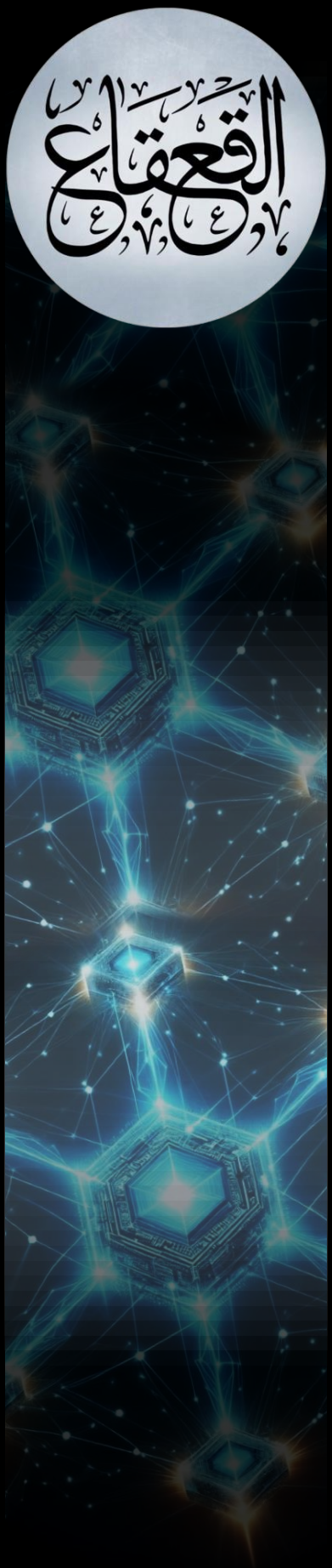
Al-Qa'qa’



Khuga Labs

Auditing Report

Auditor: Al-Qa'qa’

16 May 2025

Table of Contents

1 Introduction

1.1 About Al-Qa'qa'

Al-Qa'qa' is an independent Web3 security researcher specializing in smart contract audits. He has completed more than 10 Private audit, in addition to some Team audits, as well as placing top 1 and top 3 in multiple contests on sherlock and cantina.

He has experience in core EVM architecture, Solana/Rust, geth. Full Portfolio https://github.com/Al-Qa-qa/audits

For Private audits or consulting requests please reach out via telegram @al_qa_qa, or twitter @Al_Qa_qa

1.2 About Khuga

Khuga is a game where players can join and kill bosses. The players score increases by killing more bosses. The game is Blockchain based. And when the player kill a boss, he can mint an NFT for the boss he killed.

1.3 Disclaimer

Security review cannot guarantee `100%` the protocol's safety. In the Auditing process, we try to identify all possible issues, and we cannot be sure if we missed something.

Al-Qa'qa' is not responsible for any misbehavior, bugs, or exploits affecting the audited
code or any part of the deployment phase.

And change to the code after the mitigation process, puts the protocol at risk, and should be audited again.

1.4 Risk Classification

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

1.4.1 Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality
- Low - Funds are **not** at risk

1.4.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align or little-to-no incentive

2 Executive Summary

2.1 Overview

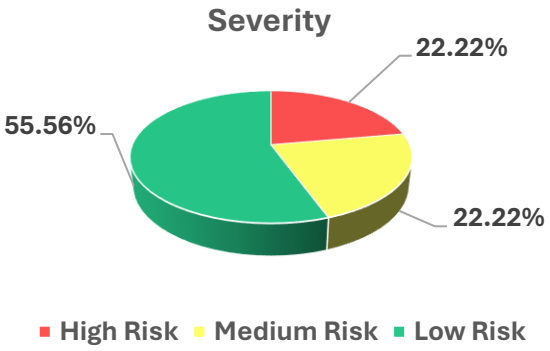
Project	**Khugabash**
Repository	khugabash-smartcontract
Commit Hash	afb2c4a8d2cec20c79d477ec3fdc004707f69478
Mitigation Hash	cb27f7f56cf854927cbd68a25ee9b846d318edc3
Audit Timeline	01 May 2025 to 03 May 2025

2.2 Scope

- src/KhugaBash.sol
- src/KtridgeNFT.sol
- src/Proxy.sol

2.3 Issues Found

Severity	Count
High Risk	2
Medium Risk	2
Low Risk	5



3 Findings Summary

ID	Title	Status
H-01	Players can use `mintKtridge signature` to acquire First position permanently	Fixed
H-02	Player Score will be updated incorrectly, because of lacking arranging	Fixed
M-01	signature length is not checked allows double signature usage	Fixed
M-02	`s` value is not checked, allows double using signature	Fixed
L-01	`KtridgeNFT::totalSupply()` returns incorrect value	Fixed
L-02	`KtridgeNFT::burnKtridge` will not delete `playerBossToToken` if the token transfered	Acknowledged
L-03	Signature schema is too weak	Acknowledged
L-04	The Approval of the token can't burn it	Fixed
L-05	Storage Collision can occur when upgrading `KhugaBash` contract	Fixed
I-01	`KhugaBash::addBoss()` is not checking `bytes32(0)` bossId input	Fixed
I-02	`KhugaBash::getTopPlayers()` sorting is a selection sort not a Heap sort	Fixed
I-03	`TierNameSet` event is not triggered at initialization	Fixed
I-04	Single-step ownership transfer mechanism by `Ownable`	Fixed

4 Findings

4.1 High Risk

4.1.1 Players can use `mintKtridge signature` to acquire First position permanently

context: KhugaBash.sol#L319

Description:
The signature used for `syncData()` encode packed the sender, with bosses Ids provided, with the score.

KhugaBash.sol#L319

```
    function syncData( ... ) external nonReentrant {
        ...
        // Check signature
>>      bytes32 messageHash = keccak256(abi.encodePacked(msg.sender, _bossIds,
score));
        ...
    }
```

We use `encodePacked`, which just puts the array values. The problem is that the user can provide empty array as `_bossIds` inputs. So when encoding packed, we will just encode `msg.sender` and `score` value.

For `mintKtridge()` function, which is used to mint an NFT to the player, we provide the signature is created using player address and the bossId the player killed to claim the NFT.

KhugaBash.sol#L380-L382

```
    function mintKtridge( ... ) external nonReentrant {
        ...

        // check signature
>>      bytes32 messageHash = keccak256(
            abi.encodePacked(msg.sender, bossId)
        );
        ...
    }
```

So the messageHash created for `mintKtridge()` (Player, bossId). Will be the same hash for `syncData()` (Player, <empty_bossIds_array>, score). This allow the Player to use the `mintKtridgesignature` at `syncData`.

Proof of Concept

- The Player has Killed bossId, its value is a large Number, lets say `1,000,000,000`
- The Player signed for `syncData`, and called it, so we stored he killed the Boss, and his score is `1000` now
- The Player signed for `mintKtridge`

- The Player used this signature, and provided it for `syncData()` providing empty `_bossIds` array, and the score value as `bossId` value.
- The message after encoding will be the same format as `mintKtridge`. And hashing it will be the same as `mintKtridge`, so the validation will success.
- The Player score will go up to reach the value of bossId.
- For large bossId values, if it is a hashing of its Metadata, it will be too huge number, the Player will make his score too large, making him dominate the leaderboard and stays a the top forever.

Recommendations:
Encode the length of the array with the arrayInputs too, so if the array is empty, we encode its length too. preventing collision with `mintKtridge` signature

Status: Fixed at commit [9f74c2f34aa6315bcb9e1a96c663da4fb06e8f7e]

4.1.2 Player Score will be updated incorrectly, because of lacking arranging

context: KhugaBash.sol#L327-L330

Description:
When calling `syncData()` we update the score whenever it is changed, either it is increased or decreased.

KhugaBash.sol#L327-L330

```
    function syncData( ... ) external nonReentrant {
        ...

        // Update player score only if different
>>      if (players[msg.sender].score != score) {
            players[msg.sender].score = score;
            emit LeaderboardUpdated(msg.sender, score);
        }

        ...
    }
```

The problem is that the contract can't know if this signature State is the newest one or not. So Old signatures can override the new ones.

Proof of Concept

- Player score is `0`
- Player killed boss with id `10`, and his score is `100`
- He signed a sig for this state (player, [10], 100) <-- Sig1
- Player killed another two bosses with ids `[20, 30]`, and his score is `400`
- He signed a sig for this state (player, [20, 30], 400) <-- Sig2
- Player called `syncData()` providing `Sig2`
- Update player state: killed bosses: [20, 30] | score: 400
- Player called `syncData()` providing `Sig1`
- Update player state: killed bosses: [10, 20, 30] | score: 100
- The score will be incorrect at this case and will be overridden by old values.

The problem is that there is no information when this signature for that score was signed. so old signatures can override new ones. making the score update goes incorrectly.

Recommendations:

- Adding Bosses killed to the player can be made without arrange as it is just filling items into the array. We can include another parameter in the signature called `timestamp`, this will represent the time by UNIX at which the backendSigner signed the message.
- We will make a new mapping called `playerLastScoreUpdated` which takes player address as key, and stores this `timestamp` as value.
- When calling `syncData()` we will only update the score if the provided `timestamp` at the signature is greater than that in `playerLastScoreUpdated`
- This will make synchronization process goes correct, as the score will only be updated if it is represents values after the current stored ones

Status: Fixed at

commit [9f74c2f34aa6315bcb9e1a96c663da4fb06e8f7e](), [8a883112e08b38bae827c4cb842ccbb6f283b28c]()

4.2 Medium Risk

4.2.1 signature length is not checked allows double signature usage

context: Solady->SignatureCheckerLib::isValidSignatureNowCalldata

Description:
The protocol uses Solady libraries 0.1.12v. It depends on `SignatureCheckerLib::isValidSignatureNowCalldata()` to check the validity of the signature, and once it is consumed it makes it as consumed to not be used again. This mechanism is used is `registerPlayer()`, `syncData()`, and `mintKtridge()` functions.

KhugaBash.sol#L316-L324

```
    function syncData( ... ) external nonReentrant {
        ...
        bytes32 signatureHash = keccak256(signature);
>>      if (usedSignatures[signatureHash]) revert SignatureAlreadyUsed();

        // Check signature
        bytes32 messageHash = keccak256(abi.encodePacked(msg.sender, _bossIds,
score));
        if (!backendSigner.isValidSignatureNowCalldata(messageHash, signature))
            revert InvalidSignature();

        // Mark signature as used
>>      usedSignatures[signatureHash] = true;

        ...
    }
```

The problem is that the signature is provided as `bytes` and it is known weather it is used or not by hashing it using `keccak256`. The EAO signature consists of `r,s,v`, which is 65 length. The problem is that the library accepts `64` signature lengths as well as `65` signature lengths. where if the signature length provided is `64` the library adds the `v` value it self of the data making it's value `27`.

SignatureCheckerLib.sol#L92-L101

```
    function isValidSignatureNowCalldata(address signer, bytes32 hash, bytes
calldata signature) ... {
        if (signer == address(0)) return isValid;
        /// @solidity memory-safe-assembly
        assembly {
            let m := mload(0x40)
            for {} 1 {} {
                if iszero(extcodesize(signer)) {
                    switch signature.length
>>                  case 64 {
                        let vs := calldataload(add(signature.offset, 0x20))
                        mstore(0x20, add(shr(255, vs), 27)) // `v`.
                        mstore(0x40, calldataload(signature.offset)) // `r`.
                        mstore(0x60, shr(1, shl(1, vs))) // `s`.
                    }
>>                  case 65 {
                        mstore(0x20, byte(0, calldataload(add(signature.offset,
0x40)))) // `v`.
                        calldatacopy(0x40, signature.offset, 0x40) // `r`, `s`.
                    }
                   ...
    }
```

This will allow users to double use the signature of the backend signer. where if he signed it with `v` value `27`. The user can provide two valid signatures now.

- The first one is 65 length bytes containing the `v` value
- The second one is 64 length bytes not containing `v` value

Both of them will be treated as valid signatures, allows the users to double spend them in `syncData()`, and the other two functions.

Recommendations:
Use OpenZeppelin SignatureChecker library, it just allows `65` signature length for EOA signatures

Status: Fixed at commit 57fdd6ce06505d5a0340ce3ab7ecfa141d200daa

4.2.2 `s` value is not checked, allows double using signature

context: Solady->SignatureCheckerLib::isValidSignatureNowCalldata

Description:
The protocol uses Solady libraries 0.1.12v. It depends on `SignatureCheckerLib::isValidSignatureNowCalldata()` to check the validity of the signature, and once it is consumed it makes it as consumed to not be used again. This mechanism is used is `registerPlayer()`, `syncData()`, and `mintKtridge()` functions.

KhugaBash.sol#L316-L324

```
    function syncData( ... ) external nonReentrant {
        ...
        bytes32 signatureHash = keccak256(signature);
>>      if (usedSignatures[signatureHash]) revert SignatureAlreadyUsed();

        // Check signature
        bytes32 messageHash = keccak256(abi.encodePacked(msg.sender, _bossIds,
score));
        if (!backendSigner.isValidSignatureNowCalldata(messageHash, signature))
            revert InvalidSignature();

        // Mark signature as used
>>      usedSignatures[signatureHash] = true;

        ...
    }
```

The problem is that the library is not checking the value of `s` is not checked, weather it is on the lower of the upeer of the curve. Per EIP-2 the value of `s` is forced to be at the lower half order. This is for the Nodes. But `ecrecover()` it is still accepting `s` values from the upper order. So the user can provide value `s` value greater than `n/2` and he will be able to double spend the signature by providing new S avalue and flipping the `v`

- s' = n - s
- v' = flip v, $_{27}$ -> 28

This will allow users to double spend the signature

SignatureCheckerLib.sol#L23

```
...
>>/// This implementation does NOT check if a signature is non-malleable.
library SignatureCheckerLib {
    ...
}
```

Both of them will be treated as valid signatures, allows the users to double spend them in `syncData()`, and the other two functions.

Recommendations:
Use OpenZeppelin SignatureChecker library, it checks for `s` value before recovery.

Status: Fixed at commit 57fdd6ce06505d5a0340ce3ab7ecfa141d200daa

4.3 Low Risk

4.3.1 `KtridgeNFT::totalSupply()` returns incorrect value

context: KtridgeNFT.sol#L185-L187

Description:
The NFT Collection totalSupply value returns the value of `_tokenIdCounter`.

KtridgeNFT.sol#L185-L187

```
    function totalSupply() public view returns (uint256) {
        return _tokenIdCounter;
    }
```

This value should store the totalSupply of the NFTs minted as it is increased before minting any token by calling `mintKtridge`, so it should result in the real number of NFTs in the circulation.

KtridgeNFT.sol#L221-L222

```
    function mintKtridge(
        address player,
        bytes32 bossId
    ) external onlyKhugaBash returns (uint256) {
        ...
        // Mint new token
>>      _tokenIdCounter++;
        uint256 tokenId = _tokenIdCounter;

        _mint(player, tokenId);
        ...
    }
```

The problem is that the NFT can be burnt, and removed from cirulation by calling `burnKtridge`. So, if the NFT is burning, this means the totalSupply decreased by `1`, and so on. But the Total Supply will still return `_tokenIdCounter` as the totalSupply of the NFTs, which is incorrect.

Recommendations:

- Add another variable named `_burnedTokensCounter` for example, and increase it when an NFT burnet.
- Then, at `totalSupply`, make the return value as `_tokenIdCounter - _burnedTokensCounter`

Status: Fixed at commit ae0ac3f27133c602a987b69e421cfc3018ddc244

4.3.2 `KtridgeNFT::burnKtridge` will not delete `playerBossToToken` if the token transfered

context: KtridgeNFT.sol#L257

Description:
When minting an NFT we mint it to the player and store it in the mapping too.

KtridgeNFT.sol#L224-L228

```
    function mintKtridge( ... ) external onlyKhugaBash returns (uint256) {
        ...
        // Mint new token
        _tokenIdCounter++;
        uint256 tokenId = _tokenIdCounter;

>>      _mint(player, tokenId);

        // Record token information
        tokenToBoss[tokenId] = bossId;
>>      playerBossToToken[player][bossId] = tokenId;
        ...
    }
```

When burning it, we check for the owner and burn it, then remove it from `playerBossToToken` using the owner of the NFT as the `player`.

KtridgeNFT.sol#L257

```
    function burnKtridge(uint256 tokenId) external {
>>      address tokenOwner = _ownerOf(tokenId);
        ...
>>      delete playerBossToToken[tokenOwner][bossId];

        emit KtridgeBurned(tokenOwner, bossId, tokenId);
    }
```

This is not `100%` correct, the owner of the NFT can be another address than the player. Where the player can sell the NFT on the market. or transfer it to another address address. So when burning this will not reset `playerBossToToken` value.

Recommendations:
We can use `_beforeTokenTransfer` to change `playerBossToToken` value before transferring. the following check should be implemented in it.

- Only works in transferring, if burn/mint skip it
- Should prevent transferring if the receiver already has a value at `playerBossToToken`. So that to not override it
- Delete the `playerBossToToken` from the sender and add it to the receiver

Status: Acknowledged

4.3.3 Signature schema is too weak

context: KhugaBash.sol#L45

Description:
The current signature schema depends only on the Game parameters. including Players, score, BossId(s). This structure is weak and can result in a lot of issues.

- The schema lacks to include the address of `KhugaBash`. So if the address changed, the old signatures can be used contract
- The schema lacks to include `chainId`. So if the Game is to be deployed on two chains, the signature can be used from one chain to another chain

Recommendations:
Use EIP-712 signatures. they are robust signature schema that mitigate most of signature replaying attacks that can occur.

Status: Acknowledged

4.3.4 The Approval of the token can't burn it

context: This issue has been introduced when migrating to OpenZeppelin fixing issue M-01, M-02

Description
When burning the token we only check if the sender is the owner of the token, or the operator for that address (can control all that address tokens), but there is no check weather the sender is an approval address for that specific token.

```
    function burnKtridge(uint256 tokenId) external {
        address tokenOwner = _ownerOf(tokenId);

        // Only token owner or approved address can burn
        if (
            tokenOwner != msg.sender &&
            !isApprovedForAll(tokenOwner, msg.sender)
        ) {
            revert NotAuthorizedToBurn();
        }

        ...
    }
```

This will prevent the token approval address to burn it.

Recommendations\

Add another check to see if the sender is the approval address

```
         if (
             tokenOwner != msg.sender &&
+            getApproved(tokenId) != msg.sender &&
             !isApprovedForAll(tokenOwner, msg.sender)
         ) {
             revert NotAuthorizedToBurn();
```

Status: Fixed at commit d38f6a1f4b02cddeb5e32e8de97ec05fa44f05ae

4.3.5 Storage Collision can occur when upgrading `KhugaBash` contract

context: This issue has been introduced when migrating to OpenZeppelin fixing issue M-01, M-02

Description

Solday library is not storing the variables in default EVM Slots [0, 1, 2, ...]. They have a custom slots for the varaibles. This includes Ownable lib and ReentrancyGuard.

When upgrading to OpenZeppelin, we upgraded Ownable to the upgrade version of it. But ReentrancyGuard is not upgraded.

```
import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
...
contract KhugaBash is
    Initializable,
    Ownable2StepUpgradeable,
>>  ReentrancyGuard,
    UUPSUpgradeable
{
```

This will make the implementation contract storage variables stored as following:

- slot(0) -> (The status of the contract weather it is entered or not)
- slot(1) -> backendSigner
- slot(2) -> ...

This is incorrect as if we made an upgrade and changed the implementation removing (ReentrancyGuard), or replacing it with the upgradable version. The new implementation will start storing variables with `slot(0)`, results in storage collision of the variables.

Recommendations

When using upgradable contracts, it is crucial to leave a gap between contracts, or store the variables in random storage slots. OpenZeppelin upgradable version uses `random storage slots`. We can use the upgradable version of the reentrancy guard.

```
diff --git a/src/KhugaBash.sol b/src/KhugaBash.sol
-import "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
 import "@openzeppelin/contracts/utils/cryptography/SignatureChecker.sol";
 import "@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol";
+import "@openzeppelin/contracts-
upgradeable/utils/ReentrancyGuardUpgradeable.sol";
...
 contract KhugaBash is
     Initializable,
     Ownable2StepUpgradeable,
-    ReentrancyGuard,
+    ReentrancyGuardUpgradeable,
     UUPSUpgradeable
 {
     using SignatureChecker for address;
```

Status: Fixed at commit [cb27f7f56cf854927cbd68a25ee9b846d318edc3]

4.4 Informational Findings

4.4.1 `KhugaBash::addBoss()` is not checking `bytes32(0)` bossId input

context: KhugaBash.sol#L137-L144

Description:
When adding a new boss at `KhugaBash`, there is no check if the bossId is `0` or not.

KhugaBash.sol#L137-L144

```
    function addBoss(bytes32 bossId) external onlyOwner {
        if (bossExists[bossId]) revert BossAlreadyExists();

        bossExists[bossId] = true;
        allBosses.push(bossId);

        emit BossAdded(bossId);
    }
```

If bossId is `0`. This will result in incorrect value by `KtridgeNFT::tokenURI()`, where any token will be treated as valid token when getting the bossId, as the default value for `tokenToBoss` for unsetted tokens is `bytes32(0)`

KtridgeNFT.sol#L275

```
    function tokenURI( ... ) public view override returns (string memory) {
>>      bytes32 bossId = tokenToBoss[tokenId];
        BossMetadata memory metadata = bossMetadata[bossId];

        if (!metadata.exists) revert BossMetadataNotSet();
        ...
        return string(abi.encodePacked("data:application/json;base64,", json));
    }
```

This will make any `tokenURI()` return valid for any token not created yet. if there is a boss added with Id equal `0`.

Recommendations:
Prevent adding `bossId` with `0` value at `KhugaBash::addBoss()`

Status: Fixed at commit c69d77c9eddd3b3b7550fcac0939f02aaa896aff

4.4.2 `KhugaBash::getTopPlayers()` sorting is a selection sort not a Heap sort

context: KhugaBash.sol#L193-L213

Description:
It is written as a comment for `KhugaBash::getTopPlayers()` that the sorting is done using Heap Sort. But this is incorrect. as the acctuall implementation uses selection sort, not the Heap sort. Where we loop through all values, then we loop through them again to get the Higest value and store it, it is the selection sort algorism.

KhugaBash.sol#L193-L213

```
    function getTopPlayers( ... ) external view returns (LeaderboardEntry[]
memory) {
        ...

        // Sort using a simple heap sort approach for the top N players
        // This is more efficient than insertion sort for large arrays
        for (uint256 i = 0; i < resultSize; i++) {
            // Find highest score player among remaining
            uint256 highestIndex = i;
            uint256 highestScore = allPlayers[i].score;

            for (uint256 j = i + 1; j < size; j++) {
                if (allPlayers[j].score > highestScore) {
                    highestIndex = j;
                    highestScore = allPlayers[j].score;
                }
            }

            // Swap if we found a higher score
            if (highestIndex != i) {
                LeaderboardEntry memory temp = allPlayers[i];
                allPlayers[i] = allPlayers[highestIndex];
                allPlayers[highestIndex] = temp;
            }
        }
        ...
    }
```

Recommendations:
Remove the Comment written and write it is the selection sort algorism.

Status: Fixed at commit [136f57ab2dcaa27018c4ef7c9172a13bbb796e42]

4.4.3 `TierNameSet` event is not triggered at initialization

context: KtridgeNFT.sol#L81-L85

Description:
When deploying `KtridgeNFT` we set the Tier names with constant values.

KtridgeNFT.sol#L81-L85

```
    constructor() {
        _initializeOwner(msg.sender);

        // Initialize tier names
>>      tierNames[0] = "Common";
|       tierNames[1] = "Uncommon";
|       tierNames[2] = "Rare";
|       tierNames[3] = "Epic";
>>      tierNames[4] = "Legendary";
    }
```

There is an event called `TierNameSet`. This event should be fired whenevert the tier name set. This can be seen clear when calling `setTierName`

KtridgeNFT.sol#L108-L112

```
    function setTierName(uint8 tier, string calldata _name) external onlyOwner {
        if (tier > 4) revert InvalidTier();
        tierNames[tier] = _name;
>>      emit TierNameSet(tier, _name);
    }
```

Recommendations:
emit `TierNameSet` at construction (it will be emited five times one for each tier).

Status: Fixed at commit: 57fdd6ce06505d5a0340ce3ab7ecfa141d200daa

4.4.4 Single-step ownership transfer mechanism by `Ownable`

context: KtridgeNFT.sol#L13

Description:
Single Step ownership transfer is dangerous as if the transfer is made to an incorrect address. the contract will be with no owner, and the role will be lost forever.

Recommendations:
Use `Ownable2Step` instead

Status: Fixed at commit 57fdd6ce06505d5a0340ce3ab7ecfa141d200daa