; //@audit this sets edge(
   memory) to acknowledged but not the storage one?
+       edges[edgeId_].acknowledged = acknowledged_;
        if (acknowledged_) {
            emit EdgeAcknowledged(edge, msg.sender, data_);
        } else {
            emit EdgeUnacknowledged(edge, msg.sender, data_);
        }
    }
```

[M2] `Edition::mintBatch` will revert if you try to mint more than 1 NFTs

Summary

The `mintBatch` function is designed to facilitate the minting of multiple NFTs in a single transaction. However, it encounters an issue where it fails to mint more than one NFT due to a problem with the handling of `msg.value` during the fee collection process. Specifically, the function sends the entire `msg.value` to `FeeManager::collectMintFee` on the first iteration, leaving no funds for subsequent iterations, which results in a transaction revert.

Vulnerability Detail

The mintBatch function is intended to allow for the minting of multiple quantities of NFTs for multiple works. It accepts an array of token IDs and corresponding amounts, along with other necessary parameters. The function iterates over these inputs, calling FeeManager::collectMintFee for each token ID to collect the minting fee. However, during the first iteration of this loop, the function transfers the entire msg.value to collectMintFee. This action depletes the value, leaving none for the subsequent token IDs, causing the function to revert when attempting to mint more than one NFT. Link to Code

```
function mintBatch1(
        address to_,
        uint256[] calldata tokenIds_,
        uint256[] calldata amounts_,
        bytes calldata data_
    ) external payable {

        for (uint256 i = 0; i < tokenIds_.length; i++) {
            Work storage work = works[tokenIds_[i]];

            // wake-disable-next-line reentrancy
            FEE_MANAGER.collectMintFee{value: msg.value}(
                this,
                tokenIds_[i],
                amounts_[i],
                msg.sender,
                address(0),
                work.strategy
            );

            _checkTime(work.opensAt, work.closesAt);
            _updateSupply(work, amounts_[i]);
        }

        _batchMint(to_, tokenIds_, amounts_, data_);
        _refundExcess();
    }
```

Impact

It prevents users from minting more than one NFT in a single transaction, which could lead to a poor user experience, more gas cost and hinder the application’s intended use. Additionally, if there is more ETH sotred in this contract, this issue could be exploited by malicious actors to intentionally to mint NFTs for free. but since there is almost no intended way to deposit ETH to this contract and all deposited ETH goes immidietly to `FeeManager` contract, probability of this to happen seems low.

Code Snippet

Here is a simple Test you can add to existing test suit to fix this issue: 1. Add a 2nd token Id to existing `Edition.t.sol::setUp` function:

setUp function changes:

```diff
function setUp() public {
        edition = new Edition();
        feeManager = new FeeManager(
            address(0xdeadbeef),
            address(0xc0ffee),
            address(new MockSplitFactory())
        );
        graph = new TitlesGraph(address(this), address(this));

        edition.initialize(
            feeManager,
            graph,
            address(this),
            address(this),
            Metadata({
                label: "Test Edition",
                uri: "ipfs.io/test-edition",
                data: new bytes(0)
            })
        );

-       edition.publish(
+       uint256 tId2 = edition.publish(
            address(1), // creator
            10, // maxSupply
            0, // opensAt
            0, // closesAt
            new Node[](0), // attributions
            Strategy({
                asset: address(0
                   xEeeeeEeeeEeEeeEeEeeEEEeeeeEeeeeeeeEEeE),
                mintFee: 0.01 ether,
                revshareBps: 2500, // 25%
                royaltyBps: 250 // 2.5%
            }),
            Metadata({
                label: "Best Work Ever",
                uri: "ipfs.io/best-work-ever",
                data: new bytes(0)
            })
        );
+       uint256 tId2 = edition.publish(
+           address(1), // creator
+           10, // maxSupply
+           0, // opensAt
+           0, // closesAt
+           new Node[](0), // attributions
+           Strategy({
+               asset: address(0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE),
+               mintFee: 0.01 ether,
+               revshareBps: 2500, // 25%
+               royaltyBps: 250 // 2.5%
+           }),
+           Metadata({
+               label: "Best Work Ever",
+               uri: "ipfs.io/best-work-ever",
+               data: new bytes(0)
+           })
+       );
+       assert(tId1 == 1);
+       assert(tId2 == 2);  // check tokenIds are correctly created

        // Normally done by the TitlesCore, but we're testing in isolation
        feeManager.createRoute(edition, 1, new Target[](0), address(0));
+       feeManager.createRoute(edition, 2, new Target[](0), address(0)); // add route for 2nd tokenId
    }
```

2. add this test function to `Edition.t.sol::setUp`:
   PoC:

```
function test_mintBatch1() public {
        address user = makeAddr("user");
        //minting 1 of each tokenId
        uint256[] memory tokenIds = new uint256[](2);
        tokenIds[0] = 1;
        tokenIds[1] = 2;
        uint256[] memory amounts = new uint256[](2);
        amounts[0] = 1;
        amounts[1] = 1;
        vm.startPrank(user);
        vm.deal(user, 1 ether);
        vm.expectRevert();
        edition.mintBatch1{value: 0.0212 ether}(
            user,
            tokenIds,
            amounts,
            new bytes(0)
        );
    }
```

Tool used

Manual Review, Foundry

Recommendation

To mitigate this issue, the `mintBatch` function should be modified to correctly distribute `msg.value` among all the NFTs being minted. This can be achieved by calculating the fee for each NFT individually and then subtracting this fee from `msg.value` before it is passed to the `collectMintFee` function. Here's a proposed implementation:

```
function mintBatch1(
        address to_,
        uint256[] calldata tokenIds_,
        uint256[] calldata amounts_,
        bytes calldata data_
    ) external payable {
        for (uint256 i = 0; i < tokenIds_.length; i++) {
            Work storage work = works[tokenIds_[i]];
+           uint256 ithMintFee = mintFee(i + 1); // get the mint fee
    for i+1 th tokenId note that tokenId starts from 1 not 0
            // wake-disable-next-line reentrancy
-           FEE_MANAGER.collectMintFee{value: msg.value}(
+           FEE_MANAGER.collectMintFee{value: ithMintFee}(
                this,
                tokenIds_[i],
                amounts_[i],
                msg.sender,
                address(0),
                work.strategy
            );

            _checkTime(work.opensAt, work.closesAt);
            _updateSupply(work, amounts_[i]);
        }

        _batchMint(to_, tokenIds_, amounts_, data_);
        _refundExcess();
    }
```