SB SECURITY

OmniNFT

Security Review

June 23, 2024

Conducted by:
Blckhv, Independent Security Researcher
Slavcheww, Independent Security Researcher

Contents

1. About SBSecurity

SBSecurity is a duo of skilled smart contract security researchers. Based on the audits conducted and numerous vulnerabilities reported, we strive to provide the absolute best security service and client satisfaction. While it's understood that 100% security and bug-free code cannot be guaranteed by anyone, we are committed to giving our utmost to provide the best possible outcome for you and your product.

Book a Security Review with us at sbsecurity.net or reach out on Twitter @Slavcheww.

2. Disclaimer

A smart contract security review can only show the presence of vulnerabilities **but not their absence**. Audits are a time, resource, and expertise-bound effort where skilled technicians evaluate the codebase and their dependencies using various techniques to find as many flaws as possible and suggest security-related improvements. We as a company stand behind our brand and the level of service that is provided but also recommend subsequent security reviews, on-chain monitoring, and high whitehat incentivization.

3. Risk classification

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

3.1. Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - leads to a moderate loss of assets in the protocol or some disruption of the protocol’s functionality.
- **Low** - funds are not at risk.

3.2. Likelihood

- **High** - almost **certain** to happen, easy to perform, or highly incentivized.
- **Medium** - only **conditionally possible**, but still relatively likely.
- **Low** - requires specific state or **little-to-no incentive**.

3.3. Action required for severity levels

- High - **Must** fix (before deployment if not already deployed).
- Medium - **Should** fix.
- Low - **Could** fix.

4. Executive Summary

OmniNFT is a cross-chain NFT, that allows its users to mint, burn and transfer on any chain supported by the OmniCat token. The OmniCat token is locked into the Blast chain (source) when user mints NFT and then these OmniCat tokens are returned to the user when the NFT is burned, no matter on which chain. There are a limited number of NFTs in the collection, and the minting phase lasts as long as all the NFTs are not minted. Burns can only be initiated after the minting phase.

OmniNFT contracts have been audited through the Hyacinth platform.

Overview

Project	OmniNFT
Repository	https://github.com/omnicat-labs/omnicat-NFT
Commit Hash	6ef4b30423d623facae36588d7b4f9533738fffa
Resolution	e3b98eb3286b4a9a9feeb82da2e0c8e574f7bbe6
Timeline	Audit: June 11 - June 15, 2024 Mitigation: June 22 - June 23, 2024

Scope

OmniNft.sol
OmniNFTA.sol
OmniNFTBase.sol

Issues Found

Critical Risk	2
High Risk	1
Medium Risk	7
Low/Info Risk	3

5. Findings

5.1. Critical severity

5.1.1. OmniNFT::mint does not estimate fees for multiple tokens

Severity: Critical Risk

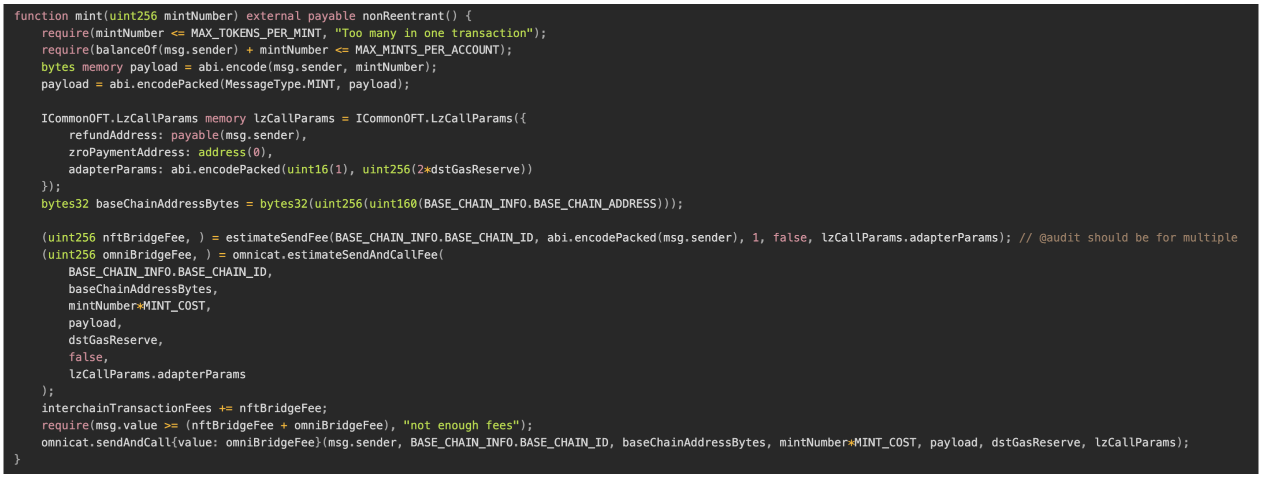
Context: OmniNFT.sol#L83

Description: Inside `OmniNFT::mint()` Lz fees for omniNFT mints should be estimated, so that users know the amount they need to send. These funds will be locked in the source chain (`OmniNFT`) and the same value will be used in the destination chain (`OmniNFTA`). Minting allows for multiple NFTs to be minted (up to 10) in one call, but the fee is only estimated for one `mint` operation, allowing the user to send fees for only one mint and specify to mint more.

- On source chain only gas for one mint will be stored inside interchainTransactionFees

- On destination chain, fee for multiple mints will be deducted from interchainTransactionFees

This will result in higher fees on the destination chain than the amount user has locked on the source chain, violating the invariant.



```
function mint(uint256 mintNumber) external payable nonReentrant() {
    require(mintNumber <= MAX_TOKENS_PER_MINT, "Too many in one transaction");
    require(balanceOf(msg.sender) + mintNumber <= MAX_MINTS_PER_ACCOUNT);
    bytes memory payload = abi.encode(msg.sender, mintNumber);
    payload = abi.encodePacked(MessageType.MINT, payload);

    ICommonOFT.LzCallParams memory lzCallParams = ICommonOFT.LzCallParams({
        refundAddress: payable(msg.sender),
        zroPaymentAddress: address(0),
        adapterParams: abi.encodePacked(uint16(1), uint256(2*dstGasReserve))
    });
    bytes32 baseChainAddressBytes = bytes32(uint256(uint160(BASE_CHAIN_INFO.BASE_CHAIN_ADDRESS)));

    (uint256 nftBridgeFee, ) = estimateSendFee(BASE_CHAIN_INFO.BASE_CHAIN_ID, abi.encodePacked(msg.sender), 1, false, lzCallParams.adapterParams); // @audit should be for multiple
    (uint256 omniBridgeFee, ) = omnicat.estimateSendAndCallFee(
        BASE_CHAIN_INFO.BASE_CHAIN_ID,
        baseChainAddressBytes,
        mintNumber*MINT_COST,
        payload,
        dstGasReserve,
        false,
        lzCallParams.adapterParams
    );
    interchainTransactionFees += nftBridgeFee;
    require(msg.value >= (nftBridgeFee + omniBridgeFee), "not enough fees");
    omnicat.sendAndCall{value: omniBridgeFee}(msg.sender, BASE_CHAIN_INFO.BASE_CHAIN_ID, baseChainAddressBytes, mintNumber*MINT_COST, payload, dstGasReserve, lzCallParams);
}
```

Recommendation:

- Use `estimateSendBatchFee()` and pass an array with the size of `mintNumber` as `_tokenIds`.

- Separate the `dstGasForCall` passed to the `Omnicat::sendAndCall` from the `dstGasReserve` , exposing 2 different setter functions for both the variables.

- In `OmniNFTA` - `dstGasReserve` should be converted to mapping by chainId and the respective `dstGasReserve` , and new `onlyOwner` setter function.

Resolution: Fixed

5.1.2. OmniNFTA can be called directly from OmniCat

Severity: Critical Risk

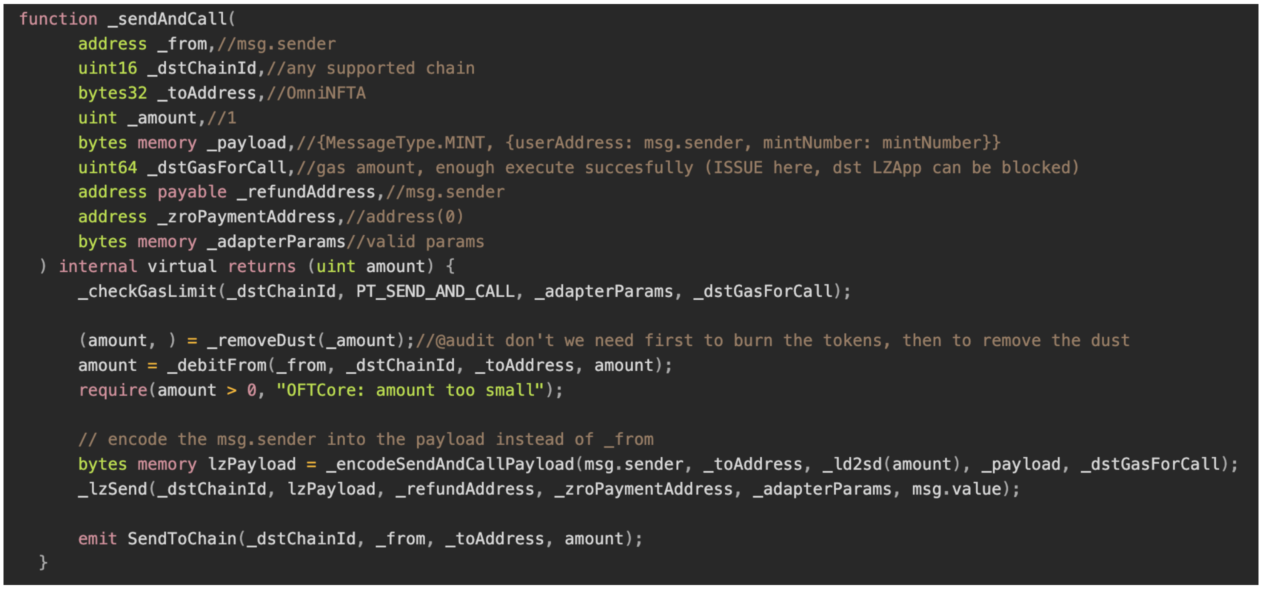
Context: *

Description: All the Omnicat tokens can be stolen from OmniNFTA by executing cross-chain send and call transfer directly from Omnicat token.

Other issues are that the omniBridgeFee can be avoided, for example, the user can execute sendAndCall from Omnicat on a supported chain, and make the OmniNFTA pay the dstGasReserve that will be taken from interchainTransactionFees, but the logic for storing the gas inside OmniNFT::interchainTransactionFees will not be executed and he will mint NFTs for free.

It is all possible because there will already be a configured trusted remote in omnicat for the Blast chain as OmniNFT::mint also relies on it.

The prerequisite needed for the attack is properly configured Omnicat::sendAndCall execution then all the attacker needs to do is to pass the minimum amount of omnicat tokens to be bridged and pass the entire balance of the OmniNFTA encoded in the payload as well as mintNumber above the max, so he is sure that omniUserRefund will be created:



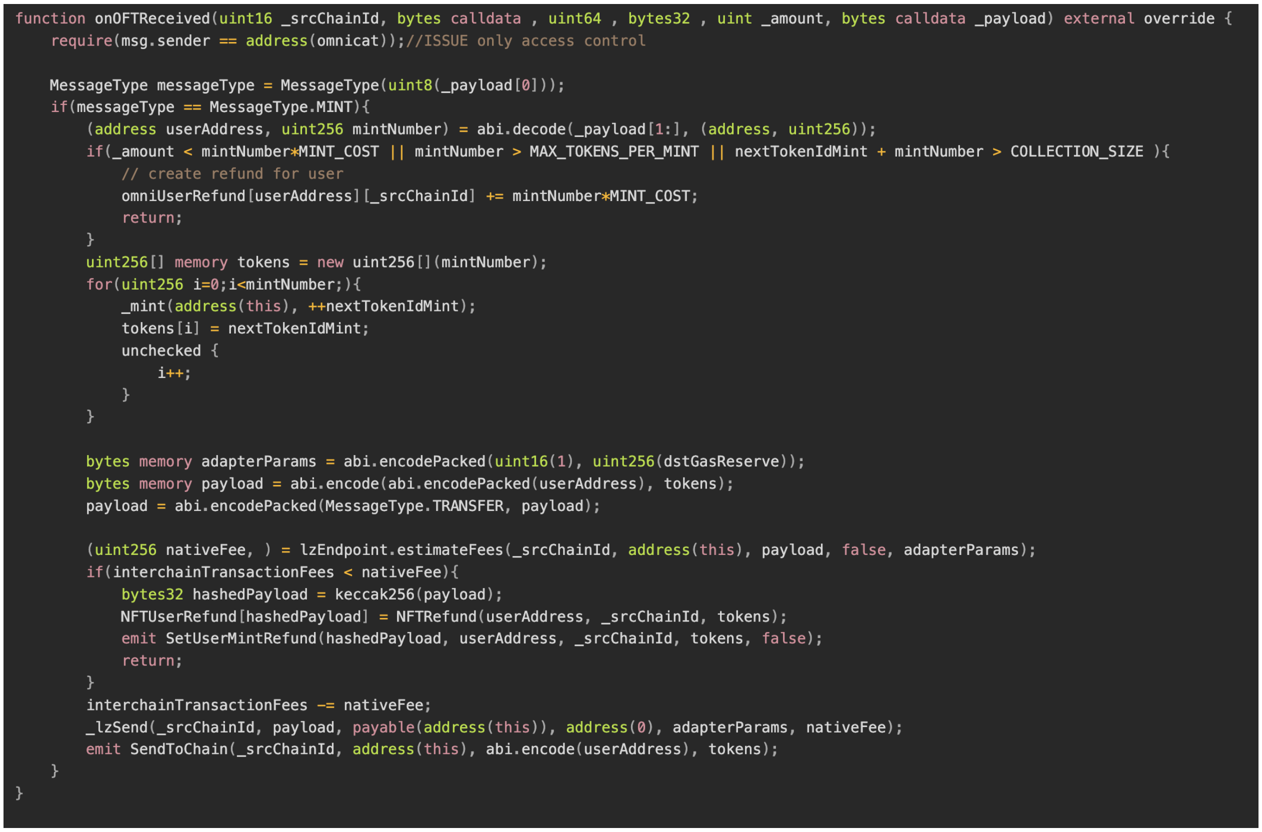
```
function _sendAndCall(
      address _from,//msg.sender
      uint16 _dstChainId,//any supported chain
      bytes32 _toAddress,//OmniNFTA
      uint _amount,//1
      bytes memory _payload,//{MessageType.MINT, {userAddress: msg.sender, mintNumber: mintNumber}}
      uint64 _dstGasForCall,//gas amount, enough execute succesfully (ISSUE here, dst LZApp can be blocked)
      address payable _refundAddress,//msg.sender
      address _zroPaymentAddress,//address(0)
      bytes memory _adapterParams//valid params
  ) internal virtual returns (uint amount) {
      _checkGasLimit(_dstChainId, PT_SEND_AND_CALL, _adapterParams, _dstGasForCall);

      (amount, ) = _removeDust(_amount);//@audit don't we need first to burn the tokens, then to remove the dust
      amount = _debitFrom(_from, _dstChainId, _toAddress, amount);
      require(amount > 0, "OFTCore: amount too small");

      // encode the msg.sender into the payload instead of _from
      bytes memory lzPayload = _encodeSendAndCallPayload(msg.sender, _toAddress, _ld2sd(amount), _payload, _dstGasForCall);
      _lzSend(_dstChainId, lzPayload, _refundAddress, _zroPaymentAddress, _adapterParams, msg.value);

      emit SendToChain(_dstChainId, _from, _toAddress, amount);
  }
```

After that LZEndpoint is executed successfully bridging to the destination Omnicat, and when the _decodeSendAndCallPayload is executed and user provided payload is decoded, OmniNFTA::onOFTReceived will be executed successfully, since there is no check for the origin sender of the transaction:

After successful bridging the if check for minting more than the MAX_TOKENS_PER_MINT will be triggered and omniUserRefund[userAddress][_srcChainId] += mintNumber*MINT_COST

(Omnicat.balanceOf(address(this)) will be created.

Then he only needs to manually process the refund with `OmniNFTA::sendOmniRefund`.



```
function onOFTReceived(uint16 _srcChainId, bytes calldata , uint64 , bytes32 , uint _amount, bytes calldata _payload) external override {
    require(msg.sender == address(omnicat));//ISSUE only access control

    MessageType messageType = MessageType(uint8(_payload[0]));
    if(messageType == MessageType.MINT){
        (address userAddress, uint256 mintNumber) = abi.decode(_payload[1:], (address, uint256));
        if(_amount < mintNumber*MINT_COST || mintNumber > MAX_TOKENS_PER_MINT || nextTokenIdMint + mintNumber > COLLECTION_SIZE ){
            // create refund for user
            omniUserRefund[userAddress][_srcChainId] += mintNumber*MINT_COST;
            return;
        }
        uint256[] memory tokens = new uint256[](mintNumber);
        for(uint256 i=0;i<mintNumber;){
            _mint(address(this), ++nextTokenIdMint);
            tokens[i] = nextTokenIdMint;
            unchecked {
                i++;
            }
        }

        bytes memory adapterParams = abi.encodePacked(uint16(1), uint256(dstGasReserve));
        bytes memory payload = abi.encode(abi.encodePacked(userAddress), tokens);
        payload = abi.encodePacked(MessageType.TRANSFER, payload);

        (uint256 nativeFee, ) = lzEndpoint.estimateFees(_srcChainId, address(this), payload, false, adapterParams);
        if(interchainTransactionFees < nativeFee){
            bytes32 hashedPayload = keccak256(payload);
            NFTUserRefund[hashedPayload] = NFTRefund(userAddress, _srcChainId, tokens);
            emit SetUserMintRefund(hashedPayload, userAddress, _srcChainId, tokens, false);
            return;
        }
        interchainTransactionFees -= nativeFee;
        _lzSend(_srcChainId, payload, payable(address(this)), address(0), adapterParams, nativeFee);
        emit SendToChain(_srcChainId, address(this), abi.encode(userAddress), tokens);
    }
}
```

Recommendation:

- Modify the `OmniNFTA::onOFTReceived` to verify the initiator of the interchain mint and make sure only the appropriate `OmniNFT` contracts can call `onOFTReceived`, as the proposed additions should be thoroughly tested:

Resolution: Fixed

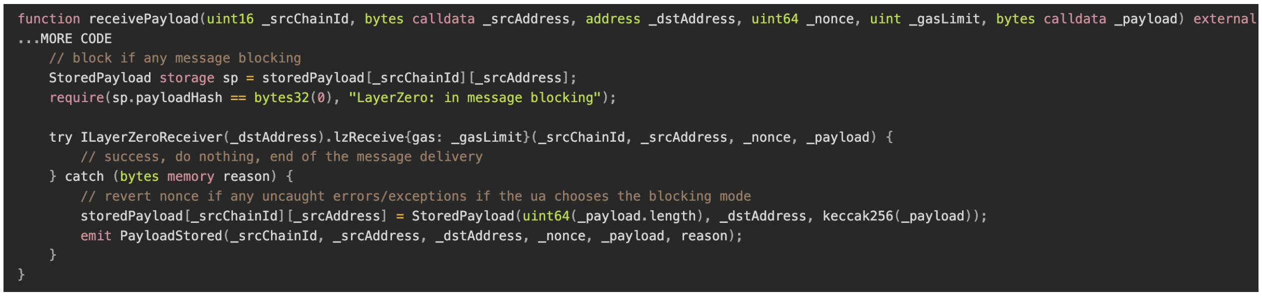
5.2. High severity

5.2.1. Any OmniNFT path can be blocked

Severity: High Risk

Context: OmniNFTBase.sol

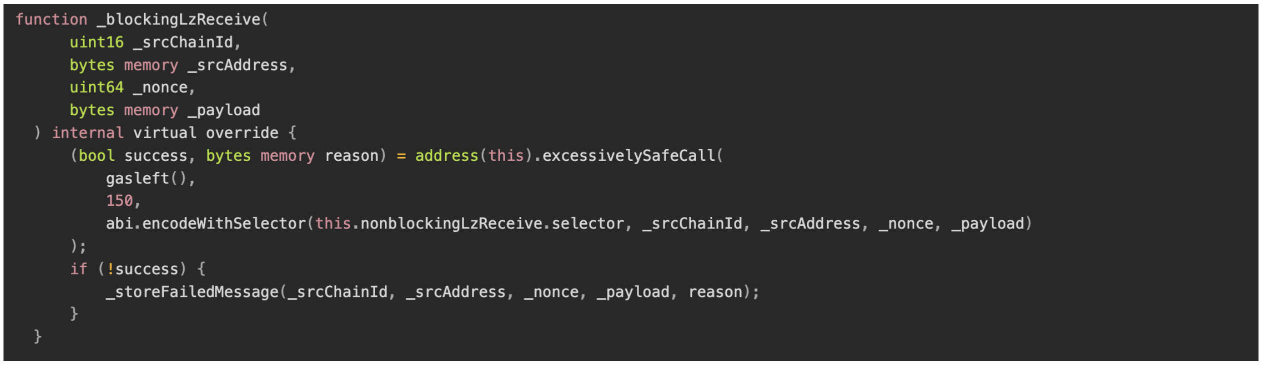
Description: To bridge ONFT from one chain to another `sendFrom` and `sendBatchFrom` can be used, both of them internally call the `OmniNFTBase::_send` function. The problem is that a malicious `OmniNFT` holder can choose any destination chain where his NFTs are not minted yet and pass the minimum gas configuration possible, which will block the entire path because `Endpoint` contract of `LayerZero` has this requirement:



```
function receivePayload(uint16 _srcChainId, bytes calldata _srcAddress, address _dstAddress, uint64 _nonce, uint _gasLimit, bytes calldata _payload) external
...MORE CODE
    // block if any message blocking
    StoredPayload storage sp = storedPayload[_srcChainId][_srcAddress];
    require(sp.payloadHash == bytes32(0), "LayerZero: in message blocking");

    try ILayerZeroReceiver(_dstAddress).lzReceive{gas: _gasLimit}(_srcChainId, _srcAddress, _nonce, _payload) {
        // success, do nothing, end of the message delivery
    } catch (bytes memory reason) {
        // revert nonce if any uncaught errors/exceptions if the ua chooses the blocking mode
        storedPayload[_srcChainId][_srcAddress] = StoredPayload(uint64(_payload.length), _dstAddress, keccak256(_payload));
        emit PayloadStored(_srcChainId, _srcAddress, _dstAddress, _nonce, _payload, reason);
    }
}
```

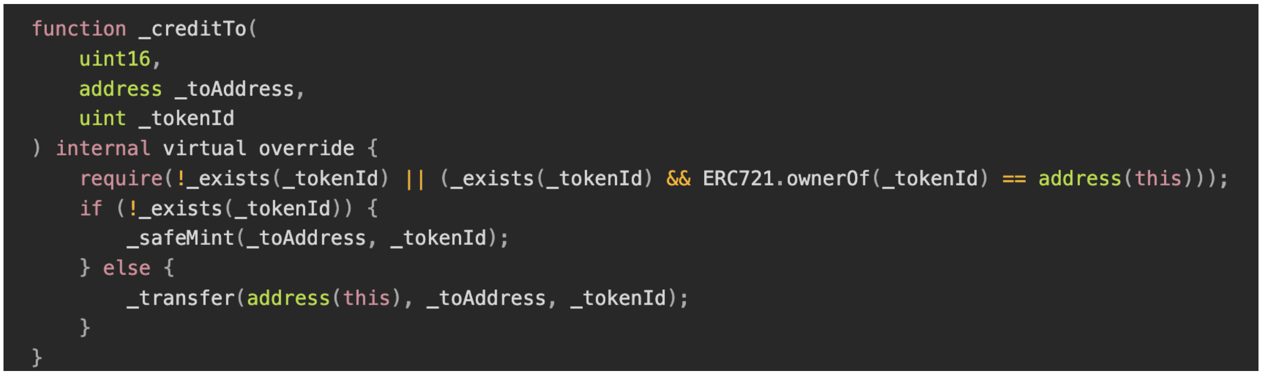
The blockage can be caused by 2 types of people - attackers simply wanting block the channel or normal users who have provided the minimum gas allowed:

1. Destination chain `_blockingLzReceive` is executed, which calls the nonblocking function and forwards all the available gas:



```
function _blockingLzReceive(
      uint16 _srcChainId,
      bytes memory _srcAddress,
      uint64 _nonce,
      bytes memory _payload
  ) internal virtual override {
      (bool success, bytes memory reason) = address(this).excessivelySafeCall(
          gasleft(),
          150,
          abi.encodeWithSelector(this.nonblockingLzReceive.selector, _srcChainId, _srcAddress, _nonce, _payload)
      );
      if (!success) {
          _storeFailedMessage(_srcChainId, _srcAddress, _nonce, _payload, reason);
      }
  }
```

2. `_creditTo` in `OmniNFT::_nonblockingLzReceive` will mint the NFT to the provided user and it will consume all the available gas in his onERC721Received callback:



```
 function _creditTo(
     uint16,
     address _toAddress,
     uint _tokenId
 ) internal virtual override {
     require(!_exists(_tokenId) || (_exists(_tokenId) && ERC721.ownerOf(_tokenId) == address(this)));
     if (!_exists(_tokenId)) {
         _safeMint(_toAddress, _tokenId);
     } else {
         _transfer(address(this), _toAddress, _tokenId);
     }
 }
```

3. OOG will be bubbled to `_blockingLzReceive` and will try to store the message in `_storeFailedMessage` which that will also run out of gas due to the above mentioned factors.

4. The last step is `Endpoint::receivePayload` and the `try/catch` here which will successfully store the failed message, as the 1/64 of the gas provided will be most amongst the previous calls.

5. As we can see until there is an non-zero `payloadHash` this path is blocked.

Although the path can be unblocked with Endpoint::forceResumeReceive, this will not resubmit the message, but will drop it instead, potentially harming non-malicious caller.

Recommendation: From our calculations the length of the payload is at most 500 with 10 ONFTs passed to the payload and it will roughly cost ~30k gas to be saved successfully in the _blockingLzReceive function. The following changes can be applied:

```
function _blockingLzReceive(
      uint16 _srcChainId,
      bytes memory _srcAddress,
      uint64 _nonce,
      bytes memory _payload
  ) internal virtual override {
+     uint256 gasToStoreFailedPayload = gasLeft() - 30000;
      (bool success, bytes memory reason) = address(this).excessivelySafeCall(
-         gasleft(),
+         gasToStoreFailedPayload,
          150,
          abi.encodeWithSelector(this.nonblockingLzReceive.selector, _srcChainId, _srcAddress, _nonce, _payload)
      );
      if (!success) {
          _storeFailedMessage(_srcChainId, _srcAddress, _nonce, _payload, reason);
      }
  }
```

Resolution: Fixed

5.3. Medium severity

5.3.1. OmniNft::estimateMintFee should be based on multiple tokens

Severity: Medium Risk

Context: OmniNft.sol#L57

Description: OmniNFT::estimateMintFees is used to provide an estimation about the fee that the caller should provide in order for execution to be successful, but the possibility of minting more than one NFT is not considered and thus the payload passed to lzEndpoint::estimateFee will be shorter, returning smaller nativeFee to be paid. In reality, a lot more gas will be considered as the size of the payload is 320 and 32, for an array with 10 tokenIds and 1 tokenId respectively.

Recommendation: The purpose of OmniNFT::estimateMintFees is to estimate the amount the user should send as gas when calling OmniNFT::mint, but since there is also a problem with gas estimation inside OmniNFT::mint (Issues C-01), based on the mitigation there, OmniNFT::estimateMintFees should be changed to the same flow.

Resolution: Fixed

5.3.2. OmniNFT excess msg.value is not refunded to the caller

Severity: Medium Risk

Context: OmniNft.sol#L94, L132

Description: When performing interchain minting and burning, users should pay both Omnicat and OmniNFT bridging fees. Currently, there is a require statement enforcing the provided native tokens exceed both fees so the protocol is properly paid on the destination chain as well.

The problem is that the remainder from this calculation: msg.value - (omniBridgeFee + nftBridgeFee) will be locked in the OmniNFT contract and there will be no way to be extracted because only amount up to interchainTransactionFees can be recovered from OmniNFTBase::extractNative.

We say it will happen pretty often because of the fluctuating fee values of LayerZero will cause users provide slightly more native tokens, in order to avoid reverts. Important to note that the excess nftBridgeFee, refunded to the user from the ULN is not part of the issue.

Recommendation: Consider refunding the excess msg.value to the caller.

Resolution: Fixed

5.3.3. OmniNFTA has no Blast gas fee configuration

Severity: Medium Risk

Context: OmniNFTA.sol#L58

Description: OmniNFTA contract will be deployed on Blast and thus is eligible to receive sequencer fees:

docs.blast.io

"Existing L2s like Optimism and Arbitrum keep sequencer fees for themselves. Blast redirects sequencer fees to the dapps that induced them, allowing smart contract developers to have an additional source of revenue."

However, since OmniNFTA has no such configuration, fees cannot be accumulated.

Recommendation: Depending on whether you want to set someone else as a governor and manage the fee configuration of the contract, there are 2 possible solutions:

1. Claim fees directly from the OmniNFTA contract
   a. Inherit the IBlast interface
   b. fIn the constructor call configureClaimableGas
   c. add additional claim access-controlled function and claim the fees regularly
2. Configure the admin (or other trusted address) as a governor, who will have the ability to enable/disable the gas mode and claim the fees
   a. inherit the IBlast interface
   b. in the constructor call configureGovernor(governor)

Resolution: Fixed

5.3.4. OmniNFT::mint check for minting above collection size is missing

Severity: Medium Risk

Context: OmniNft.sol#L70

Description: Currently, interchain mints rely on the OmniNFTA to decide whether the last NFT from the collection has been minted, and if so queue an Omnicat token refund. This approach is problematic because users will have to pay for both source and destination calls when this happens - on the source pay the LayerZero to bridge the omnicat tokens, and on the destination to initiate a refund from OmniNFTA::sendOmniRefund. As a result, users will spend money on executing unnecessary cross-chain calls.

Recommendation: The easiest approach is to have a function to manually disable the interchain mints, but you will have to execute it across all the supported chains independently.

Resolution: Acknowledged

5.3.5. OmniNFT underestimates the nftBridgeFee when executed from chains with more expensive gas than Blast

Severity: Medium Risk

Context: OmniNft.sol#L83, L128

Description: For interchain mints the nftBridgeFee is taken from the user to pay for the Blast → Source Chain transaction to pay to LayerZero when bridging the minted NFT back to the originator chain. But Blast chainId is used from the source chain for this calculation, which will make the users pay only a fraction of the actual fee that will be taken when the OmniNFTA contract sends the message with the minted NFTs back, especially when the source chain has more expensive gas prices, for example, ETH:

Recommendation: As discussed, a possible solution would be to estimate the actual fees in OmniNFTA and send them with the payload each time, so it can be decoded and used in the subsequent omnichain mints. This approach depends on the frequency the transactions but will give more accurate estimation than the current approach.

Resolution: Fixed

5.3.6. OmniNFT interchain mint provides small dstGasForCall

Severity: Medium Risk

Context: OmniNft.sol

Description: Currently, dstGasForCall, which is being used in the omnicat::callOnOFTReceived as a gas limit for the OmniNFTA::onOFTReceived is set to 1e6, but it will most likely be insufficient as minting a single NFT will consume ~47711 gwei (in Foundry). The provided dstGasForCall should be enough for the transaction flow to be executed all the way up to UltraLightNodeV2::send, but it definitely won't be.

Note that the extraGas passed in adapter params will be used only for the execution from LzEndpoint::lzReceive to the onOFTReceived call below. After that, the dstGasForCall will be passed:

There is no loss of tokens, nor NFTs but the failed transactions will have to be manually processed from the Blast chain from here.

Recommendation: Extensive testing should be performed in order to validate the proper dstGasForCall that has to be provided, in case there is a big deviation from the gas costs from chain to chain mapping with access-controlled setter can be used to have more granular control.

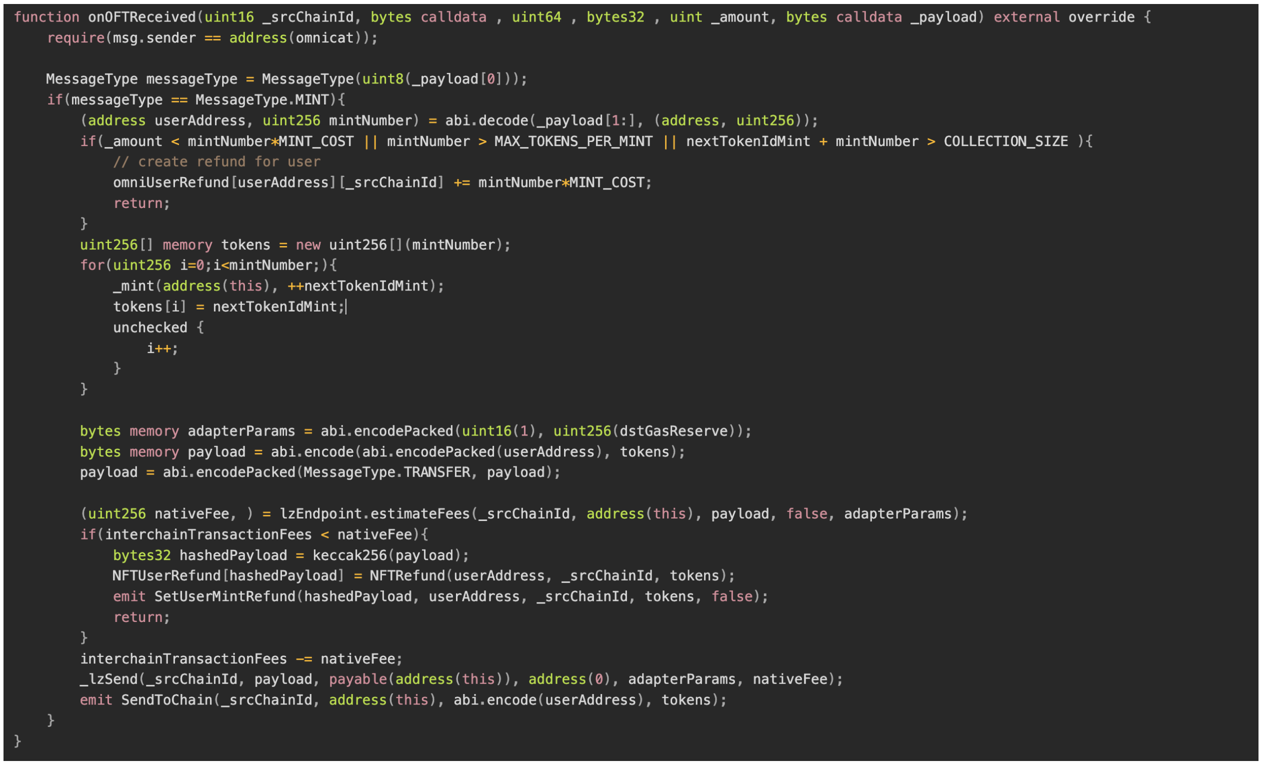
Resolution: Fixed

5.3.7. MAX_MINTS_PER_ACCOUNT can be bypassed

Severity: Medium Risk

Context: OmniNFTA.sol#L164

Description: There is no check in the OmniNFTA::onOFTReceived to prevent users from minting above the MAX_MINTS_PER_ACCOUNT and now is applied only to OmniNFT::mint. That way users can simply mint from different chains up to avoid the limitation of gaining an unfair advantage. Another possible way for a user to mint more is to intentionally fail the transaction before NFTs are minted in OmniNFT::_nonblockingLzReceive so he can freely retry it later, this can be performed until the desired number of tokens are minted in Blast contract.



```
function onOFTReceived(uint16 _srcChainId, bytes calldata , uint64 , bytes32 , uint _amount, bytes calldata _payload) external override {
    require(msg.sender == address(omnicat));

    MessageType messageType = MessageType(uint8(_payload[0]));
    if(messageType == MessageType.MINT){
        (address userAddress, uint256 mintNumber) = abi.decode(_payload[1:], (address, uint256));
        if(_amount < mintNumber*MINT_COST || mintNumber > MAX_TOKENS_PER_MINT || nextTokenIdMint + mintNumber > COLLECTION_SIZE ){
            // create refund for user
            omniUserRefund[userAddress][_srcChainId] += mintNumber*MINT_COST;
            return;
        }
        uint256[] memory tokens = new uint256[](mintNumber);
        for(uint256 i=0;i<mintNumber;){
            _mint(address(this), ++nextTokenIdMint);
            tokens[i] = nextTokenIdMint;
            unchecked {
                i++;
            }
        }

        bytes memory adapterParams = abi.encodePacked(uint16(1), uint256(dstGasReserve));
        bytes memory payload = abi.encode(abi.encodePacked(userAddress), tokens);
        payload = abi.encodePacked(MessageType.TRANSFER, payload);

        (uint256 nativeFee, ) = lzEndpoint.estimateFees(_srcChainId, address(this), payload, false, adapterParams);
        if(interchainTransactionFees < nativeFee){
            bytes32 hashedPayload = keccak256(payload);
            NFTUserRefund[hashedPayload] = NFTRefund(userAddress, _srcChainId, tokens);
            emit SetUserMintRefund(hashedPayload, userAddress, _srcChainId, tokens, false);
            return;
        }
        interchainTransactionFees -= nativeFee;
        _lzSend(_srcChainId, payload, payable(address(this)), address(0), adapterParams, nativeFee);
        emit SendToChain(_srcChainId, address(this), abi.encode(userAddress), tokens);
    }
}
```

Recommendation: The most easy-to implement solution will be to have a mapping per address and number of minted tokens in the OmniNFTA contract, since all the new mints happen there.

Resolution: Acknowledged

5.4. Low/Info severity

5.4.1. Lack of pausing mechanism

Severity: Low Risk

Context: OmniNFTA.sol, OmniNft.sol, OmniNFTBase.sol

Description: `OmniNFT` contracts do not use any pausable mechanism and have no way to react when a critical situation happens. Although system utilizes `LayerZero` that have such mechanism, it will not be useful in situations when for example the owner of the project has given wrong configuration, either for `LZ` or the `NFTs` itself. No direct issue from the absence of such a mechanism can be observed but in general, all the important functions should be protected with `whenNotPaused` modifier.

Recommendation: Inherit `Pausable` from `OpenZeppelin`, apply the `whenNotPaused` modifier and do not forge to expose `onlyOwner` external `pause/unpause` functions.

Resolution: Fixed

5.4.2. OmniNFTA::onOFTReceived uses single dstGasReserve for all the chains

Severity: Low Risk

Context: OmniNFTA.sol#L178

Description: Single `dstGasReserve` is being used, no matter what is the `_srcChainId`, this can cause issues because different chains are being passed, and their `LzApp::minDstGasLookup` and the gas needed on the destination for the message to be processed are different.

Now, suppose there are failures on one chain. In that case, you will have to increase the `dstGasReserve` for all the others also, potentially depleting the `interchainTransactionFees` as the excess provided fees will not be refunded because `estimateFees` uses the same function to calculate the gas as `UltraLightNodeV2::send` and the unused amount will be lost on the destination chain.

Recommendation: One option is to have a mapping with the `dstGasReserve` for each one of the chains and configure only the particular `_srcChainId` record.

Resolution: Fixed

5.4.3. Unnecessary else clause

Severity: Low Risk

Context: OmniNFTBase.sol#L117

Description: `OmniNFTBase.sol::tokenURI` has logic to reveal the true URI of the NFT. But the else statement can be removed and leave only `super.tokenURI(tokenId);` as inside the if `return` is used.

Resolution: Fixed