Gigaverse Security Review

Pashov Audit Group

Conducted by: unforgiven, Hals, shaflow

January 18th 2025 - January 23th 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 **0xgarnish/gigaverse-contracts** repository was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Gigaverse

Gigaverse is a game where players explore a dungeon-crawling RPG, collecting and spending items stored in a game database, with the ability to import/export items on-chain using the ImportExportSystem, while roles like SERVER_JUDGE (game server) and MANAGER (admin) manage different interactions, and players authenticate via Account Creation, eventually minting accounts based on NFT ownership or ETH. This system connects smart contracts, an indexer, and game services to manage user accounts, game assets, and interactions by recording and retrieving data on-chain and off-chain.

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 - 9ab59658213f842741f3451265659eee39138dcf

fixes review commit hash - e9cf5c47deb344a0e3dc770208502db8d4a10e31

Scope

The following smart contracts were in scope of the audit:

- `ImportExportSystem`
- `AccountSystem`
- `ColumnConstants`
- `RoleConstants`
- `GameRegistry`
- `GameRegistryConsumer`
- `ColumnInitializer`
- `DataStore`
- `DataTable`
- `DataTypes`
- `GameItems`
- `GameNFT`
- `GigaNameNFT`
- `GigaNameNFTBeforeUpdateHandler`
- `GigaNoobNFT`
- `GigaNoobNFTBeforeUpdateHandler`

7. Executive Summary

Over the course of the security review, unforgiven, Hals, shaflow engaged with Gigaverse to review Gigaverse. In this period of time a total of **17** issues were uncovered.

Protocol Summary

Protocol Name	Gigaverse
Repository	https://github.com/0xgarnish/gigaverse-contracts
Date	January 18th 2025 - January 23th 2025
Protocol Type	Game

Findings Count

Severity	**Amount**
High	5
Medium	6
Low	6
Total Findings	**17**

Summary of Findings

ID	Title	Severity	Status
[H-01]	Not updating MINT_COUNT_CID and BURN_COUNT_CID	High	Resolved
[H-02]	Unauthorized access to update() function	High	Resolved
[H-03]	Soulbound tokens cannot be minted or burnt due to an invalid check	High	Resolved
[H-04]	Stuck ETH in GigaNameNFT contract	High	Resolved
[H-05]	Zero Username price when it is minted with mintUsername()	High	Resolved
[M-01]	Missing cleanup of certain registered data when burning an NFT	Medium	Resolved
[M-02]	mintUsername() cannot be called by the minter role only	Medium	Resolved
[M-03]	Max supply enforcement issues	Medium	Resolved
[M-04]	Improper address validation in update function	Medium	Resolved
[M-05]	mintWithEth() and mintUsername() does not return extra ETH	Medium	Resolved
[M-06]	maxSupply check can be bypassed by mintBatch	Medium	Resolved
[L-01]	confirmMint() does not initialize the INITIALIZED_CID	Low	Resolved
[L-02]	Burning Game NFTs will increase the balance of the address(0)	Low	Resolved
[L-03]	burn() functions are not accessible within the system	Low	Resolved

[L-04]	Using _mint() instead of _safeMint()	Low	Resolved
[L-05]	DOS username or account creation	Low	Resolved
[L-06]	mint() does not validate tokenId against username rules	Low	Resolved

8. Findings

8.1. High Findings

[H-01] Not updating `MINT_COUNT_CID` and `BURN_COUNT_CID`

Severity

Impact: High

Likelihood: Medium

Description

The `GameItems` contract provides the `mintBatch` and `burnBatch` functions for batch minting and burning of game items.

```
function mintBatch(
        address to,
        uint256[] memory ids,
        uint256[] memory amounts,
        bytes memory data
    ) external onlyRole(MINTER_ROLE) whenNotPaused {
        _mintBatch(to, ids, amounts, data);
    }

    function mintBatch(
        address to,
        uint256[] memory ids,
        uint256[] memory amounts
    ) external onlyRole(MINTER_ROLE) whenNotPaused {
        _mintBatch(to, ids, amounts, "");
    }

    function burnBatch(
        address from,
        uint256[] memory ids,
        uint256[] memory amounts
    ) external onlyRole(GAME_LOGIC_CONTRACT_ROLE) whenNotPaused {
        _burnBatch(from, ids, amounts);
    }
```

However, during the batch processing, the `MINT_COUNT_CID` and `BURN_COUNT_CID` are not updated, which will lead to incorrect supply calculations.

Recommendations

The logic to handle `MINT_COUNT_CID` and `BURN_COUNT_CID` in batch operations should be added.

[H-02] Unauthorized access to `update()` function

Severity

Impact: Medium

Likelihood: High

Description

The `GigaNameNFTBeforeUpdateHandler` and `GigaNoobNFTBeforeUpdateHandler` handler contracts are called by the `GameNFT._update()` function when **GigaNameNFT** or **GigaNoobNFT** tokens are minted, transferred, or burnt, where these handler contracts rely on the `GAME_LOGIC_CONTRACT_ROLE` granted in the `AccountSystem` contract to store the new owner data and remove the old owner data from the datastore. However, the `update()` function in these handler contracts isn't protected and can be accessed by anyone, which allows anyone to call the `update()` function on tokens that have already been minted, and removing the original owner's record from the datastore and setting a new owner, which will result in the new owner not being able to mint a **GigaNoobNFT** via **AccountSystem.mintWithEth()**, as the minting process will revert with the error: "**User already has an account**"

```
function update(
        address tokenContract,
        address to,
        uint256 tokenId,
        address //auth
    ) external override {
        //...
    }
```

Recommendations

Restrict the access of the `GigaNameNFTBeforeUpdateHandler.update()` function to the `GigaNameNFT` contract.

[H-03] Soulbound tokens cannot be minted or burnt due to an invalid check

Severity

Impact: Medium

Likelihood: High

Description

The `GameNFT` contract is inherited by `GigaNoobNFT` and `GigaNameNFT` contracts, and when a token is minted, transferred, or burnt, the inherited `_update()` function (that overrides the `ERC721._update()`) is called, however, the function contains an invalid check when determining if a token is a soulbound, if a soulbound token is being minted, the check incorrectly prevents the minting/burning processes from completing:

```
function _update(
        address to,
        uint256 tokenId,
        address auth
    )
        internal
        virtual
        override(
            ERC721
        ) returns (address)
    {

         if (beforeUpdateHandler != address(0)) {
            IERC721UpdateHandler(
                    beforeUpdateHandler
                ).update(
                address(this),
                to,
                tokenId,
                auth
            );
        }

        address prevOwner = _ownerOf(tokenId);

        //...
        bool isSoulbound = getDocBoolValue(tokenId, IS_SOULBOUND_CID);
        require(!isSoulbound, "GameNFT: Token is soulbound");
        //...
    }
```

Recommendations

Update the `GameNFT._update()` function to add the following checks:

- when minting a token: verify that the previous owner is `address(0)` (indicating no prior owner).
- when burning a token: ensure that the recipient address (to) is `address(0)` (indicating the token is being burnt).

[H-04] Stuck ETH in `GigaNameNFT` contract

Severity

Impact: Medium

Likelihood: High

Description

The `GigaNameNFT` contract is supposed to receive ETH when the minter role calls `mintUsername()` to mint a **GigaNameNFT** for a player, however, this contract doesn't implement a mechanism to enable the manager from withdrawing collected ETH (nor does the parent `GameNFT` contract), which would result in permanently locking the collected ETH in the contract.

Recommendations

Update `GigaNameNFT` contract to implement a withdrawal mechanism for the collected ETH.

[H-05] Zero Username price when it is minted with `mintUsername()`

Severity

Impact: High

Likelihood: Medium

Description

The `AccountSystem` contract allows users to mint new accounts by paying Ether through the `mintWithEth` function. This function correctly enforces the payment requirement by checking the mint price stored in the `ETH_MINT_PRICE_CID` column of the `AccountSystem` table. Manager can set this price using the `setMintPrice` function in the AccountSystem contract.

However, there is an alternative path to mint an account for free by calling the `mintUsername` function in the `GigaNameNFT` contract. This function also checks the mint price against the `ETH_MINT_PRICE_CID` value, but it references the `GigaNameNFT` table, which has the value set to zero by default. Additionally, there is no function in the `GigaNameNFT` contract to set the mint price in its table, leaving it perpetually zero.

```
function mintUsername(address to, string memory username) external payable {
        require(msg.value >= getTableUint256Value
          (ETH_MINT_PRICE_CID), "Insufficient payment");
        confirmMint(to, username);
    }
```

Recommendations

Ensure that both the `AccountSystem` and `GigaNameNFT` contracts reference the same price in the AccountSystem table.

8.2. Medium Findings

[M-01] Missing cleanup of certain registered data when burning an NFT

Severity

Impact: Medium

Likelihood: Medium

Description

For `GigaNameNFT`, when minting, `IS_GIGA_NAME_CID`, `NAME_CID`, and `INITIALIZED_CID` are set. However, these data entries are not removed upon burning the NFT.

```
function _initializeTraits(
      uint256tokenId,
      stringmemoryusername
    ) internal nonReentrant onlyRole(GAME_LOGIC_CONTRACT_ROLE
        require(validateUsername(username), "Invalid username");
        require(tokenId == uint256(keccak256(abi.encodePacked
          (username))), "Token ID does not match name");

        _setDocBoolValue(tokenId, IS_GIGA_NAME_CID, true);
        validateUsername(username);
        _setDocStringValue(tokenId, NAME_CID, username);
    }
```

```
function _safeMint
      (address to, uint256 tokenId, bytes memory data) internal override {
        ...
        // Conditionally initialize traits
        if (getDocBoolValue(tokenId, INITIALIZED_CID) == false) {
            _initializeTraits(tokenId);
            _setDocBoolValue(tokenId, INITIALIZED_CID, true);
        }
    }
```

For `GigaNoobNFT`, during minting, `IS_NOOB_CID`, `LEVEL_CID`, and `INITIALIZED_CID` are set. However, these data entries are not removed upon burning the NFT.

```
function _initializeTraits(uint256 tokenId) internal override {
        _setDocBoolValue(tokenId, IS_NOOB_CID, true);
        _setDocUint256Value(tokenId, LEVEL_CID, 1);
    }
```

Recommendations

It is recommended to clear the relevant data set during minting in the table when destroying.

[M-02] `mintUsername()` cannot be called by the minter role only

Severity

Impact: Medium

Likelihood: Medium

Description

The `GigaNameNFT.mintUsername()` function is meant to be called by the `MINTER_ROLE` (as checked by the `confirmMint()`) to mint a `GigaNameNFT` for a player, however, the call will fail if the caller doesn't hold `GAME_LOGIC_CONTRACT_ROLE` as the `_initializeTraits()` is only accessible o addresses with that role, and by knowing that `GAME_LOGIC_CONTRACT_ROLE` is only granted for the system contracts; this will result in the `mintUsername()` function being by EOAs:

```
function mintUsername(address to, string memory username) external payable {
        require(
            msg.value >= getTableUint256Value(ETH_MINT_PRICE_CID),
            "Insufficient payment"
        );
        confirmMint(to, username);
    }
```

```
function confirmMint(
        address to,
        string memory username
    ) public onlyRole(MINTER_ROLE) returns (uint256) {
        uint256 tokenId = uint256(keccak256(abi.encodePacked(username)));
        _initializeTraits(tokenId, username);
        //...
    }
```

```
function _initializeTraits(
        uint256 tokenId,
        string memory username
    ) internal nonReentrant onlyRole(GAME_LOGIC_CONTRACT_ROLE) {
        //...
    }
```

Recommendations

Since the `AccountSystem` contract holds both the `MINTER_ROLE` & `GAME_LOGIC_CONTRACT_ROLE`; then update `AccountSystem` to implement the `mintUsername()` function, where players will be able to mint their own **GigaNameNFT**.

[M-03] Max supply enforcement issues

Severity

Impact: Medium

Likelihood: Medium

Description

The `GigaNameNFT` contract currently allows unlimited minting due to two key issues:

1. `confirmMint` lacks a `maxSupply` check: This function does not validate the total number of minted tokens against a supply limit, allowing infinite token creation.

2. `mint` function calls `GameNFT._safeMint` which assumes token IDs are sequential and represent the total supply, but this GigaNameNFT uses non-sequential IDs generated by hashing the username.

```
function _safeMint
      (address to, uint256 tokenId, bytes memory data) internal override {
        uint256 _maxSupply = maxSupply();
        if (_maxSupply != 0 && tokenId > _maxSupply) {
            revert TokenIdExceedsMaxSupply();
        }
```

3. `GameNFT._safeMint` does not consider burned tokens for calculating the total supply

While the `maxSupply` is currently set to 0 (infinite supply), this is a potential issue for future versions where a finite `maxSupply` might be introduced. If not addressed, these issues could lead to over-minting or incorrect enforcement of supply limits.

Recommendations

Introduce a `totalSupply` counter to accurately track minted tokens and enforce `maxSupply` in both `confirmMint` and `_safeMint` without assuming sequential token IDs and considering burned tokens.

[M-04] Improper address validation in update function

Severity

Impact: Low

Likelihood: High

Description

In the `GigaNameNFTBeforeUpdateHandler` contract, the `update` function does not properly check if the `to` address is `address(0)`, which is critical for handling cases where the NFT is being burned. The code currently attempts to set the username for `to` without verifying if to is `address(0)` This can result in an inconsistent or invalid state in the `AccountSystem`

Recommendations

```
function update(
    address tokenContract,
    address to,
    uint256 tokenId,
    address //auth
) external override {
    IAccountSystem accountSystem = IAccountSystem(_gameRegistry.getSystem
      (ACCOUNT_SYSTEM_ID));
    IGigaNameNFT gigaNameNFT = IGigaNameNFT(tokenContract);

    if (gigaNameNFT.exists(tokenId)) {
        address prevOwner = gigaNameNFT.ownerOf(tokenId);
        if (prevOwner != address(0) && accountSystem.getPlayerUsernameId
          (prevOwner) == tokenId) {
            accountSystem.removeUsername(prevOwner);
        }
    }

    // Add a check to ensure `to` is not the zero address
    if (to != address(0)) {
        if (accountSystem.getPlayerUsernameId(to) == 0) {
            accountSystem.setUsername(to, tokenId);
        }
    }
}
```

[M-05] `mintWithEth()` and `mintUsername()` does not return extra ETH

Severity

Impact: Medium

Likelihood: Medium

Description

The `mintWithEth` function in `AccountSystem` contract, allows users to mint by paying Ether. However, if the user sends more Ether than the required `mintPrice`, the contract does not refund the excess amount. This results in users losing the extra Ether unintentionally.

```
function mintWithEth
      (string memory _username) override external payable whenNotPaused {
        uint256 _mintPrice = mintPrice();
        require(_mintPrice > 0, "Mint price not set");
        require(msg.value >= _mintPrice, "Insufficient payment");
        _mint(msg.sender, _username);
    }
```

The same is true in `GigaNameNFT.mintUsername()` function.

Recommendations

Refund the excess ether to the user in both functions. For example in `mintWithEth`:

```
function mintWithEth(string memory _username) external payable whenNotPaused {
    uint256 _mintPrice = mintPrice();
    require(_mintPrice > 0, "Mint price not set");
    require(msg.value >= _mintPrice, "Insufficient payment");

    _mint(msg.sender, _username);

    uint256 excess = msg.value - _mintPrice; // Calculate excess Ether
    if (excess > 0) {
        payable(msg.sender).transfer(excess); // Refund excess Ether
    }
}
```

Or, require a strict `msg.value` amount:

```
function mintWithEth
      (string memory _username) override external payable whenNotPaused {
        uint256 _mintPrice = mintPrice();
        require(_mintPrice > 0, "Mint price not set");
-       require(msg.value >= _mintPrice, "Insufficient payment");
+       require(msg.value == _mintPrice, "Insufficient payment");
        _mint(msg.sender, _username);
    }
```

```
function mintUsername(address to, string memory username) external payable {
        require(
-           msg.value >= getTableUint256Value(ETH_MINT_PRICE_CID),
+           msg.value == getTableUint256Value(ETH_MINT_PRICE_CID),
            "Insufficient payment"
        );
        confirmMint(to, username);
    }
```

[M-06] `maxSupply` check can be bypassed by `mintBatch`

Severity

Impact: Low

Likelihood: High

Description

The `mint` function internally calls `_safeMint`, which performs a `maxSupply` check to ensure that the total minted amount does not exceed the defined supply limit for the token type. However, the mintBatch function directly calls `_mintBatch` without performing a similar `maxSupply` check. This allows users to bypass the `maxSupply` limitation by using the `mintBatch` function to mint tokens, leading to the potential over-minting of token types.

Recommendations

Modify the `mintBatch` function to include a `maxSupply` check for each token type in the ids array, similar to the logic in `_safeMint`.

8.3. Low Findings

[L-01] `confirmMint()` does not initialize the `INITIALIZED_CID`

The `confirmMint` function calls `_mint` to create the NameNFT.

```
function confirmMint(address to, string memory username) public onlyRole
      (MINTER_ROLE) returns (uint256) {
        uint256 tokenId = uint256(keccak256(abi.encodePacked(username)));
        _initializeTraits(tokenId, username);
        _mint(to, uint256(keccak256(abi.encodePacked(username))));
        return tokenId;
    }
```

However, `INITIALIZED_CID` is only initialized when the `_safeMint` function is called. The `MANAGER_ROLE` needs to call `setTraitsInitialized` separately to set this.

[L-02] Burning Game NFTs will increase the balance of the `address(0)`

The GameNFT contract is the base abstract contract for NFTs in the game, and the overridden `_update` function handles the Soulbound NFT logic.

```
function _update(
        address to,
        uint256 tokenId,
        address auth
    )
        internal
        virtual
        override(
            ERC721
        ) returns (address)
    {
        ...
        address result = super._update(to, tokenId, auth);
        _incrementAmount(getAccountKey(to), BALANCE_CID, 1);
        ...
    }
```

However, when burning the NFT, the `_incrementAmount` call will unexpectedly increase the balance of address(0) in the `dataStore`. It is recommended that when the `to` parameter is `address(0)`, the aforementioned logic should not be executed.

[L-03] `burn()` functions are not accessible within the system

`GigaNameNFT.burn()` & `GigaNoobNFT.burn()` functions are intended to be called by the `GAME_LOGIC_CONTRACT_ROLE` to burn players name & noob NFTs, **since this role can only be held by/assigned to the system contracts**; it was noticed that none of the system contracts have these functions implemented to be called by the users:

```
//GigaNoobNFT
   function burn(
        uint256 id
    ) external onlyRole(GAME_LOGIC_CONTRACT_ROLE) whenNotPaused {
        _burn(id);
    }
```

```
//GigaNameNFT
    function burn(
        uint256 id
    ) external onlyRole(GAME_LOGIC_CONTRACT_ROLE) whenNotPaused {
        _burn(id);
    }
```

Recommendation: add functions to the `AccountSystem` contract to enable users to burn their NFTs.

[L-04] Using `_mint()` instead of `_safeMint()`

The `GigaNameNFT.confirmMint()` function uses `_mint()` function instead of `_safeMint()`, which can cause issues if players are interacting with the game via contracts, as `_mint()` doesn't check for ERC721 receiver compliance:

```
function confirmMint(
        address to,
        string memory username
    ) public onlyRole(MINTER_ROLE) returns (uint256) {
        uint256 tokenId = uint256(keccak256(abi.encodePacked(username)));
        _initializeTraits(tokenId, username);
        _mint(to, uint256(keccak256(abi.encodePacked(username))));
        return tokenId;
    }
```

Recommendation: replace the call to `_mint()` with `_safeMint()` to ensure compatibility with ERC721 standards and proper handling of token transfers, specially when the recipient is a contract.

[L-05] DOS username or account creation

A malicious user can front-run the `mintWithEth` and `mintWithGameItem` functions to take a username, preventing legitimate users from registering an account with that username.

Note: The username is hashed into a token ID, and ERC721 tokens do not allow the same token ID to be minted again.

[L-06] `mint()` does not validate `tokenId` against `username` rules

The `GigaNameNFT.mint()` function allows a `MINTER_ROLE` holder to mint a `GigaNameNFT` with a precalculated tokenId based on the username, however, it doesn't validate if the `tokenId`, derived from the `username`, adheres to the constraints enforced by the `validateUsername()` function, potentially allowing invalid usernames to be minted:

```
function mint(
        address to,
        uint256 id
    ) external onlyRole(MINTER_ROLE) whenNotPaused {
        _safeMint(to, id);
    }
```

Update the `GigaNameNFT.mint()` function to validate the `tokenId`, derived from the `username`, meets the required `username` constraints via `validateUsername()` function.