Competitive Security Assessment

TENGOKU

Oct 19th, 2022

Secure3

secure3.io

Summary

TENGOKU is an NFT project that has created an unprecedented metaverse of virtual humans,where every NFT holder will have the ultra-real vitrual human experience.

This report has been prepared for the project to identify issues and vulnerabilities in the smart contract source code. A comprehensive examination with Static Analysis and Manual Review techniques has been performed by Secure3 team. Also, a group of KYC-and-NDA'ed experienced security experts have participated in the Secure3's Competitive Auditing as well to provide extra auditing coverage and scrutiny of the code.

The examination and auditing scope includes:

- Cross checking contract implementation against functionalities described in the documents and white paper disclosed by the project owner.
- Contract Privilege Role Review to provide more clarity on smart contract roles and privilege.
- Using static scanner to analyze smart contracts against common known vulnerabilities patterns.
- Verify the code base is compliant with the most up-to-date industry standards and best practices.
- Comprehensive line-by-line manual code review of the entire codebase by industry experts.

The security assessment resulted in findings that are categorized in three severity levels: Informational, Low, Medium, Critical. For each of the findings we have provided recommendation of a fix or mitigation for security and best practices.

Overview

Project Detail

Project Name	TENGOKU
Platform & Language	Ethereum, Solidity
Codebase	• repo - https://github.com/tengoku-space/contracts • audit commit - 1a1dbc6d40fff9ae2df0a933ad13804a346aa294 • final commit - 48c4b6c58060453a7cb9c44ce7d7c7d2906b949b
Audit Methodology	• Audit Contest • Business Logic and Code Review • Privileged Roles Review • Static Analysis

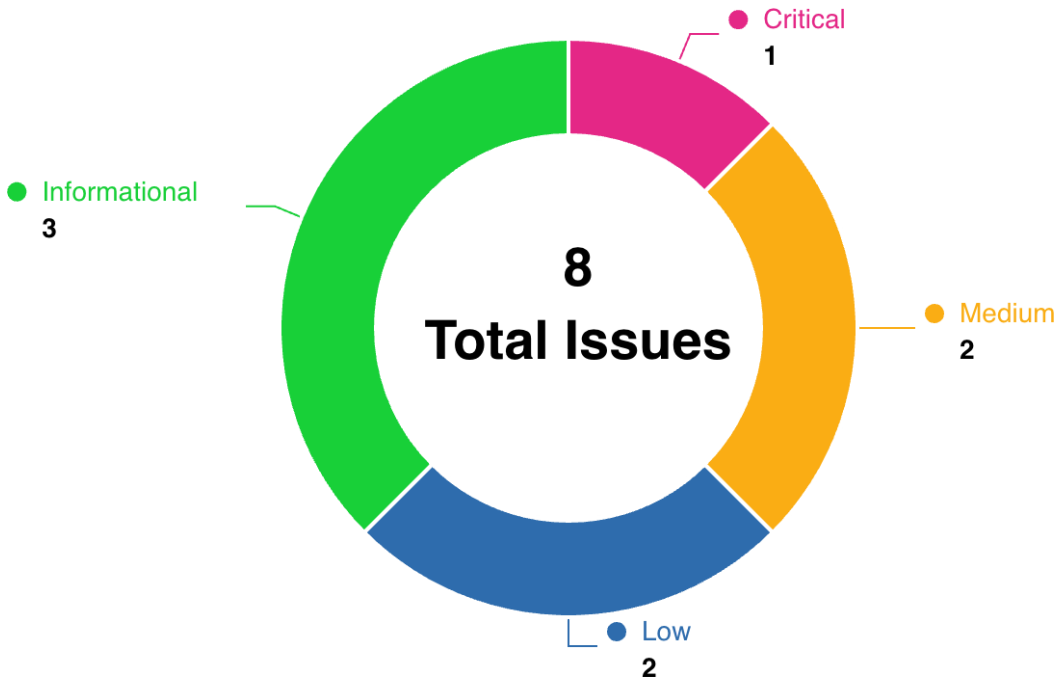
Code Vulnerability Review Summary

Vulnerability Level	Total	Reported	Acknowledged	Fixed	Mitigated	Declined
Critical	1	0	0	1	0	0
Medium	2	0	0	1	1	0
Low	2	0	1	1	0	0
Informational	3	0	0	3	0	0

Audit Scope

File	Commit Hash
contracts/nft/Tengoku.sol	1a1dbc6d40fff9ae2df0a933ad13804a346aa294
contracts/Incubator.sol	1a1dbc6d40fff9ae2df0a933ad13804a346aa294
contracts/nft/TengokuProps.sol	1a1dbc6d40fff9ae2df0a933ad13804a346aa294
contracts/token/BLESS.sol	1a1dbc6d40fff9ae2df0a933ad13804a346aa294
contracts/interface/IIncubator.sol	1a1dbc6d40fff9ae2df0a933ad13804a346aa294
contracts/interface/IInsure.sol	1a1dbc6d40fff9ae2df0a933ad13804a346aa294
contracts/interface/ITengoku.sol	1a1dbc6d40fff9ae2df0a933ad13804a346aa294
contracts/interface/IERC1155Mint.sol	1a1dbc6d40fff9ae2df0a933ad13804a346aa294
contracts/interface/IERC20Mint.sol	1a1dbc6d40fff9ae2df0a933ad13804a346aa294

Code Assessment Findings



ID	Name	Category	Severity	Status	Contributor
TGK-1	`Incubator::claim()` Reentrancy vulnerability leads to signature replay	Reentrancy	Critical	Fixed	hellobloc, calldata
TGK-2	The Problem of Signature Replay for Different Contracts and Chains	Signature Forgery or Replay	Informational	Fixed	hellobloc
TGK-3	Useless `wlMax` Constant	Code Style	Informational	Fixed	hellobloc
TGK-4	`CollectionSize` can be Defined as a Constant	Code Style	Informational	Fixed	hellobloc
TGK-5	Unreasonable Project Schedule Values	Logical	Low	Acknowledged	hellobloc
TGK-6	`Tengoku.sol::batchMints` can mint only `collectionSize - 1` tokens.	Logical	Low	Fixed	calldata
TGK-7	`DEFAULT_ADMIN_ROLE` can withdraw all ETH before users get refund	Privilege Related	Medium	Fixed	hellobloc, Secure3
TGK-8	`MINTER_ROLE` can burn anyone's token and make user un-refundable.	Privilege Related	Medium	Mitigated	calldata

TGK-1:`Incubator::claim()` Reentrancy vulnerability leads to signature replay

Category	Severity	Code Reference	Status	Contributor
Reentrancy	Critical	code/contract/Incubator.sol#100-126	Fixed	hellobloc, calldata

Code

```
100:     function claim(ClaimParams memory params, bytes memory signature)
101:         public
102:         override
103:         whenNotPaused
104:     {
105:         require(
106:             !holderClaimNonces[params.holder][params.nonce],
107:             "WareHourse: already claimed"
108:         );
109:         bytes32 hash = ECDSA.toEthSignedMessageHash(hashMessage(params));
110:         require(
111:             SignatureChecker.isValidSignatureNow(validator, hash, signature),
112:             "WareHourse: Invalid signature"
113:         );
114:         if (params.claimAmount > 0) {
115:             token.mint(params.holder, params.claimAmount);
116:         }
117:         if (params.propTokenIds.length > 0) {
118:             props.mintBatch(
119:                 params.holder,
120:                 params.propTokenIds,
121:                 params.propAmounts
122:             );
123:         }
124:         holderClaimNonces[params.holder][params.nonce] = true;
125:         emit ClaimToken(params);
126:     }
```

Description

hellobloc : To prevent the risk of signature replay, `Tengoku` uses `mark` operation to mark the content of already used signatures, thus preventing signature replay attacks to some extent.
However, this anti-replay measure can lead to signature replay attacks due to the reentrancy problem of the `mintBatch` method in ERC1155.

Exploit

1. Listen to Mempool transactions until a `claim` transaction appears for the `to` address you control
2. Immediately front-running to create an re-entry attack contract which have the `claim` 's signature information in `to` address.
3. Reentry to `claim` function.
4. Eventually you can mint a lot more `token` than expected

calldata : `Incubator.sol#claim` function doesn't follw the "Check-Effect-Interaction" design pattern which leads to it can be reentered. Storage variable checked before calling `props.mintBatch` in code/contracts/Incubator.sol#L105-L108 and modified after `props.mintBatch` in code/contracts/Incubator.sol#L124.
`props.mintBatch` call openzeppelin `_mintBatch` for ERC1155. Let's have a look at the implmentation of `_mintBatch` in openzeppelin ERC1155:

```
function _mintBatch(
        address to,
        uint256[] memory ids,
        uint256[] memory amounts,
        bytes memory data
    ) internal virtual {
        ....

        _doSafeBatchTransferAcceptanceCheck(operator, address(0), to, ids, amounts, data);
    }

function _doSafeBatchTransferAcceptanceCheck(
        address operator,
        address from,
        address to,
        uint256[] memory ids,
        uint256[] memory amounts,
        bytes memory data
    ) private {
        if (to.isContract()) {
            try IERC1155Receiver(to).onERC1155BatchReceived(operator, from, ids, amounts, data)
returns (
                bytes4 response
            ) {
                if (response != IERC1155Receiver.onERC1155BatchReceived.selector) {
                    revert("ERC1155: ERC1155Receiver rejected tokens");
                }
            } catch Error(string memory reason) {
                revert(reason);
            } catch {
                revert("ERC1155: transfer to non-ERC1155Receiver implementer");
            }
```

```
        }
    }
```

In the `_doSafeBatchTransferAcceptanceCheck` function if the `to`(`params.holder`) address is a smart contract address, the control flow is handled to the `to` address via the `onERC1155BatchReceived` call and the attacker can call `Claim` again in `onERC1155BatchReceived` function, which leads to `params.holder` can mint more than allowed `TengokuProps` tokens.

Recommendation

hellobloc : 1. Use the Checks-Effects-Interactions best practice and make all state changes before calling external contracts. 2. Consider using function modifiers such as Reentrancy Guard to prevent re-entrancy from contract level. **calldata :** use the Checks-Effects-Interactions best practice and make all state changes before calling external contracts. make this line code/contracts/Incubator.sol#L124 before `prosp.mintBatch` is called.

```
    function claim(ClaimParams memory params, bytes memory signature)
        public
        override
        whenNotPaused
    {
        require(
            !holderClaimNonces[params.holder][params.nonce],
            "WareHourse: already claimed"
        );
        holderClaimNonces[params.holder][params.nonce] = true;

        bytes32 hash = ECDSA.toEthSignedMessageHash(hashMessage(params));
        require(
            SignatureChecker.isValidSignatureNow(validator, hash, signature),
            "WareHourse: Invalid signature"
        );
        if (params.claimAmount > 0) {
            token.mint(params.holder, params.claimAmount);
        }
        if (params.propTokenIds.length > 0) {
            props.mintBatch(
                params.holder,
                params.propTokenIds,
                params.propAmounts
            );
        }
        emit ClaimToken(params);
    }
```

Client Response

Fixed. Added `nonReentrant` modifier in the `claim()` function and removed `mintBatch()`.

TGK-2:The Problem of Signature Replay for Different Contracts and Chains

Category	Severity	Code Reference	Status	Contributor
Signature Forgery or Replay	Informational	code/contract/Incubator.sol#109	Fixed	hellobloc

Code

```
109:     bytes32 hash = ECDSA.toEthSignedMessageHash(hashMessage(params));
```

Description

hellobloc : The signed message in the current code lacks the important chainid and address(this) information, which makes the contract vulnerable to replay attacks at different addresses and different chains

Recommendation

hellobloc : We recommend following the recommendations of SWC-121 as follows.
In order to protect against signature replay attacks consider the following recommendations:

- Store every message hash that has been processed by the smart contract. When new messages are received check against the already existing ones and only proceed with the business logic if it's a new message hash.
- Include the address of the contract that processes the message. This ensures that the message can only be used in a single contract.
- Under no circumstances generate the message hash including the signature. The ecrecover function is susceptible to signature malleability (see also SWC-117).

Client Response

Fixed

TGK-3:Useless `wlMax` Constant

Category	Severity	Code Reference	Status	Contributor
Code Style	Informational	code\contracts\nft\Tengoku.sol#40	Fixed	hellobloc

Code

```
40:     uint256 internal constant wlMax = 1;
```

Description

hellobloc : The following constant is defined in the `Tengoku` code, but the constant is not actually used.

```
uint256 internal constant wlMax = 1;
```

This could lead to redundant gas consumption and useless bytecode.

Recommendation

hellobloc : We recommend that project team could confirm the role of this constant and remove the variable if it is useless.

Client Response

Fixed

TGK-4:`CollectionSize` can be Defined as a Constant

Category	Severity	Code Reference	Status	Contributor
Code Style	Informational	code\contracts\nft\Tengoku.sol#72	Fixed	hellobloc

Code

```
72:     collectionSize = 5555;
```

Description

hellobloc : The `collectionSize` variable is set to `5555` in `initialize`, but there is no other code that can modify the value.

```
function initialize() public initializer {
        ...
        collectionSize = 5555;
        ...
    }
```

This means that this variable can be defined as a `constant`.

Recommendation

hellobloc : We recommend that if the project does not intend to change the `collectionSize`, the value can be changed to a `constant`.

Client Response

Fixed

TGK-5:Unreasonable Project Schedule Values

Category	Severity	Code Reference	Status	Contributor
Logical	Low	code\contracts\nft\Tengoku.sol#74-76,343-346	Acknowledged	hellobloc

Code

```
74:     conf.publicBegin = 1665177300;
75:     conf.publicDuration = 1 days;
76:     conf.insureDuration = 30 days;

343:     function isRefundGuaranteeActive() public view override returns (bool) {
344:         return (block.timestamp >= getRefundGuaranteeBeginTime() &&
345:             block.timestamp <= getRefundGuaranteeBeginTime() + 1 days);
346:     }
```

Description

hellobloc : The project exists at four time points `publicBegin`, `publicEnd`, `RefundGuaranteeBeginTime`, and `RefundGuaranteeEndTime`.

```
conf.publicBegin = 1665177300;
conf.publicDuration = 1 days;
conf.insureDuration = 30 days;
...
function isRefundGuaranteeActive() public view override returns (bool) {
return (block.timestamp >= getRefundGuaranteeBeginTime() &&
        block.timestamp <= getRefundGuaranteeBeginTime() + 1 days);
}
```

As of now there are 29 days between `publicEnd` and `RefundGuaranteeBeginTime`, and 1 day between the two `Begin` and `End` time points.

And unreasonable schedule-related values may lead to the following problems

1. Due to dos attacks based on `Gaslimit` in the block, it may lead to users not being able to perform `mint` and `refund` operations within such a short specified time of 1 day.
2. `publicBegin` time is set to `1665177300` i.e. `2022-10-08`. the current time has exceeded the `publicEnd` time point. This will cause `publicMintBatch` to be unavailable.

Recommendation

hellobloc : We recommend resetting the `publicBegin` value to a reasonable value and increasing the time between `End` and `Begin`.

Client Response

This is by design

TGK-6:`Tengoku.sol::batchMints` can mint only `collectionSize - 1` tokens.

Category	Severity	Code Reference	Status	Contributor
Logical	Low	code/contracts/nft/Tengoku.sol#L181-189	Fixed	calldata

Code

```
181:         uint256 tokenId = _tokenIdCounter.current();
182:         for (uint256 i = 0; i < tos.length; i++) {
183:             for (uint256 j = 0; j < vols[i]; j++) {
184:                 _safeMint(tos[i], tokenId);
185:                 tokenId++;
186:                 _tokenIdCounter.increment();
187:             }
188:         }
189:         require(tokenId < collectionSize, "Tengoku: over max supply");
```

Description

calldata : There is an edge case leads to `batchMints` can only mint `collectionSize` - 1 tokens.
code/contracts/nft/Tengoku.sol#L189

```
require(tokenId < collectionSize, "Tengoku: over max supply");
```

Recommendation

calldata : change it to:

```
require(tokenId <= collectionSize, "Tengoku: over max supply");
```

Client Response

Fixed

TGK-7:DEFAULT_ADMIN_ROLE can withdraw all ETH before users get refund

Category	Severity	Code Reference	Status	Contributor
Privilege Related	Medium	code\contracts\nft\Tengoku.sol#211-216,243-263	Fixed	hellobloc, Secure3

Code

```
211:     function withdraw() external onlyRole(DEFAULT_ADMIN_ROLE) nonReentrant {
212:         (bool success, ) = conf.withdrawTo.call{value: address(this).balance}(
213:             ""
214:         );
215:         require(success, "Tengoku: Transfer failed.");
216:     }

243:     function _beforeTokenTransfer(
244:         address from,
245:         address to,
246:         uint256 tokenId
247:     )
248:         internal
249:         override(ERC721Upgradeable, ERC721EnumerableUpgradeable)
250:         whenNotPaused
251:     {
252:         super._beforeTokenTransfer(from, to, tokenId);
253:         if (from != address(0) && enableRefund[from][tokenId]) {
254:             require(!hasRefunded[tokenId], "Tengoku: Refunded");
255:             hasRefunded[tokenId] = true;
256:             enableRefund[from][tokenId] = false;
257:             uint256 refundPrice = conf.publicPrice;
258:             if (isWl[tokenId]) {
259:                 refundPrice = conf.wlPrice;
260:             }
261:             payable(conf.withdrawTo).transfer(refundPrice);
262:         }
263:     }
```

Description

hellobloc : The current codebase transfers `eth` to `msg.sender` or `withdrawTo` accounts for the`enableRefund` token's first transfer and refund operations.

```
function _beforeTokenTransfer(
        address from,
        address to,
        uint256 tokenId
    )
        ...
            payable(conf.withdrawTo).transfer(refundPrice);
        }
    }
...
function refund(uint256[] calldata tokenIds) external override {
       ...
            payable(msg.sender).transfer(refundAmount);
    }
```

However, the above design may cause the `Erc721` project to be unavailable in some cases, and users will not be able to perform `refund` and `transfer` operations on the `enableRefund` tokens they own.

As of now, the scenarios that may lead to the unavailability of Erc721 tokens are as follows.

1. The project owner uses `withdraw` to take `eth` that should be used for `refund`, resulting in the user not having any `eth` left for `refund`.
2. The `withdrawto` account is set as an `unpayable` account, so the `transfer` operation cannot succeed because of the failure of `_beforeTokenTransfer`.

Recommendation

hellobloc : We suggest a `multi-signature` account management `withdraw` method and making the contract `payable`.

Secure3 : Add the restriction that the owner can withdraw the tokens only after the user's refund is completed

Client Response

Fixed

TGK-8:MINTER_ROLE can burn anyone's token and make user un-refundable.

Category	Severity	Code Reference	Status	Contributor
Privilege Related	Medium	code/contracts/nft/Tengoku.sol#L218-L227 code/contracts/nft/Tengoku.sol#L243-L263	Mitigated	calldata

Code

```
218:     function burn(uint256 tokenId) public onlyRole(MINTER_ROLE) whenNotPaused {
219:         _burn(tokenId);
220:     }

243:     function _beforeTokenTransfer(
244:         address from,
245:         address to,
246:         uint256 tokenId
247:     )
248:         internal
249:         override(ERC721Upgradeable, ERC721EnumerableUpgradeable)
250:         whenNotPaused
251:     {
252:         super._beforeTokenTransfer(from, to, tokenId);
253:         if (from != address(0) && enableRefund[from][tokenId]) {
254:             require(!hasRefunded[tokenId], "Tengoku: Refunded");
255:             hasRefunded[tokenId] = true;
256:             enableRefund[from][tokenId] = false;
257:             uint256 refundPrice = conf.publicPrice;
258:             if (isWl[tokenId]) {
259:                 refundPrice = conf.wlPrice;
260:             }
261:             payable(conf.withdrawTo).transfer(refundPrice);
262:         }
263:     }
```

Description

calldata : When burning token, there is no check if the `msg.sender` has the `tokenId` either in `burn` function or in `_beforeTokenTransfer` hook, so the `MINTER_ROLE` can burn anyone's token.

```
function burn(uint256 tokenId) public onlyRole(MINTER_ROLE) whenNotPaused {
        _burn(tokenId);
    }

    function _burn(uint256 tokenId)
        internal
        override(ERC721Upgradeable, ERC721URIStorageUpgradeable)
    {
        super._burn(tokenId);
    }

function _beforeTokenTransfer(
        address from,
        address to,
        uint256 tokenId
    )
        internal
        override(ERC721Upgradeable, ERC721EnumerableUpgradeable)
        whenNotPaused
    {
        super._beforeTokenTransfer(from, to, tokenId);
        if (from != address(0) && enableRefund[from][tokenId]) {
            require(!hasRefunded[tokenId], "Tengoku: Refunded");
            hasRefunded[tokenId] = true;
            enableRefund[from][tokenId] = false;
            uint256 refundPrice = conf.publicPrice;
            if (isWl[tokenId]) {
                refundPrice = conf.wlPrice;
            }
            payable(conf.withdrawTo).transfer(refundPrice);
        }
    }
```

`_beforeTokenTransfer` hook is called when burn token. The `from` address is the token owner whose refund right is disabled due to

```
enableRefund[from][tokenId] = false;
```

That is to say, the MINTER_ROLE can make user un-refunable by burn its token and user the fund being tranfered to `conf.withdrawTo` address.

Recommendation

calldata : MINTER_ROLE should only burn its own token

```
function burn(uint256 tokenId) public onlyRole(MINTER_ROLE) whenNotPaused {
        require(ownerOf(tokenId) == msg.sender, "Not token owner");
        _burn(tokenId);
    }
```

Client Response

added check only can burn after the refund end time.

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Invoices, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Invoice. This report provided in connection with the services set forth in the Invoices shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Invoice. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Secure3’s prior written consent in each instance.

This report is not an “endorsement” or “disapproval” of any particular project or team. This report is not an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Secure3 to perform a security assessment. This report does not provide any warranty or guarantee of free of bug of codes analyzed, nor do they provide any indication of the technologies, business model or legal compliancy.

This report should not be used in any way to make decisions around investment or involvement with any particular project. Instead, it represents an extensive assessing process intending to help our customers increase the quality of their code and high-level consistency of implementation and business model, while reducing the risk presented by cryptographic tokens and blockchain technology.

Secure3’s position on the final decisions over blockchain technologies and corresponding associated transactions is that each company and individual are responsible for their own due diligence and continuous security.

The assessment services provided by Secure3 is subject to dependencies and under continuing development. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.