NFTMirror Security Review

Pashov Audit Group

Conducted by: Ch_301, Shaka, Dan Ogurtsov

December 30th 2024 - January 5th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **yuga-labs/NFTMIrror** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About NFTMirror

NFTMirror is a protocol that allows users to mirror their NFT collections on ApeChain. It works by deploying NFTShadows on ApeChain, which reflect ownership of NFTs locked on Ethereum mainnet via a Beacon contract. Users can trade, delegate, or use their mirrored NFTs on ApeChain while retaining control over the original asset on mainnet.

5. Risk Classification

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

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

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

6. Security Assessment Summary

review commit hash - 763a1a9ddbd340f24babba19cc4c7c73c39aa4f4

fixes review commit hash - 4929802e666a375bbd95f8ceed976ff42ab4180b

Scope

The following smart contracts were in scope of the audit:

- `Beacon`
- `NFTShadow`
- `CollectionConfig`
- `MetadataReadRenderer`
- `SafeCall`
- `ShadowFactory`

7. Executive Summary

Over the course of the security review, Ch_301, Shaka, Dan Ogurtsov engaged with Yuga Labs to review NFTMirror. In this period of time a total of **15** issues were uncovered.

Protocol Summary

Protocol Name	NFTMirror
Repository	https://github.com/yuga-labs/NFTMIrror
Date	December 30th 2024 - January 5th 2025
Protocol Type	NFT Crosschain messaging

Findings Count

Severity	**Amount**
Critical	1
High	2
Medium	3
Low	9
Total Findings	**15**

Summary of Findings

ID	Title	Severity	Status
[C-01]	Anyone can re-mint a token that was burned by the owner	Critical	Resolved
[H-01]	executeCallback call causes that ownership cannot be updated on the base chain	High	Resolved
[H-02]	lzReceive() call for releaseOnEid() results in OOG error	High	Resolved
[M-01]	No way to call triggerMetadataRead() from NFTShadow.sol	Medium	Resolved
[M-02]	Shadow factory does not work	Medium	Resolved
[M-03]	NFTs can get locked after the call to releaseOnEid	Medium	Resolved
[L-01]	Zero timestamp for confirmation value	Low	Acknowledged
[L-02]	The MessageCached event, always emits an empty reason	Low	Resolved
[L-03]	Check the user input in releaseOnEid()	Low	Resolved
[L-04]	Beacon's gas helper functions can underestimate the gas limit required for lzReceive()	Low	Resolved
[L-05]	tokenURI() does not revert if the id is not a valid NFT	Low	Resolved
[L-06]	multicall() will revert if empty data is included in the multicallData	Low	Resolved

[L-07]	Ownership status cannot be guaranteed to be up-to-date	Low	Acknowledged
[L-08]	Function can return address(0) if the owner has a large number of delegations	Low	Resolved
[L-09]	Incorrect function signature in initialize function	Low	Resolved

8. Findings

8.1. Critical Findings

[C-01] Anyone can re-mint a token that was burned by the owner

Severity

Impact: High

Likelihood: High

Description

The minting of tokens is restricted to the beacon contract by enforcing in the `_beforeTokenTransfer` hook that only the beacon can transfer locked tokens. As the default status of a token is locked, non-existing tokens are expected to be locked.

```
function _beforeTokenTransfer
      (address from, address to, uint256 tokenId) internal view override {
        if (msg.sender != BEACON_CONTRACT_ADDRESS) {
@>          if (tokenIsLocked(tokenId)) revert CallerNotBeacon();
        }
```

However, it has not been taken into account that tokens can be burned by the owner when they are not locked, and after they are burned, they are kept unlocked. This allows anyone to mint them again.

```
function burn(uint256 tokenId) external {
        if (tokenIsLocked(tokenId)) {
            _burn(tokenId);
        } else {
@>          _burn(msg.sender, tokenId);
        }
    }
```

Proof of concept

```
function testBurnAndMint() public {
    testUnlockTokens_ShadowCollection();

    vm.prank(baycShadow.ownerOf(tokenId));
    baycShadow.burn(tokenId);

    assertEq(baycShadow.tokenIsLocked(tokenId), false);
    baycShadow.mint(recipient, tokenId);
}
```

Recommendations

Lock tokens after they are burned by the owner.

```
function burn(uint256 tokenId) external {
        if (tokenIsLocked(tokenId)) {
            _burn(tokenId);
        } else {
            _burn(msg.sender, tokenId);
+           _setExtraData(tokenId, LOCKED);
        }
    }
```

8.2. High Findings

[H-01] `executeCallback` call causes that ownership cannot be updated on the base chain

Severity

Impact: Medium

Likelihood: High

Description

When `Beacon.triggerOwnershipUpdate()` is called, the new owners of the tokens are read from the target chain, and `Beacon._updateOwnership()` is executed on the source chain to update the ownership.

When this is executed on the NFT's native chain, `shadowAddress` is the address of the NFT contract, not the NFTShadow contract. As such, when `_shadow.executeCallback(guid)` is called at the end of the function, the transaction will revert, as the NFT contract does not have the `executeCallback` function.

```
564:     function _updateOwnership
  (bytes calldata _message, bytes32 guid) internal {
565:         (
  addressshadowAddress,
  address[]memorystaleOwners,
  address[]memorynewOwners,
  uint256[]memorytokenIds
) =
566:             abi.decode(_message,
  (address, address[], address[], uint256[]));
    (...)
575:         INFTShadow _shadow = INFTShadow(shadowAddress);
    (...)
595:         _shadow.executeCallback(guid);
596:     }
```

As a result, the ownership cannot be updated on the native chain.

Recommendations

```
-     _shadow.executeCallback(guid);
+     if (!isNative) _shadow.executeCallback(guid);
```

[H-02] `lzReceive()` call for `releaseOnEid()` results in OOG error

Severity

Impact: Medium

Likelihood: High

Description

The release of a shadow NFT on a different chain is initiated in the `NFTShadow.releaseOnEid()` function. This function calculates the `options` parameter for the LayerZero message as follows:

```
uint128 private constant _BASE_OWNERSHIP_UPDATE_COST = 80_000;
    (...)
    uint128 private constant _INCREMENTAL_OWNERSHIP_UPDATE_COST = 20_000;
    (...)
    function getSendOptions(uint256[] calldata tokenIds) public pure returns
      (bytes memory) {
        uint128 totalGasRequired =
            _BASE_OWNERSHIP_UPDATE_COST +
              (_INCREMENTAL_OWNERSHIP_UPDATE_COST * uint128(tokenIds.length));

        return OptionsBuilder.newOptions().addExecutorLzReceiveOption
          (totalGasRequired, 0);
    }
```

The issue is that the gas limit used for the execution of the `lzReceive()` call in the target chain is insufficient. For example, if the target chain is another shadow chain, it might be required to mint or transfer the shadow NFT. Minting a new shadow NFT costs ~46,700 gas and transferring a shadow NFT costs ~27,300 gas. That amount might be much higher if a transfer validator contract is used, as a call to `validateTransfer()` would be made before each mint or transfer. However, in the calculation of the gas limit, only 20,000 gas is added for each token ID.

This will lead to an out-of-gas error when executing the `lzReceive()` call, always requiring to retry executing the message with a higher gas limit.

Proof of concept

Add the following test to the file `NFTShadow.t.sol` and run `forge test --mt test_unlockShadowOOG -vvvv`.

```
function test_unlockShadowOOG() public {
    uint256[] memory tokenIds = new uint256[](1);
    tokenIds[0] = tokenId;

    // Calculate gasLimit used on the lzReceive() call
    uint128 _BASE_OWNERSHIP_UPDATE_COST = 80_000;
    uint128 _INCREMENTAL_OWNERSHIP_UPDATE_COST = 20_000;
    uint128 totalGasRequired = _BASE_OWNERSHIP_UPDATE_COST +
      (_INCREMENTAL_OWNERSHIP_UPDATE_COST * uint128(tokenIds.length));

    vm.selectFork(apechainForkId);
    vm.startPrank(address(beacon.endpoint()));
    bytes memory transferMessage = abi.encode(bayc, recipient, tokenIds);
    Origin memory origin = Origin(
      {srcEid:mainnetEid,
      sender:AddressCast.toBytes32
    )
    // DVN calls lzReceive() on the target chain
    beacon.lzReceive{ gas: totalGasRequired }(
        origin,
        bytes32(0),
        transferMessage,
        address(0),
        ""
    );
    vm.stopPrank();

    // Check that the message has been stored for retrial
    bytes32 payloadHash = beacon.payloadHashes
      (origin.srcEid, origin.sender, origin.nonce);
    assert(payloadHash != bytes32(0));
}
```

The snippet of the console logs:

```
|   |   ├─ [27985] NFTShadow::unlockTokens
      ([8903], 0x5C04911bA3a457De6FA0357b339220e4E034e8F7)
    |   |   |   └─ ← [OutOfGas] EvmError: OutOfGas
    |   |   └─ ← [Revert] EvmError: Revert
```

Recommendations

- Increase the amounts used in the calculation of the gas limit for the `lzReceive()` call in the `NFTShadow.getSendOptions()` function.
- Allow users to increase the gas limit for the `lzReceive()` call in the `NFTShadow.releaseOnEid()` function, beyond the default value.

8.3. Medium Findings

[M-01] No way to call `triggerMetadataRead()` from `NFTShadow.sol`

Severity

Impact: Low

Likelihood: High

Description

The `NFTShadow.sol#tokenURI()` function returns the token URI for token IDs by invoking `tokenURI()` from `MetadataReadRenderer.sol`. However, the `MetadataReadRenderer.sol` contract has a function `triggerMetadataRead()` to read token URIs from the base collection (from the original chain) and render them on the shadow collection. using the **lzRead** from LayerZero protocol.

But, because it takes `msg.sender` to determine the `baseCollectionAddress` value.

```
address baseCollectionAddress = IBeacon(beacon).shadowToBase(
            msg.sender
        );
```

The `triggerMetadataRead()` should get called from `NFTShadow.sol` which is not implemented yet. With the current implementation `MetadataReadRenderer.sol` can do nothing and `NFTShadow.sol#tokenURI()` can just return the base URI.

Recommendations

Create a new function in `NFTShadow.sol` to trigger `MetadataReadRenderer.sol#tokenURI()`.

[M-02] Shadow factory does not work

Severity

Impact: Low

Likelihood: High

Description

`onlyOwner` check in ShadowFactory is not initialized and thus has no owner set.

Solady's `Ownable` does not set the owner automatically on the deployment and requires calling `_initializeOwner()`, the way it is done in NFTShadow, for instance. As a result, any call of `deployAndRegister()` will always revert to checking the owner against `msg.sender`.

So the deployment flow is not centalized (is not going through the single source) and owners will have to deploy Shadow instances manually, which can be vulnerable to `initiailize()` frontrunning (if not called in the same transaction with the contract creation).

Recommendations

Add the constructor in ShadowFactory and call `_initializeOwner(msg.sender)`.

[M-03] NFTs can get locked after the call to `releaseOnEid`

Severity

Impact: High

Likelihood: Low

Description

Users can call `Beacon.releaseOnEid()` to lock tokens in the source chain and unlock them in the target chain. If the amount of tokens released is too high, it is possible that all the gas of the block is consumed in the target chain when unlocking the tokens. This would cause the tokens to be locked forever.

Recommendations

Limit the number of tokens that can be released at once.

8.4. Low Findings

[L-01] Zero timestamp for confirmation value

The function `Beacon.sol#_buildCmd()` sets the value of the confirmation to zero.

```
EVMCallComputeV1 memory computeSettings = EVMCallComputeV1(
            _MAP_AND_REDUCE,
            THIS_CONTRACT_EID,
            false,
            uint64(block.timestamp),
            0,//confirmations
            address(this)
        );
```

It is the number of confirmations required to wait for the timestamp finality on the target chain. This can be resolved by setting a reasonable number.

[L-02] The `MessageCached` event, always emits an empty reason

The event `MessageCached` on `Beacon.sol#_lzReceive()` will get emitted in case the execute of `safeCall()` failed

```
emit MessageCached(
                _origin.srcEid,
                _origin.sender,
                _origin.nonce,
                reason
            );
```

However, by setting zero as the `maxCopy` value in `safeCall()` the `reason` will always be empty. Consider cache at least the few first bytes will help.

[L-03] Check the user input in `releaseOnEid()`

In case `msg.sender` in `Beacon.sol#releaseOnEid()` is a shadow contract it will calculate the total gas required using `getSendOptions` However, if the collection is native, the caller must be the owner and he needs to build the `_options` himself, with no check for the value of `_options` in `releaseOnEid()` function, The `msg.sender` can set the total gas required to potentially a wrong value.

This will put the user at risk of locking both the original NFT and the shadow NFT permanently if the transaction reverts before caching the failed message in `payloadHashes[][][]`, especially in this line from `_lzReceive()`:

```
// Calculate gas to forward, leaving some in reserve
        uint256 externalGas = gasleft() - GAS_RESERVE;
```

To resolve this, use similar logic from `NFTShadow.sol#getSendOptions()` to check the input values. Note: This is classified as a user mistake, but the impact is scary.

[L-04] `Beacon`'s gas helper functions can underestimate the gas limit required for `lzReceive()`

The `getSendOptions()` and `getReadOptions()` functions of the `Beacon` contract are helper functions that can be used to estimate the gas limit for the `lzReceive()` call. However, these calculations do not account for the `GAS_RESERVE` value, which is fixed at 50,000.

Additionally, `getReadOptions` does not account for the execution of the callback function done in the `NFTShadow` contract.

This could lead to an underestimation of the gas limit required for the `lzReceive()` call.

[L-05] `tokenURI()` does not revert if the `id` is not a valid NFT

The specification of ERC-721 states the following regarding the `tokenURI` function: `Throws if _tokenId is not a valid NFT`.

However, the `tokenURI()` function in `NFTShadows` does not check if the `id` is a valid NFT, and therefore it is possible to call the function with an invalid `id` and get a response.

Consider adding a check to ensure that the `id` is a valid NFT before returning the token URI.

```
function tokenURI(uint256 id) public view override returns (string memory) {
+       if (!_exists(id)) revert TokenDoesNotExist();
+
        if (metadataRenderer != address(0)) {
            return ERC721(metadataRenderer).tokenURI(id);
        }

        return LibString.concat(_baseTokenUri, LibString.toString(id));
    }
```

[L-06] `multicall()` will revert if empty data is included in the `multicallData`

When an ownership update is triggered in the base chain of the NFT, the response `_updateDelegations()` is executed. In this function, `multicallData` is filled with the calls to update the delegations of the tokens for the stale owner and the new owner, and then `multicall()` is called.

```
function _updateDelegations(bytes calldata _message) internal {
        // message should now be an abi encoded array of abi encoded
        //(address, address, address, uint256) tuples
        (
          addresscollectionAddress,
          address[]memorystaleOwners,
          address[]memorynewOwners,
          uint256[]memorytokenIds
        )
        = abi.decode(_message, (address, address[], address[], uint256[]));
@>      bytes[] memory multicallData = new bytes[](tokenIds.length * 2);

@>      for (uint256 i = 0; i < tokenIds.length; ++i) {
            address staleOwner = staleOwners[i];
            address newOwner = newOwners[i];
            uint256 tokenId = tokenIds[i];

            if (staleOwner != address(0)) {
                multicallData[i] = abi.encodeWithSelector(
                    IDelegateRegistry.delegateERC721.selector,
                    staleOwner,
                    collectionAddress,
                    tokenId,
                    _GLOBAL_RIGHTS_WITH_MAX_EXPIRY,
                    false
                );
            }

@>          if (IERC721(collectionAddress).ownerOf(tokenId) == address(this)) {
                multicallData[i + tokenIds.length] = abi.encodeWithSelector(
                    IDelegateRegistry.delegateERC721.selector,
                    newOwner,
                    collectionAddress,
                    tokenId,
                    _GLOBAL_RIGHTS_WITH_MAX_EXPIRY,
                    true
                );

                delegatedOwners[collectionAddress][tokenId] = newOwner;
            } else {
                // if token is not owned by this contract, there should be no
                // delegation
                delegatedOwners[collectionAddress][tokenId] = address(0);
            }
        }

@>      IDelegateRegistry(DELEGATE_REGISTRY).multicall(multicallData);
    }
```

The code applies conditionals to determine what calls to include in the `multicallData` array. What is important to note here is that if any of the calls are not populated in the `multicallData` array, `multicall()` will revert, which will cause the ownership update to fail.

For this to happen, either `staleOwner` (taken from `delegatedOwners`) has to be `address(0)` or `IERC721(collectionAddress).ownerOf(tokenId)` has to be different from the beacon address. Both of these conditions should never happen, as a successful ownership update on the base chain implies that a shadow NFT is unlocked, and thus, the NFT in the base chain is locked.

Therefore, the conditional statements do not have any effects, as they will always be true. However, the potential reversion of `multicall()` if the `multicallData` array is not populated with all the calls should be taken into account in the case of future changes to the code. If at any point in the future, it is expected that the conditions might not be met, the size of the `multicallData` array should be reduced to only include the calls that are populated.

[L-07] Ownership status cannot be guaranteed to be up-to-date

The main purpose of the protocol is to allow contracts to read ownership of NFTs on other chains different from the native chain of the NFT. This could be used to give token owners access to airdrops, governance, rewards, or other benefits on other chains.

The update of ownership is triggered on the `Beacon` contract by calling `triggerOwnershipUpdate()`. However, due to the nature of cross-chain communication, there will always be a delay until the response from the target chain is received. This provokes the idea that it can never be guaranteed that the ownership status is up-to-date. That could lead to a situation where the old owner is unfairly considered as the owner of the NFT.

[L-08] Function can return `address(0)` if the owner has a large number of delegations

In the `Beacon` contract, the `unlockedExclusiveOwnerByRights()` function is executed on the target chain of an ownership update and fetches the owner of a token from the delegate resolver.

```
try IExclusiveDelegateResolver
      (EXCLUSIVE_DELEGATE_RESOLVER_ADDRESS).exclusiveOwnerByRights(
        _collectionAddress, tokenId, SHADOW_TOKEN_RIGHTS
    ) returns (address owner) {
        return owner;
    } catch {
        return address(0);
    }
```

The `exclusiveOwnerByRights()` function loops over all the delegations of the owner over the NFT. This can lead to gas exhaustion if the owner has a large

number of delegations and thus the `address(0)` will be returned. This will cause the shadow token to be burned in the source chain.

When the call to `exclusiveOwnerByRights()` reverts, consider returning `IERC721(collectionAddress).ownerOf(tokenId)` if it exists. Also, limit the amount of gas forwarded to the `exclusiveOwnerByRights()` function to ensure that `unlockedExclusiveOwnerByRights()` always returns an address.

[L-09] Incorrect function signature in `initialize` function

The function signature of the `initialize` function in the `INFTShadow` interface is incorrect, as it is missing the last parameter, `address _metadataRenderer`. This means that the interface cannot be used to call the `initialize` function in the `NFTShadow` contract.