Security Audit Report for DigiAvatar Contracts

Date: Nov 29, 2021

Version: 1.0

Contact: contact@blocksecteam.com

Contents

Report Manifest

Item	Description
Client	DGN Game Pte.
Target	DigiAvatar Contracts

Version History

Version	Date	Description
1.0	Nov 29, 2021	First Release

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

The target contract is DigiAvatar Contracts.

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The files that are audited in this report include the following ones.

Repo Name	Etherscan URL
DigiAvatar	https://etherscan.io/address/0x3f35d80eaff9dc376cf19eaf7035e1284f0e5d7d#code

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report do not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.

- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- Reentrancy
- DoS
- Access control
- Data handling and data Flow
- Exception handling
- Untrusted external call and control flow
- Initialization consistency
- Events operation
- Error-prone randomness
- Improper use of the proxy system

1.3.2 DeFi Security

- Semantic consistency
- Functionality consistency
- Access control
- Business logic
- Token operation
- Emergency mechanism
- Oracle security
- Whitelist and blacklist
- Economic impact
- Batch transfer

1.3.3 NFT Security

- Duplicated item
- Verification of the token receiver
- Off-chain metadata security

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [1] and Common Weakness Enumeration [2]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[2] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find one potential issues, three recommendations in DigiAvatar Contracts, as follows:

- Low Risk: 1
- Recommendation: 3

ID	Severity	Description	Category
1	Low	*Potential Flawed Implementation of the* `_isContract` *function*	Software Security
2	-	*Using Parameters Rather Than the Hardcode Numbers*	Recommendation
3	-	*Avoiding Duplicated Checking*	Recommendation
4	-	*Adopting a Decentralized Mechanism to Manage the Funds*	Recommendation

The details are provided in the following sections.

2.1 Software Security

2.1.1 Potential Flawed Implementation of the `_isContract` function

Status Confirmed. However, it is not a big issue for the current use. As stated by the developers: 1) it is not worth launching such an attack due to the gas price of Ethereum; 2) the mint processing is expected to be completed in a short time window.

Description The implementation of the `_isContract` function will return `true` if the "account" address points to a contract. However, returning `false` does NOT means it is an externally-owned account (EOA). Specifically, a contract in the construction will lead to the return of `false`, which has been abused to launch attacks in the wild.

```
303  function _isContract(address account) internal view returns (bool) {
304    uint256 size;
305    assembly {
306      size := extcodesize(account)
307    }
308    return size > 0;
309  }
```

Listing 2.1: _isContract:DigiAvatar.sol

Impact In brief, a malicious user could bypass this function and launch potential attacks.

Suggestion Seek more reliable and secure approaches.

2.2 Additional Recommendation

2.2.1 Using Parameters Rather Than the Hardcode Numbers

Status Confirmed. As the mint processing on Ethereum has already been completed, it will be modified in the next version on other chains.

Description The assignment of the variable `publicMintStartTime` (line 40) is hardcoded.

```
22  function initialize(
23    string memory name,
24    string memory symbol,
25    string memory baseTokenURI
26  ) public virtual initializer {
27    __Context_init_unchained();
28    __ERC165_init_unchained();
29    __Ownable_init_unchained();
30    __ERC721_init_unchained(name, symbol);
31    __ERC721Enumerable_init_unchained();
32    __ERC721Burnable_init_unchained();
33    __Pausable_init_unchained();
34    __ERC721Pausable_init_unchained();
35
36    _publicTokenIdTracker = 74;
37    _ambassadorTokenIdTracker = 0;
38    _baseTokenURI = baseTokenURI;
39
40    publicMintStartTime = 1637330400;  // 2021-11-19 22:00:00 UTC+8
41  }
```

Listing 2.2: initialize:DigiAvatar.sol

Impact N/A

Suggestion Specify the value as a parameter.

2.2.2 Avoiding Duplicated Checking

Status Confirmed. As the mint processing on Ethereum has already been completed, it will be modified in the next version on other chains.

Description The checking in line 182 is a duplicate with the checking in line 172. Such a duplicate may lead to an unnecessary waste of gas.

```
165    modifier checkMsgValue(uint256 amount) {
166    require(msg.value >= PRICE * amount, "Incorrect price");
167    _;
168  }
169
170  function mintByAmbassador(uint8 gender, bytes32[] memory proof)
171    external
172    checkMsgValue(1)
173    payable
174  {
175    if (block.timestamp >= publicMintStartTime.add(publicMintDuration) && ambassadorMintStartTime
           == 0){
176      ambassadorMintStartTime = publicMintStartTime.add(publicMintDuration);
177    }
178
179    require(gender == 0 || gender == 1, "Gender is 0 or 1");
180    require(ambassadorMintStartTime > 0, "Ambassador Mint has not started");
```

```
181    require(block.timestamp < ambassadorMintStartTime.add(ambassadorMintDuration), "Ambassador
           Mint is over");
182    require(msg.value >= PRICE, "Incorrect price");
183    require(_ambassadorTokenIdTracker < 74, "Exceed max supply");
184    require(_ambassadorMintAmount[_msgSender()] == 0, "Each ambassador address holds up to 1");
185    require(!_isContract(_msgSender()), "Caller cannot be contract");
186
187    bytes32 leaf = keccak256(abi.encodePacked(_msgSender()));
188    require(MerkleProofUpgradeable.verify(proof, _ambassadorMerkleRoot, leaf), "MerkleProof verify
           faild");
189
190    _mint(_msgSender(), _ambassadorTokenIdTracker);
191    _avatarAttributes[_ambassadorTokenIdTracker] = _createAvatarAttributes(gender, 0);
192    _ambassadorTokenIdTracker += 1;
193
194    _ambassadorMintAmount[_msgSender()] = 1;
195
196    emit AmbassadorMintedAmount(_msgSender(), _ambassadorMintAmount[_msgSender()]);
197  }
```

Listing 2.3: DigiAvatar.sol

Impact An unnecessary waste of gas.

Suggestion Remove the duplicated checking.

2.2.3 Adopting a Decentralized Mechanism to Manage the Funds

Status Not an issue. As stated by the developers, the funds are expected to be the sale revenue of the project, which has been acknowledged by the community.

Description

It is suggested that the developers may enforce a decentralized mechanism (e.g., DAO) to manage the funds of the contract.

```
132  function fetchSaleFunds() external onlyOwner {
133    payable(_msgSender()).transfer(address(this).balance);
134  }
```

Listing 2.4: fetchSaleFunds:DigiAvatar.sol

Impact N/A

Suggestion N/A

Chapter 3 Conclusion

In this audit, we have analyzed the business logic, the design, and the implementation of the DigiAvatar Contracts. Overall, the current code base is well structured and implemented. Meanwhile, as previously disclaimed, this report does not give any warranties on discovering all security issues of the smart contracts. We appreciate any constructive feedback or suggestions.