Exec Sum

A time-boxed security review of the Ark Lane protocol was done by **Antoine**, **Erim** and **Credence**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where I try to find as many vulnerabilities as possible. I 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.

About us

Antoine: I'm an independant security researcher for the Starknet ecosystem. Find me on Twitter@Meckerrr

Erim: Independent security researcher for more than 4 years. Twitter@0xerim

Credence: I'm an independant security researcher for the Starknet ecosystem. Find me on Twitter @credence0x

About Ark_lane

The Starklane NFT Bridge: seamless transfer of NFTs between ETH L1 & Starknet L2.

Observations

Threat Model

Privileged Roles & Actors

Starklane admin. Starklane admin is able to upgrade the contracts. The admin could potentially upgrade the contracts and introduce a vulnerability.

Security Interview

Severity classification - OWASP

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/Difficulty - likelihood or difficulty is a rough measure of how likely or difficult this particular vulnerability is to be uncovered and exploited by an attacker.

Severity - the overall criticality of the risk

Security Assessment Summary

review commit tag - audit-2024-02-17

Scope

The following smart contracts were in scope of the audit:

Cairo

All but erc721_brigeable.cairo

Solidity

All but ERC721Bridgeable and ERC1155Bridgeable.

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

- Critical & High: 2 issues
- Medium: 5 issues
- Low: 4 issues

Findings Summary

ID	Title	Severity	Status
[C-01]	Anyone can withdraw any token held by the L1 bridge	Critical	Fixed
[C-02]	Impossible to withdraw L1 native tokens back on L1 after bridging to L2. Also, a different collection address will be generated for every native l2 token bridged to l1	Critical	Fixed
[M-01]	Unchecked parameter	Medium	Fixed
[M-02]	Unchecked parameter	Medium	Fixed
[M-03]	Unchecked parameter	Medium	Fixed
[M-04]	Access control issue cancellation	Medium	Fixed
[M-05]	Should use OZ Initializabl	Medium	Fixed
[L-01]	Matching pattern bug	Low	Fixed
[L-02]	Remove TODO	Low	Fixed
[L-03]	Wrong selector	Low	Fixed
[L-04]	Whitelist indexer	Low	Fixed
[I-01]	Useless function	Informational	Fixed
[I-02]	Confusing variable name	Informational	Fixed
[I-03]	Should use OZ Initializable contract	Informational	Fixed

Detailed Findings

[C-01] Anyone can withdraw any token held by the L1 bridge

Severity - Critical

Impact: HIGH

Likelihood: HIGH

Description

contract: `Bridge.sol` function: `withdrawTokens` & `_consumeMessageAutoWithdraw`

An attacker can carefully craft a request such that they will be able to withdraw any token being held by the `Bridge.sol` contract. This can be done by setting `WITHDRAW_AUTO` in request header param to true and the setting all other values to bypass other restrictions. The reason this will work is because the `_consumeMessageAutoWithdraw` function does not confirm that the request originated from an l2 request.

The error originates here:

```
if (Protocol.canUseWithdrawAuto(header)) {
    _consumeMessageAutoWithdraw(_starklaneL2Address, request);
} else {
    _consumeMessageStarknet(_starknetCoreAddress, _starklaneL2Address, r
}
```

```
    function _consumeMessageAutoWithdraw(
        snaddress fromL2Address,
        uint256[] memory request
    )
        internal
    {
        bytes32 msgHash = keccak256(
            abi.encodePacked(
                snaddress.unwrap(fromL2Address),
                uint256(uint160(address(this))),
                request.length,
                request)
        );

        uint256 status = _autoWithdrawn[msgHash];

        if (status == WITHDRAW_AUTO_CONSUMED) {
            revert WithdrawAlreadyError();
        }

        _autoWithdrawn[msgHash] = WITHDRAW_AUTO_CONSUMED;
    }
```

We would expect that the `_consumeMessageAutoWithdraw` function to consume the message from starknet messaging but it is doesn't! So no check is actually performed on the request to confirm that a message was sent from l2 so an attacker could simply pass a `request` into the `withdrawTokens` endpoint that didn't originate from L2

Recommendations

We suggest the l2 message is consumed, whether or not it is an auto withdraw transaction

```
starknetCore.consumeMessageFromL2(snaddress.unwrap(fromL2Address), reque
if(autowithdraw) then
    do stuff
```

[C-02] Impossible to withdraw L1 native tokens back on L1 after bridging to L2. Also, a different collection address will be generated for every native l2 token bridged

to l1

Severity - Critical

Impact: HIGH

Likelihood: HIGH

Description

contract: `Bridge.sol` function: `withdrawTokens` & `_verifyRequestAddresses`

The issue can be explained like so: Lets say you own `EVERAI #51` and you bridge to Starknet, you'll never be able to bridge back. Also, lets assume you own both `Blobert #30` and `Blobert #31` on starknet, and you decide to bridge both in two separate transactions, the bridge would create 2 different collection addresses for each time you bridge.

Both issues stem from the fact that `_l2ToL1Addresses` storage variable is never updated anywhere in the l1 contracts. Let's take a look at these code blocks.

```
address collectionL1 = _verifyRequestAddresses(req.collectionL1, req.coll
CollectionType ctype = Protocol.collectionTypeFromHeader(header);
if (collectionL1 == address(0x0)) {
    if (ctype == CollectionType.ERC721) {
        collectionL1 = _deployERC721Bridgeable(req.name, req.symbol, req
    } else {
        // TODO ERC1155.
    }
}
```

```
    function _verifyRequestAddresses(
        address collectionL1Req,
        snaddress collectionL2Req
    )
        internal
        view
        returns (address)
    {
        address l1Req = collectionL1Req;
        uint256 l2Req = snaddress.unwrap(collectionL2Req);
        address l1Mapping = _l2ToL1Addresses[collectionL2Req];
        uint256 l2Mapping = snaddress.unwrap(_l1ToL2Addresses[l1Req]);

        if (l2Req > 0 && l1Req == address(0)) {
            if (l1Mapping == address(0)) {
                return address(0);
            } else {
                return l1Mapping;
            }
        }

        if (l2Req > 0 && l1Req > address(0)) {
            if (l1Mapping != l1Req) {
                revert InvalidCollectionL1Address();
            } else if (l2Mapping != l2Req) {
                revert InvalidCollectionL2Address();
            } else {
                return l1Mapping;
            }
        }

        revert ErrorVerifyingAddressMapping();
    }
```

`l1Req` will always be `address(0)` when the token being bridged from l2 is a native l2 token. we also then explect that `l1Mapping` should be `address(0)` the first time a native l2 token is bridged to ethereum and should subsequently be the new address gotten from `_deployERC721Bridgeable`. This is however not the case as `_l2ToL1Addresses` storage variable is never updated so it always return `address(0)`

```
        if (l2Req > 0 && l1Req == address(0)) {
            if (l1Mapping == address(0)) {
                return address(0);
            } else {
                return l1Mapping;
            }
        }
```

Also `l1Req` will be the address of the l1 collection contract when the token is a native l1 token and so has previously been bridged to starknet. However, since `_l2ToL1Addresses` storage variable is never updated, the following code block will always revert in such cases

```
    if (l2Req > 0 && l1Req > address(0)) {
        if (l1Mapping != l1Req) {
            revert InvalidCollectionL1Address();
        } ...
    }
```

Recommendations

The `_verifyRequestAddresses` is a bit complicated at the moment and we think it should be made simpler. We can always trust `l1Req` value since it always comes from l2 bridge contract. So, if `l1req` value is set, we use that as the l1 address. it will always be set when token is a native l1 token. where it isn't, token isn't l1 native so we check whether `_l2ToL1Addresses[collectionL2Req]` is present. if it is, then return it, else deploy a new erc721 bridgeable contract and update `_l2ToL1Addresses`

[M-01] Unchecked parameter

Severity - Medium

Description

contract: `Bridge.sol` function: `depositTokens`

The function expects the parameter `ownerL2` to be of type `snaddress`. The problem is that `snaddress` is a wrapper around uint256 and its size is only checked if the custom `snaddressWrap` function is called. It is not the case here. `ownerL2` could potentially be higher than the felt prime number which will case trouble on L2, the message will never arrive.

Recommendations

Add checks to ensure `ownerL2` is indeed a `felt252`. You can either pass a `uint256` and wrap it or keep the parameter as is and simply add a `isFelt252` require.

[M-02] Unchecked parameter

Severity - Medium

Description

contract: `State.sol` function: `setStarklaneL2Selector`

The function expects the parameter `l2Selector` to be of type `felt252`. It is called in the initializer with `setStarklaneL2Selector(Cairo.felt252Wrap(starklaneL2Selector))`. If a conversion is performed here and it will work perfectly fine, the function is external and could potentially be called elsewhere. There is no check to ensure the value is indeed a felt. It is better to wrap the value inside the function.

Recommendations

Add a `isFelt252` require. Pass the param as `uint256` and wrap it inside the function.

[M-03] Unchecked parameter

Severity - Medium

Impact:

Likelihood:

Description

contract: `State.sol` function: `setStarklaneL2Address`

The function expects the parameter `l2Address` to be of type `snaddress`. It is called in the initializer with `setStarklaneL2Address(Cairo.snaddressWrap(starklaneL2Address))` . If a conversion is performed here and it will work perfectly fine, the function is external and could potentially be called elsewhere. There is no check to ensure the value is indeed a felt. It is better to wrap the value inside the function.

Recommendations

Add a `isFelt252` require. Pass the param as `uint256` and wrap it inside the function.

[M-04] Access control issue cancellation

Severity - Medium

Impact: High

Likelihood: Low

Description

contract: `Bridge.sol` function: `startRequestCancellation(uint256[] memory payload, uint256 nonce)`

The function has the modifier `onlyOwner` . Only the owner is able to initiate cancellation while any user must be able to do that.

Recommendations

Remove the modifier like in `cancelRequest()` .

[L-01] Matching pattern bug

Severity - Low

Impact: Medium

Likelihood: Medium

Description

contract: `collection_manager.cairo` function: `token_uri_from_contract_call`

There is an issue regarding how the snake_case versus camelCase is handled. It is not a coding issue, the problem comes from StarknetOS. The function implements this:

```
match starknet::call_contract_syscall(collection_address, token_uri_sele
    Result::Ok(span) => span.try_into(),
    Result::Err(e) => {
        match starknet::call_contract_syscall(
            collection_address, tokenUri_selector, calldata,
        ) {
            Result::Ok(span) => span.try_into(),
            Result::Err(e) => { Option::None }
        }
    }
}
```

The StarknetOS doesn't return an `Err` if a syscall fails. It will panic everytime. Hence, here, if the contract doesn't implement the snake_case entrypoint, it will never call the camelCase entrypoint :/.

https://github.com/OpenZeppelin/cairo-contracts/issues/904

Recommendations

There is no way to handle errors on starknet at the moment so we could wait for the 0.15 update.

Alternatively, we can pass the token uri selector as a parameter to the `deposit_tokens` function then verify that the token uri param is either `selector!("token_uri")` or `selector!("tokenURI")`

[L-02] Remove TODO

Severity - Low/Informational

Description

Accross the repo, there are a lot of TODO ERC1155. It is better to remove all that and only handle ERC721 scenarios. The contracts are upgradeable so it ould be both cleaner and safer. A lot of TODOs dont revert with errors.

[L-03] Wrong selector

Severity - Low

Description

contract: `collection_manager.cairo` function: `token_uri_from_contract_call`

The tokenUri variable is wrong. `selector("tokenUri")` is used instead of `selector("tokenURI")`. selector("tokenUri") = `0x0362dec5b8b67ab667ad08e83a2c3ba1db7fdb4ab8dc3a33c057c4fddec8d3de` selector("tokenURI") = `0x012a7823b0c6bee58f8c694888f32f862c6584caa8afa0242de046d298ba684d`

Recommendations

There is a `selector!` macro.

[L-04] Whitelist indexer

Severity - Low/Informational

Description

We would be more comfortable with the function `withdrawTokens` checking that `msg.sender` is the Arklane indexer in case of autowithdraw.

[I-01] Useless function

Severity - Low/Informational

Description

contract: `Messaging.sol` function: `addMessageHashForAutoWithdraw(uint256 msgHash)`

Function not used and should be removed.

[I-02] Confusing variable name

Severity - Informational

Description

contract: `collection_manager.cairo` function: `verify_collection_address`

Confusing variable name. This might lead to bugs in future upgrades.

Recommendations

You could use `l2_storage` instead of `l2_bridge`.

[I-03] Should use OZ Initializable instead of custom implem

Severity - Medium

Description

UUPS Upgradeable proxy pattern is used extensively in Bridge contract and when dealing with upgradeable contracts, it matters that all calls to constructor should now be done inside an `initializer` contract. while many contracts do not have an initializer function and work properly, it is adviseable that they do to prevent future errors.
https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable